ABSTRACT

ZHANG, QINGHUA. Polymorphic and Metamorphic Malware Detection. (Under the direction of Professor Douglas S. Reeves.)

Software attacks are a serious problem. Conventional anti-malware software expects malicious software, *malware*, to contain fixed and known code. Malware writers have devised methods of concealing or constantly changing their attacks to evade anti-malware software. Two important recent techniques are *polymorphism*, which makes uses of code encryption, and *metamorphism*, which uses a variety of code obfuscation techniques. This dissertation presents three new techniques for detection of these malware.

The first technique is to recognize polymorphic malware that are encrypted and that self-decrypt before launching the attacks in network traffic. We propose a new approach that combines static analysis and instruction emulation techniques to more accurately identify the starting location and instructions of the decryption routine, which is characteristic of such malware, even if self-modifying code is used. This method has been implemented and tested on current polymorphic exploits, including ones generated by state-of-the-art polymorphic engines. All exploits have been detected (i.e., a 100% detection rate), including those for which the decryption routine is dynamically coded or self-modifying. The method has also been tested on benign network traffic and Windows executables. The false positive rates are approximately .0002% and .01% for these two categories, respectively. Running time is approximately linear in the size of the network payload being analyzed and is between 1 and 2 MB/s.

The second technique is a means of recognizing metamorphic malware which has a transformed program image with equivalent or updated functionalities. We propose a new approach that uses fully automated static analysis of executables to summarize and compare program semantics, based primarily on the pattern of library or system functions which are called. This method has been prototyped and evaluated using randomized benchmark programs, instances of known malware program variants, and utility software available in multiple releases. The results demonstrate three important capabilities of the proposed method: (a) it does well at identifying metamorphic variants of common malware. (b) it distinguishes easily between programs that are not related and, (c) it can identify and detect program variations, or code reuse. Such variations can be due to the insertion of malware (such as viruses) into the executable of a host program.

The third technique improves the applicability of a semantic metamorphic malware detector which is the second technique of this dissertation. We propose an automated approach to generate common malware behavior patterns for detection of metamorphic malware or new malware instances. This method combines static analysis and data-mining techniques. This method has been prototyped and evaluated on real world malicious bot software and benign Windows programs. Through the experimental comparison with the metamorphic malware detector, this method results in an about 80% reduction in semantic pattern population to detect known and new malware instances. It is more robust to a junk behavior pollution attack than the malware detector is. A set of experiments was performed to test the quality of the common behavior patterns which were generated with different parameter configurations. Two optimized common behavior patterns were obtained. The corresponding detection rates and true false positive rates are 94%, 8.3%, and 78%, 0.32% respectively.

Polymorphic and Metamorphic Malware Detection

by
Qinghua Zhang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. S. Purushothaman Iyer	Dr. Wenye Wang
Dr. Douglas S. Reeves Chair of Advisory Committee	Dr. Peng Ning

DEDICATION

To my parents and sister. Your love and support passes me the biggest strength.

BIOGRAPHY

Qinghua Zhang was born in Jiang Xi, P.R.China. She received her Bachelor's degree in the school of Computer Science and Technology at Beijing University of Posts and Telecommunications, Beijing, P.R.China in 1999. Since spring 2002, she has been a graduate student in the Department of Computer Science at North Carolina State University (NCSU). She began her doctoral study at NCSU in spring 2004.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Douglas S. Reeves, for his support during this work. His constant guidance, motivation, and constructive criticism have provided a good basis for the present thesis.

I would like to thank Dr. Peng Ning, Dr. S. Purushothaman Iyer, and Dr. Wenye Wang for taking the time to be on my Ph.D. thesis committee, and Dr. Mladen A. Vouk and Dr. Ting Yu for being my Ph.D. qualifying exam committee members. I thank them for providing me insightful comments and valuable feedbacks.

I am very grateful to Dr. Mihail L. Sichitiu for his generosity. I thank Dr. Sichitiu for being willing to substitute Dr. Wenye Wang to attend my Ph.D. final oral defense and being so flexible to allow me successfully schedule my defense.

I would like to thank National Science Foundation (NSF) for its funding support.

I would like to thank Cigital Inc., which offers me a great internship opportunity for three months to work with smart professionals and have valuable experience of working on practical and interesting software security problems.

I would like to thank my friends and colleagues for their help in various ways throughout the entire duration of this thesis. They are (sorted by name) - Vinod Arjun, Keith Irwin, Chongkyung Kil, Brian Lauber, An Liu, Donggang Liu, YoungHee Park, Pai Peng, YoungJune Pyun, Kyuyong Shin, Kun Sun, Pan Wang, Ting Wang, Dingbang Xu, Qing Zhang, Yi Zhang and others whom I may have mistakenly forgotten to mention.

Especially, I would like to give my special thanks to my parents and my sister for their constant encouragement, trust and love that I have relied on throughout my entire time at Academy. I affectionately dedicate this thesis to them. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

The Internet plays an essential role in all areas of society, from the economy to the military to the government. The Internet now connects hundreds of millions of computers, which are vulnerable to attacks by malicious software, or *malware*. Malware exploits vulnerabilities or flaws in software systems and critical applications to intentionally disrupt their use, or to subvert them for other purposes. The rapid increase in malware threatens not only individual computers, but the availability of the Internet itself.

Three trends, including the growth of the Internet *connectivity*, system *extensibility* and *complexity*, contribute to the growth and evolution of this problem [3]. The mono-culture nature of current hardware and software makes it possible to exploit a single vulnerability which will compromise a large number of host computers. The increasing connectivity of computers via high-speed Internet connections increases the visibility of vulnerable systems and exposes them to the attacks.

A typical example of malware is an Internet *worm*, which is self-propagating code. Worms use one or more *remote exploits* to compromise a vulnerable victim host via the Internet. Having done this, the worm launches an identical attack on other vulnerable hosts reachable by the Internet. The automated nature of worms makes them virulent and destructive. According to *Computer Economics* [4], the estimated worldwide damages caused by three well-known worms in 2001 (*Code Red*, *Nimda* and *Slammer*), exceeded $4 billion. The financial loss caused by malware has been as high as $13.3 billion in 2006 [4]. Software extensibility and rapid evolution is a double-edged sword. On the positive side, it satisfies the demands of customers for new features. On the negative side, rapidly shipped but incompletely tested new software releases tend to contain more vulnerabilities that can be exploited. The increase in software complexity also increases the potential number of vulnerabilities and flaws. Fixing these overlooked flaws has led to a constant cycle of necessary software patching; an un-patched system is an easy target for malware.

There are many types of malware besides Internet worms. Examples include viruses, trojans, rootkits, spyware, and spam. Different instances of malware have a variety of penetration methods, malicious purposes, and effects [5]. A malware instance can be transported through remote exploit, by e-mail, over a peer-to-peer network, or through removable media [6]. It can also be automatically downloaded and installed by visiting a web site containing exploit code [7]. The spread of malware instances can be extremely fast, with some infections requiring only a few seconds [8]. As one illustration of the problem, in a study published in 2004 the average survival time for an unpatched Windows machine connected to the Internet was about 20 minutes [9].

Nowadays the goals of those creating and unleashing malware have shifted, from simple vandalism and craving for recognition, to financial gain [10, 11, 12]. Malicious attacks have become more organized and purposefully directed. Botnets in particular confirm this trend. Botnets are armies of remotely-control computers, or *zombies*. These computers are compromised and then infected with software robots, or *bots*, that allow the zombie computers to be controlled remotely through established command and control channels (C&C). Collectively, under the control of C&C servers, botnets become powerful and effective slave computing assets that can be rented for illegal activities. Such activities include phishing attacks, installing backdoors or rootkits on host systems to obtain private information, sending spam for advertising, and launching large scale distributed denial-of-service (DDoS) attacks [13, 14, 15].

In summary, malware has become a major threat, and there is substantial financial incentive for such malware to continue to develop. It is necessary and important to fight this trend.

1.1 Motivation

This dissertation focuses on the research problem of malware detection, which is generally considered as the first step in the malware defense. With proper identification of malware, it is possible to defend against infection. Unfortunately, there are multiple reasons why it is unlikely that one identification method will be universally effective. Those reasons include:

- *Miscellaneous types and penetration methods:* As briefly introduced, malware consists of many types and with many penetration methods. A compromised system displays different symptoms when infected with different types of malware instances. For example, upon the release of a new Internet worm, a compromised machine will instantly send out packets to probe other vulnerable machines and infect them. This quick action is important for the malware to reach a large percentage of vulnerable hosts before a detection approach is developed. The sharp burst of similar network packets observed during worm spreading can trigger a quick mitigation response [16]. Other malware instances, such as bots, do not necessarily launch new infection attempts immediately. Most of the time, they silently lurk in the compromised machines and wait for external commands to perform malicious actions [17]. As another example, there are numerous ways to acquire private information, including the installation of keyloggers, or the launching of phishing attacks to lure victims to visit fake web sites to capture personal information.

- *No source code:* It is generally impractical to acquire malware source code for analysis purposes. Many times it is only possible to obtain the binary executables of malware, through methods such as honeypot trapping [13, 18]. Static analysis of the malware binaries, or emulation of malware execution is generally used to study the malware characteristics. However, both static and dynamic analysis of executables have limitations, as will be discussed in Chapter 2.

- *Rapid infection:* The spread of malware instances can be extremely fast, with global infection potentially taking only a few minutes [8]. It takes about 20 minutes for an unpatched Windows machine connected to the Internet to get infected [9]. No host computer system today has been shown to be malware-proof for a relatively long time. Therefore, it is highly desirable to have an automated malware detection system, as manual identification will be ineffective and/or burdensome.

- *Advanced self-defense approaches:* Malware writers have invented a variety of *self-defense* techniques that are are crucial to the success of these attacks. These techniques aim to hinder malware analysis, and can be classified in a variety of ways. Figure 1.1 shows a classification due to [1].

- *Numerous malware mutants:* It is also quite common that new malware variants (or *mutants*) rapidly evolve from old malware, to which new functions have been added or in which existing functionalities have been tweaked [19]. For example, the VX Heavens website [20] provides access to thousands of malware variants in a variety of different categories. For each malware variant, a signature may be identified, packaged, and downloaded to the base of users expecting protection from the new attack. The huge range of possible variants, and the speed with which they appear, makes manual creation of signatures less and less a practical approach.

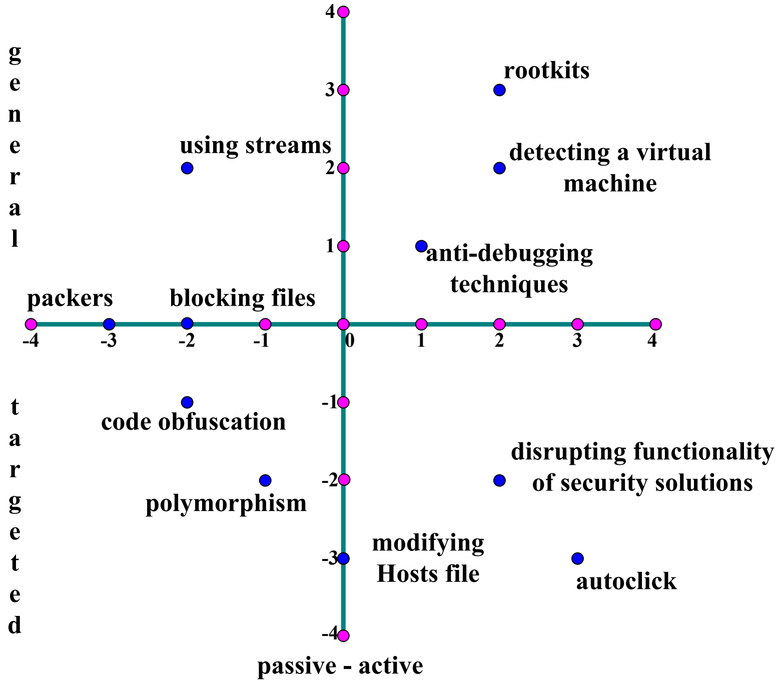


Figure 1.1: Malware Self-Defense Technologies [1]

In this dissertation, I study multiple research problems related to the malware self-defense techniques of *polymorphism* (self-decrypting malicious code) and *metamorphism* (automated code obfuscation). These techniques attempt to bypass the most popular malware detection method, which is based on fixed code *signatures.* We have proposed three methods. These methods are fully automated, and rely only on the availability of binary executables (i.e., do not require source code for analysis).

1.1.1 Polymorphic Remote Exploit Code Detection

Remotely-launched software exploits are a common way for attackers to intrude into vulnerable systems and gain control of them. Such exploits often make use of polymorphism.

Conventional signature-based intrusion detection is the most popular approach for detection of such malware. Some well-known and very successful systems include Snort [21] and Bro [22]. They monitor every inbound packet and examine this incoming data for specific signatures of known malware. When such a signature is found the IDS raises an alarm indicating malicious traffic has been found, and then blocks future traffic from this source. According to *Snort.org*, “a signature is defined as any detection method that relies on distinctive marks or characteristics being present in an exploit [21].” A common type of signature consists of substrings, or byte sequences, from the attacking packet's payload. A signature should be detailed enough (consist of enough information) to minimize the likelihood of a “false alarm” (normal, non-malicious traffic that innocently has similar characteristics to an attack). However, the signature should not be short for performance reasons, as network speeds provide very little time for packet analysis.

As mentioned, malware writers have resorted to defensive techniques that degrade the utility of signature-based detection. Two such techniques are polymorphism, which encrypts exploit code and then self-decrypts on download, and metamorphism, which uses code obfuscation to make each mutant seemingly different. With such techniques, no sufficiently long non-varying byte sequence can be found in the exploit code for use as a signature, even though the malware functionality is unchanged. These techniques thereby confound static signature-checking. There are already many toolkits [23, 24, 25] to automatically generate *polymorphic* exploit code. Although self-mutating metamorphic exploits have not been detected in large numbers yet, toolkits for generating metamorphic code are under development [20]. We conjecture self-mutating metamorphic exploits (or worms) will occur eventually, since there are already real metamorphic viruses such as W32/Apparition and W95/Zmist seen in the wild [26].

There has been substantial research to combat such malware self-defense techniques. The first type of research uses static analysis [27, 28, 29, 30] of binary code. This has shown promise in detecting a specific and non-trivial class of exploits. These exploits target buffer overflow vulnerabilities, accounting for more than 20% of the vulnerabilities reported by CVE [31]. This class of exploits contains *program-like* code that has distinctive control flow and data flow characteristics that can be detected. Figure 1.2 shows a characteristic structure of polymorphic exploits produced by one of the available exploit toolkits [23, 24, 25].

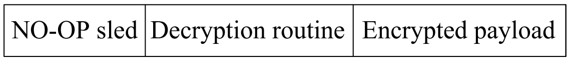


Figure 1.2: A typical structure of a polymorphic exploit of a buffer overflow vulnerability.

The effectiveness of static analysis depends on how well the program-like payload (e.g., the decryption routines) can be distinguished from non-code data, and from non-exploit code. There are several significant challenges. First, exploit code is often hidden inside network traffic, at a non-obvious starting location [27]. Second, the exploit code (instructions) may be interspersed or intermingled with data (whether valid or not) [27]. Unfortunately, it has been shown that most bytes of data can be disassembled into legitimate instructions in Intel assembly code [27]. Hence, code bytes cannot be distinguished from data bytes solely by the use of disassembly. Third, exploit code that is "visible" (i.e., that is not encrypted) is usually manually crafted and does not follow the same programming conventions as executable programs generated by a compiler. For instance, compiled code rarely makes use of overlapping instructions, or of self-modifying instructions. Exploits may do so for the express purpose of defeating static disassembly.

Available static analysis approaches [29, 30, 27, 28] are only partially successful. They are not effective on exploit code containing self-modifying and indirect control transfer instructions, partly due to the non-obvious starting location of the exploit code. No previous methods offer a mechanism to clearly identify the starting location of the polymorphic exploit code. Toth and Kruegel [29] and Akritidis et al. [30] proposed methods that look for a NO-OP sled to detect exploits. However, more advanced exploit code may not need to use a NO-OP sled [32, 33]. Chinchani et al. [27]'s approach extracts the control flow of the exploits based on a disassembly technique that is not resilient to data injection attacks. Wang et al. [28] proposed a code abstraction method to distill useful instructions from an instruction sequence to detect exploits. However, the abstraction is based on a dataflow anomaly rule that is easily evaded by some obfuscation techniques. For instance, two instructions referring to the same undefined variable can still be useful. (i.e., they can be used to clear the registers for initialization.) If malware exploits this property, most of the useful instructions of the malicious code would not be detected by the method of Wang et al.

The second research approach is based on emulated instruction execution, or dynamic analysis. Polychronakis et al. [34] proposed a method that uses emulated execution to more effectively identify self-modifying polymorphic exploit code than is possible with static analysis. The execution of potential instruction sequences is emulated to reveal the run-time behavior of polymorphic exploit code. Their approach does not provide a comprehensive mechanism to identify the starting location of polymorphic exploit code in network traffic. There will be too much overhead if all potential starting locations are tried, but simple heuristics for narrowing down the possible starting locations may miss some attacks.

In summary, existing static and dynamic analysis research is not adequate for defeating such code concealment techniques. This dissertation proposes a new approach that combines static and dynamic analysis to address the problem of detecting exploit code within network traffic. The proposed approach is discussed in Chapter 3.

1.1.2 Metamorphic Malware Identification

Malware detection tools such as virus scanners have been the major defense against malware attacks on personal computers. They are useful but often criticized for being overly simplistic, particularly when dealing with unknown malware instances, or variants of known malware [35, 36, 37, 38]. These detectors use signatures and focus only on the identification of characteristic instruction sequences derived from collected malware samples. They lack insight into malware program semantics or malicious behavior. This limitation is easy to exploit by employment of program concealment techniques such as polymorphism and metamorphism [26, 23, 24].

Malware variants (or mutants) that rapidly evolve from existing malware cause difficulties for signature-based detectors. Thousands of malware variants in a variety of different categories can be easily obtained through Internet web sites such as VX Heavens [20, 39]. For each malware variant, a signature may be identified, packaged, and downloaded to a base of users expecting protection from the new attack. The huge range of possible variants, and the speed with which they appear, makes this a less and less practical approach. Zmist is an advanced metamorphic virus that demonstrates a set of polymorphic and metamorphic code writing skills which include entry-point obscuring, random use of an additional polymorphic decryptor, code permutation, and code integration [26].

One recent research proposal used *semantic templates* of malicious behavior (such as the decryption loop in polymorphic malware) for detection of such malware [37]. The templates are generated by studying the common behavior of a set of collected malware instances. However, these templates must be generated manually for each defensive technique.

In summary, to deal with metamorphic malware or new malware mutants, code semantics should be considered to augment the signature generation process. This summary of the programs' semantics should be an automated process. This dissertation proposes a new approach that uses fully automated static analysis of executables to summarize and compare program semantics. This approach is presented in Chapter 4.

1.1.3 Automated Common Malware Behavior Generation

Given such an approach, it is challenging and non-trivial to generate semantic patterns for malware identification. This is due to the overwhelming number of malware types and malware mutants.

The third work in this dissertation proposes an approach to *automatically* discover common malicious program behavior from a set of malware samples. The output of this approach is a general semantic program pattern that occurs in these malware samples. This general pattern can then be used by the malware detection method mentioned above to identify an entire family of malware. Since this pattern is not tailored to a particular malware instance, and represents a set of common behaviors of an entire malware family, it can effectively overcomes the obfuscation techniques utilized by attackers when creating malware variants. Experimental validation of this idea shows a major reduction in the number of patterns needed to identify mutants. In addition, use of common behavior patterns improves the detector's resilience to a type of data attack that deliberately inserts "junk" code (not important to the exploit's intended function). This work is discussed in Chapter 5.

1.2 Summary of Contributions

The contributions of this dissertation are as follows:

- *Polymorphic remote exploits detection*: This dissertation presents a new method to detect, in incoming network packets, encrypted polymorphic exploits that are self-decrypting. The proposed method harnesses static analysis and emulated instruction execution to find the program entry point, and to identify the instructions of the polymorphic exploit code. Previous approaches [34, 27, 28, 29, 30] do not offer such capabilities. In addition, the proposed method can detect polymorphic exploit code that is self-modifying, which has previously defeated static analysis. The proposed method has been implemented and evaluated against real polymorphic exploits produced by Metasploit [25], and also those produced by polymorphic engines [23, 24]. It achieved a 100% detection rate on polymorphic exploits which use statically coded decryption routines. It likewise had a 100% detection rate for decryption routines that are self-modifying. The method was also tested on typical network traffic not containing polymorphic exploits, and on Windows executable files. The false positive rate was .0002% and .01% for these two categories, respectively. We also measured the running time of our non-optimized implementation. The running time was roughly linear in the size of the traffic being analyzed and was between 1 and 2 MB/s.

- *Metamorphic malware identification*: This dissertation proposes a semantic characterization of programs, and a method for using such characterizations as a basis for malware detection. The method is resilient to many commonly-used obfuscation techniques. The proposed method has been implemented and evaluated on actual malware variants, widely-used benchmark programs that have been randomized, and different releases of the GNU binutils programs [40]. The evaluation results demonstrate three important capabilities of the proposed method: (`a`) it shows promise in identifying metamorphic variants of common malware; (`b`) it distinguishes easily between programs that are not related; and, (`c`) it can identify and detect program variations, or code reuse. Such variations can be due to insertion of malware (such as viruses) into the executable of a host program or program revision. Thus, an indirect application of the proposed work is to help localize an occurrence of one fragment of code inside another program.

- *Automated common malware behavior generation*: This dissertation proposes a new approach to *automatically* discover common malicious behavior from a set of malware samples, for detection of metamorphic malware or new malware instances. This method combines static analysis and data-mining techniques. This method has been prototyped and evaluated on real world malicious bot software, and on normal (non-malicious) Windows programs. Through experimental comparison with the metamorphic malware detector, this method results in approximately an 80% reduction in the number of signatures needed to detect known and new malware instances. It is more robust to "junk code" attacks than the metamorphic malware detection technique. A set of experiments was performed to test the quality of the common behavior patterns which were generated, with a variety of parameter configurations. Two optimized common behavior patterns were obtained. For one, the detection and false positive rates were 94% and 8.3%, respectively, while for the other, the detection and false positive rates were 78% and 0.32%.

1.3 Organization of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 describes the problem background, and related work. Chapter 3 presents a new method for detection of polymorphic exploit code in network traffic. Chapter 4 discusses a new method to summarize and compare program semantics for identification of metamorphic malware executables. Chapter 5 presents a new method for automatically generating common malware semantic patterns. Chapter 6 concludes the dissertation and discusses possible directions for future research. The appendix A presents background information on analysis of binary executables, and on common code obfuscation techniques. It also presents an additional example of self-modifying polymorphic exploit code disassembly.

Chapter 2

Background and Related Work

There are seven broad areas which are relevant to this dissertation and overlap somewhat. Significant advances have been made in these areas. We discuss some of them here to put our work in perspective.

2.1 Binary Code Disassembly

Disassembly is the process of recovering a symbolic representation of a program from its binary representation [41]. Disassembling binaries is a difficult task for two primary reasons: variable-length instructions and fundamentally indistinguishable data embedded inside code regions. There are two standard disassembly techniques. The *linear sweep* method, which decodes bytes sequentially, has deficiencies in distinguishing between embedded data and actual instructions. Therefore it can be defeated by data injection attacks and by other attacks, such as the use of overlapping instructions. Figure 3.2(a) shows an example where this is the case. *Recursive traversal*, which decodes bytes by following the control flow of the program, can better deal with such attacks. However, it requires the entry point of the program to be known in advance. Moreover, the target address of a branch instruction cannot always be statically determined by recursive traversal. In this case, linear sweep may discover more valid instructions. Various techniques have been proposed to improve the disassembly coverage or accuracy. Cifuentes et al. [42] used speculative disassembly techniques to improve disassembly coverage. Their approach made certain assumptions on the properties of the machine and the conventions of the programming language or the operating system. Program obfuscation has been used to protect software security. For example, it has been used to protect software content from malicious reverse engineering [43]. Linn and Debray proposed two techniques to foil disassemblers [44]. One is through inserting unreachable junk code into the program and the other is through using a branching function in place of regular call instructions. Kruegel et al. [41] proposed static binary analysis techniques to improve the disassembly success rate for obfuscated binaries. This method uses the program's control flow graph and statistical techniques to correctly identify a large fraction of the program's instructions. These static disassembly methods cannot correctly handle code which contain self-modifying and/or overlapping instructions or in which other static-resilient techniques are used. Additionally, to apply these various techniques to disassemble code buried in network traffic, the starting point of the code should be found first.

2.2 Conventional Anti-Virus Software

Virus scanners may be the most important line of defence against malware. These tools typically rely on signatures that are extracted from malware bodies to detect known malware. Malware signatures are generated in such a way that they are typical of the analyzed malware programs but not likely to be found in benign programs. As malware evolves, the types of signature and the methods that are used for the search of such signatures also evolve. A modern anti-virus software may use many types of signature, some of which are listed as follows - 1) regular expression of code byte sequences [26]; 2) checksum [26]; and 3) statistical distribution of code bytes [45].

Signature databases of virus scanners need to be quickly updated whenever an unknown malware sample is detected in the wild. However, the signature generation for an unknown malware sample is usually a manual process which is laborious and slow. A human expert runs the unknown program in a restricted environment (i.e., a debugger) and analyzes its actions or behaviors.

The above described manual process is inadequate to keep up with the speed of new malware generation. Malware writers have devised various *automated* techniques of concealing or constantly changing their attacks to defeat detection by signatures. One such technique is *polymorphism* which makes uses of code encryption. Early generations of polymorphic viruses used fairly simple encryption schemes in which only the keys were changed from one copy to another while the encryption routines remained the same [26].

More sophisticated polymorphic viruses were developed such that the encryption routines and keys could both be changed, i.e., using code obfuscation techniques that are called *metamorphism* or using multiple encryption layers [26].

Anti-virus software can use signatures to detect a polymorphic virus after the virus program has been decrypted. There are several widely used methods for decrypting polymorphic viruses such as x-ray scanning and emulation [26].

2.3 Static Analysis and Dynamic Analysis

Static analysis and dynamic analysis are two main approaches that are used for analyzing unknown malware.

Static analysis techniques analyze the code of a program without executing it. To perform static analysis on binary code, three steps are usually involved. The binary code is converted into corresponding assembler instructions first. This step is called disassembly. Then, the conclusions about the program behavior can be derived by applying various control flow and data flow analysis techniques. Static analysis can cover the complete program code and examine all possible paths of execution, theoretically. It is usually faster than the dynamic analysis. However, static analysis has deficiencies. Many interesting questions regarding program behaviors or properties are generally undecidable. Attackers can deliberately craft malware that are hard to analyze statically. Particularly, they can make use of various code obfuscation techniques to confuse the disassembly and code analysis [44]. Various techniques, such as overlapping instructions, indirect addressing, and self-modifying code, can thwart static disassembly.

In contrast, dynamic analysis techniques analyze the code of a program by actually executing it. They are robust to the obfuscation techniques and those anti-static-analysis techniques (i.e, self-modifying). Dynamic analysis also has deficiencies. First, it incurs much more overhead than the static analysis. There could a lengthy code sequence that has to be executed to reach conclusions about the code behavior. Second, it only covers a part of all possible program execution paths. Therefore, many important behaviors of the analyzed program cannot be discovered. Third, it is hard to simulate the execution conditions under which the analyzed malware exhibits its malicious behaviors [46]. For instance, a bot program needs to receive control and commands from a bot master to exhibit its malicious behaviors. Fourth, if the code is executed in a virtual machine like

VMWare, there are techniques [47, 48] that can be utilized by the attackers to determine whether the code is running in virtual environment. As a result, the code can be designed to modify its runtime behavior.

2.4 Network-level Polymorphic Exploit Code Detection

There are four major categories of approaches that detect exploit code (or other intrusions) by analyzing the network traffic.

Signature Based. This category of approaches is most widely used for its ease of being developed and deployed. The IDS (Intrusion Detection System) has certain techniques for generating the signatures of detected or known threats and uses them for detection and blocking of future attacks from the same threat. This category of approaches can serve as a *quick* response to a *new emerging* large scale threat when the vulnerability has not been detected or fixed yet. However, in general, this category of methods has a high maintenance cost, in the sense that signature repositories need to be constantly updated as new polymorphic variants are encountered.

Polygraph [49] and Hamsa [50] are examples of the signature-based approach. These methods generate polymorphic worm signatures by finding common invariant content substrings among multiple polymorphic worm samples. This approach requires a network traffic classifier to preselect suspicious traffic for training. The detection and false positive rates depend on the effectiveness of the classifier. If the background "noise" (network traffic not containing exploits, but selected by the classifier) is significant, the accuracy can be significantly reduced. Newsome et al. [38] presented an attack that causes the signature learning approach to fail. Nemean [51] is a method that uses protocol semantics to group similar worm traffic, and that uses machine-learning techniques to generate connection and session level signatures. This approach requires detailed protocol specification for every application protocol. It is also sensitive to background noise.

Wang et al. [52] proposed a packet vaccine mechanism that randomizes address-like strings in packet payloads for fast exploit code signature generation and detection. This mechanism uses a confirmed exploit as a template to generate a number of variations of that exploit. These exploit variants will likely cause an exception in a vulnerable program's process when they attempt to hijack the control flow and expose themselves thereby. This approach detects and filters exploits in a black-box fashion and avoids the expense of tracking the program's execution flow. This approach has a benign side effect that can better characterize software application's vulnerabilities. However, two drawbacks are obvious. First, this approach does not work on polymorphic exploit code which uses encryption to protect the content. Therefore, the address-like strings from the encrypted payload do not make any sense. Second, this approach requires a pre-detected and confirmed exploit code.

Static Analysis Based. This category of approaches [29, 30, 27, 28] uses static analysis as well as binary disassembly techniques to derive the control flow and data flow of the exploit code buried inside the network traffic and distinguish the code from the data. However, due to unknown code starting location, the efficiency of these static approaches is highly affected by various anti-static techniques such as self-modifying, overlapping instructions and indirect branch instructions.

Emulation Based. Polychronakis et al. [34] proposed a method that uses instruction emulation to more effectively identify self-modifying polymorphic exploit code than is possible with static analysis. The execution of potential instruction sequences is emulated to reveal the execution behavior of polymorphic exploit code. Their approach does not provide a comprehensive mechanism to identify the starting location of polymorphic exploit code in network traffic. There will be too much overhead if all potential starting locations are tried, but simple heuristics for narrowing down the possible starting locations may miss some attacks.

Data-Mining Based. Payer et al. [33] combined neural networks with a simple NO-OP sled detector (as used in [29]) to detect exploit code. The neural network has to be carefully trained with negative and positive data sets, which highly affects its detection rates. In practice, high quality training sets may be difficult to obtain and keep updated.

2.5 Obfuscated Malware Detection

The papers reviewed in this section describe host-level detection methods which have access to a complete representation of the malware and the runtime environment for either static or dynamic analysis.

Kruegel et al. [53] proposed a technique based on the structural analysis of binary code by constructing its control flow graph that allows one to identify structural similarities between different worm mutants. The efficiency of this approach depends on how accurately the control flow graph can be recovered because disassembly is a generally hard problem and does not yield 100% accuracy. This approach has limitations with respect to its application on detection of exploits or worms inside network traffic. One critical problem addressed in the paper is to determine which k-subgraphs of the derived control flow graph appear in network streams. This problem can be converted to a canonical graph labeling problem [54] which is as difficult as the graph isomorphism problem [55] and has no known polynomial algorithm [53]. This means that if k is very large the running time for their approach can potentially be quite large, and if k is small the approach tends to generate a high false positive rate. How to determine the value of k during polymorphic worm detection is not addressed in their work. An exploit code is generally small and quite different with a Windows executable program. Therefore, the control flow graph may be very simple without many blocks. This can further invalidate the approach's assumption that there are many connected blocks inside the flow containing worm code.

Christodorescu et al. [56] presented a unique view of malicious code detection as an obfuscation-deobfuscation game. Based on this view, the authors constructed ten obfuscated versions of four viruses and tested the resilience of three commercial virus scanners against code-obfuscation attacks. In this paper, the authors used control flow graph comparison to detect some simple obfuscation techniques easily used by the virus writers.

A recent method named PolyUnpack [57] automatically identifies and extracts the hidden-code bodies of unpacking-executing malware, with knowledge of the instance's static code model. Christodorescu et al. [37] proposed detecting malware through the use of semantic behavior models which are called templates. One of the presented templates models the decryption loop of polymorphic malware. However, this method does not provide a way to model self-modifying decryption loops. How to define a general semantic behavior model remains as an open problem.

Chouchane and Lakhotia [58] proposed using "engine signature" to assist in detecting metamorphic malware. Basically, it evaluates collected forensic evidence from x86 code segments through a code scoring function to get some measure of how likely it is that they have been generated by some known instruction-substituting metamorphic engine. However this method has limitations. It can only deal with *known instruction-substituting* metamorphic engines. There are many, perhaps even infinitely many ways to create metamorphic engines, not necessarily limited to substitution. Moreover, this technique can be defeated by shrinking substitution methods.

2.6 Data Mining Based Program Behavior Modeling

Data mining based approaches [59, 60, 61, 62] have been frequently used to build behavior models for instruction detection purposes. These methods are applied to system calls or network data to learn how to detect new intrusions. However, these methods try to construct benign behavior models and detect deviated behaviors via run-time monitoring. A recent paper [46] developed a novel mining technique to specify malicious program behaviors by mining differences of execution traces between a malware sample and a set of benign programs. However, a general limitation to this dynamic execution based approach is its difficulty in simulating the malicious execution environment to obtain a high coverage of all possible malicious execution traces.

The work by Schultz et al. [63] is closest to our work. It built a framework that uses three different data-mining algorithms (i.e., RIPPER, Naive Bayes, and Multi-Naive Bayes) to train multiple classifiers on a set of malicious and benign executables to detect new malware instances. The training binaries are statically analyzed to extract the properties of using DLLs, strings, and byte sequences of the binaries.

2.7 Other Miscellaneous Malware Defense Mechanisms

Malware has posed a great threat to the Internet's infrastructure. Great efforts have been made to 1) prevent the attacks, 2) detect the malware or attacks, 3) identify the vulnerabilities, and 4) diagnose and recover from an error or crash when an attack succeeds. Each field has achieved significant advances. Some papers from each area are reviewed here.

Prevention. There are many tools which involve compiler analysis and transformation for dynamic prevention of buffer overflow attacks. StackGuard, StackShield, ProPolice, Libsafe, Libverify, and RAD are such tools and are compared in paper [64]. StackGuard places a 'canary word' on the stack between local variables and return address in the function prologue and monitors the return address by checking the integrity of the 'canary word' at the function's epilogue. This method can be bypassed by the corruption of old frame pointers on the stack or local pointer variables. StackShield and RAD save an extra copy of the return address at the function's prologue and check it with the return address on the stack at the epilogue. ProPolice, similar to StackGuard, does local variable reodering so that char buffers are always placed at the bottom (low addresses) and cannot overflow other local variables. Libsafe and Libverify rely on run-time library call interception and checking. They intercept unsafe library function call such as strcpy() and then perform bounds checking on the arguments.

The mono-culture nature of the hardware and software makes it possible to explore a single vulnerability to compromise a large number of hosts. *Address space randomization* is a program obfuscation technique which defeats or at least makes difficult the memory error exploits (such as buffer overflow exploits) [65]. The idea is to load or run the vulnerable program at a randomized address location to make the predication of a targeted vulnerable program's address space difficult so that an attack that succeeds against one victim will likely not succeed against another. However, running time address space randomization only thwarts the attack to a limited degree [66].

Sidiroglou et al. [67] proposed an end-point architecture to *automatically repair* software flaws to counter various attacks. This approach requires information about software source code.

Detection. Various types of techniques are proposed to identify attacks. In this section they are discussed briefly as follows.

Behavior-based malware detection. Various approaches have studied malware (i.e. worms, spyware) network behaviors or host behaviors to come up with corresponding solutions. This type of technique is more robust against polymorphism and metamorphism. However, sometimes the studied behaviors are tailored to some specific malware. Formulating the behavior signatures is currently ad hoc.

Ellis et. al [68] detected worms by looking for certain patterns of worm behaviors in network traffic. They proposed using three behavioral signatures that are common in worm traffic. They observed the following three worm behaviors in network traffic: 1) sending similar data from one machine to the next; 2) tree-like propagation and reconnaissance; and 3) changing a server to a client [68]. This approach appears promising but has some difficulties. First, it is not clear whether sending similar data from one machine to another is often indicative of suspicious activity. In P2P networks, it is a common scenario that ingress/egress traffic exhibit great similarities. Second, the behavioral techniques need to maintain large amounts of state information about network host behaviors. This could be quite expensive in practice.

E. Kirda and C. Kruegel [69] proposed a technique for spyware detection based on the characterization of spyware-like behavior. This approach is tailored to one particular and popular class of spyware applications that use Internet Explorer's Browser Helper Object(BHO) and toolbar interfaces to monitor a user's browsing behavior space. Their approach uses a composition of static and dynamic analysis to determine whether the behavior of BHOs and toolbars in response to simulated browser events should be considered malicious.

Moshchuk et al. [7] proposed an interesting study that examines the amount, the types, the frequencies, and the most dangerous web zones of spyware on the World Wide Web. Their basic approach is to simply crawl the web and then analyze all captured binary with the help of a VM and Ad-Aware as well as web sites containing malicious content that exploit browser vulnerabilities. This work points out there are two common methods that a spyware can use to install itself surreptitiously. The two methods are piggy-backed spyware download by a user with another application and a "drive-by download" attack which exploits a vulnerability in the user's browser to install software without the user's consent when the user visits the malicious web pages. This work finds 13.4% of 21200 executables are identified as spyware and 5.9% of the Web pages processed contain "drive-by download" attacks.

Kruegel et al. [70] used static analysis techniques to detect kernel-level rootkits which exhibit a certain behavior at load-time. These specific rootkits exist within the operating systems which support loadable kernel modules. They operate within the kernel and modify critical data structures such as system tables at load-time.

Wang et al. [71] designed a tool to automatically generate network-level signatures for spyware by correlating user input with network traffic generated by the untrusted program. This method has several fundamental limitations as pointed out by [71].

Binary instrumentation to monitor/sandbox the execution of vulnerable programs. Many current software attacks build on exploits that subvert the intended machine code execution. Therefore, various proposals [72, 73, 74, 75, 76] design techniques to enforce one basic security property: control flow integrity. They do this via binary instrumentation. They insert binary checking code to monitor the execution of the binary program. Due to disassembly limitations and performance consideration, these designs are generally not able to deal with software utilizing self-modifying, runtime code generation, unanticipated dynamic loading of code as well as other obfuscation techniques for protection [74].

Program shepherding proposed by Kiriansky et al. [73] provided three techniques for enforcing a security policy. First, it restricts execution privileges on the basis of code origins to ensure the malicious code masquerading as data never gets the chance to be executed. Second, it restricts control transfer based on instruction class, source, and target through techniques such as forbidding execution of shared library code except through declared entry points and restricting a return instruction targeting the instruction after a call. Third, it guarantees the sandboxing checks will never be bypassed. However, this method cannot detect the overflow within a structure.

Prasad and Chiueh [75] used a binary rewriting approach to augment Win32/Intel Portable Executable binary programs with a return address defense mechanism which protects the integrity of the return address on the stack with a redundant copy. The paper demonstrates safe binary instrumentation in practical cases.

TaintCheck proposed by Newsome and Song [72] used a software emulator to track the tainted application data. Depending on its configuration and policies, TaintCheck performs binary rewriting at run time to check various program behaviors, e.g., use of tainted data as jump targets, use of tainted data as format strings, and use of tainted data as system call arguments in order to prevent exploits at running time. As pointed in this paper, TaintCheck can detect only some of the attacks that use tainted data as system call arguments as some applications require legitimately embedding tainted data in the system call arguments. Moreover, the approach can bring a non-trivial runtime slowdown of 1.5-40 times as pointed out in this work. It requires further effort to address the issues of security coverage, false positive rate, and performance overhead.

Abadi et al. [74] relied on dynamic checks for enforcing control flow integrity (CTI) by machine-code rewriting. CTI instrumentation modifies each source instruction and each possible destination instruction of computed control transfer according to a given control flow graph which is derived from static program analysis. The instrumentation inserts a bit pattern (ID) to identify an equivalence class of destinations and inserts before each source a dynamic ID-check to ensure the runtime destination has the ID of the proper equivalence class. However their approach has three assumptions which are not always valid for complex applications.

W. Li et al. [76] proposed a solution to build a system call-based sandboxing system, called BASS, which can derive a system call model for win32/X86 binaries that involves dynamically linked libraries, multi-threading, and exception handlers. The BASS system call model representation checks three things: system call ordering, system call coordinates (which is defined by the sequence of function calls from the program's main function to the function containing the system call site and the system call site itself), and system call arguments. The three aspects highly improve the accuracy of the system call model, which is frequently used in host-based intrusion detection systems.

Generic vulnerability signatures. Wang et al. [77] proposed a first-line worm defense in the network stack. Their approach is to install network filters on end host systems which match vulnerability signatures to known exploitable vulnerabilities. This approach removes the dilemma regarding applying a not yet fully tested software patch which may be unreliable and disruptive. The method is essentially self-limited. It can only be applied to some well-known vulnerabilities. How to derive a generic vulnerability signature is an open research problem.

Brumley et al. [78] proposed data-flow analysis and adopted existing techniques such as constraint solving for automatic generation of vulnerability signatures. They presented three forms of signatures: Turing machine, symbolic constraint, and regular expression signatures. However, their work is not complete and still has limitations. In their formal definition framework, the vulnerability condition is assumed to be given. This is, however, the most difficult and critical component for the vulnerability signature generation framework. In their framework, there is no algorithm or method to describe how to use the generated vulnerability signature, i.e. for the Turing machine signature.

Newsome et al. [79] proposed two vulnerability-specific execution-based filters(VSEF): taint-based VSEF and destination-based VSEF that achieve better performance than full-blown execution monitoring. Their methods require the vulnerable program's execution trace to identify the instrumentation location of the checking code. For their destination-based VSEF, debugging-table information is needed for identifying the bounds of the input buffer in order to be able to insert the checking code.

Honey Pot. According to SANS.org, "Honey Pot Systems are decoy servers or systems set up to gather information regarding an attacker or intruder into your system. [80]" The hosts in honey pot system have untreated vulnerabilities. Therefore, any unsolicited and malicious inbound activities can be directly observed and saved for further analysis. Consequently, any unsolicited outbound traffic will be a strong indication of infection. Kreibich et al. [81] designed a host based system - Honeycomb, that can generate signatures of intrusions using honeypots. Honeycomb uses the longest common substring (LCS) algorithm to find similarities in packet payloads which are monitored and saved, and generates signatures thereafter. Honeypot systems are passive. They rely on the hosts being infected to generate signatures. Hence they can be too slow to manage worm outbreaks.

Collaborative Method. Vigilante [82] relies on collaborative worm detection at end hosts which can share self-certifying alerts (SCAs) upon worm detections while not necessarily trust each other. SCAs are provided as proofs of vulnerability. Once hosts receive an SCA, they can generate filters to block infection by analyzing the SCA-guided execution of the vulnerable software. This method may suffer from DDOS attacks. It is insufficient to defend against extremely fast worms such as hit-list worms.

Traffic Analysis. Some methods collect network traffic and find the patterns of characteristic malware (i.e., worms, spams, and botnet) for identifying the outbreaks of some large scale attacks.

1. Scan anomaly detection. Using exploits intensively to find vulnerable hosts in the Internet, a worm may result in highly abnormal network traffic in terms of the number, frequency and distribution of scanned addresses and the number of failed connections [83]. Staniford et al. [83] designed a portscan detector that is effective against stealthy scans as well as all the other scans detected by concurrent techniques. The basic idea is to have an anomaly sensor and a correlator. The sensor monitors the traffic and assigns an anomaly score to each event based on packets header fields such as source and destination IP address, source and destination port, protocol and protocol flag. It then passes those sufficiently anomalous events to the correlator.

2. String counting and sifting. Earlybird [84] and Autograph [85] are two systems that automatically derive worm signatures from suspicious network traffic by sifting through the traffic and identifying the most frequently appearing strings as signatures. However, their methods can be easily evaded by obfuscated worms and can cause high false positive rates due to non-worm traffic which exhibits similar traffic frequency patterns similar to worms.

3. Traffic payload anomaly detection and correlation. PAYL [86] is a worm detection and signature generation system that can detect new worms without signatures by using machine learned models of normal network content traffic. It works by firstly identifying anomalous packet payloads and generating alerts, and secondly correlating ingress/egress anomalous payloads to detect a worm propagation and generate signatures. A "normal profile" is pre-established and represents the statistical distribution of the byte values extracted from the training network datagrams. Anomalous packets are detected by comparison with the normal profile. Packets are flagged as anomalous when their "Mahalanobis distance"(see

paper [86] for detail) with the normal file exceeds a threshold. The correlation approach is used to reduce false alarms. The normal file or the *trained model* is site-specific, service port, and length specific. However, this approach can be defeated by polymorphic blending traffic [87]. Furthermore, P2P traffic exhibits a strong ingress/egress correlation that may be falsely flagged as worms.

Garriss et al. [88] proposed a whitelisting system that can automatically populate whitelists by exploiting friend-to-friend relationships among email correspondents in order to filter spam emails. Xie et al. [89] proposed a system DBSpam to detect and break proxy-based email spam laundering activities inside a customer network and trace out the relevant spam sources outside the network. They revealed one characteristic of proxy-based spamming activities, *packet symmetry*, by analyzing protocol semantics and timing causality. DBSpam used a statistical method- Sequential Probability Ratio Test- to detect the occurrence of spam laundering by monitoring the bi-directional traffic passing through a network gateway.

Bug Finding. There has been a great deal of work about bug finding including static analysis tools [90], better type systems [91], software model checking [92], and input generation [93]. Each field has much more papers than that could be listed here. These tools work on source code program and sometimes need to re-compile the program.

Diagnosis and Recovery. Rx [94] is an approach that can quickly recover programs from many types of software bugs by rolling back the program to a recent checkpoint in the event of a software failure and then re-executing the program in a modified environment. This work shows the method is 21-53 times faster than a whole program restart. Rx cannot guarantee recovery from all software failures because neither semantic bugs nor resource leaks can be directly addressed by Rx. Furthermore, a false recovery can be made due to not detecting a failure in time, thus making a false checkpoint.

Smirnov and Chiueh [95] proposed a GCC compiler extension DIRA that transforms a program's source code to enable the resulting program to automatically detect any buffer overflow attack against it, repair the memory damage left by the attack, and identify the attacking packets. DIRA, based on memory update logging, inserts logging code into an application's source code and restores the execution to a nearest checking point. Their approach can only deal with certain buffer overflow attacks that aim to control-hijacking the return addresses or certain types of function pointers.

2.8 Summary

In summary, malware detection is an active research area in which polymorphic and metamorphic malware detection is a hot topic. In the next three chapters we present three new approaches that address the polymorphic and metamorphic malware problems and present the evaluation results. These approaches are fully automated.

Chapter 3

Polymorphic Remote Exploit Code Detection

This chapter introduces the first proposed technique, which is to detect polymorphic exploit code in network traffic through static and dynamic analysis. The proposed method works by scanning the network traffic for the presence of a decryption routine, which is characteristic of such exploits.

3.1 Overview

Attackers frequently use remote exploits to intrude into vulnerable systems and gain control of them. Their remote exploitation techniques are evolving. To evade signature-based detection methods, which are the most popular and widely deployed methods, the attackers come up with various techniques to conceal their attacking traces, ensuring no sufficiently long constant content can be fingerprinted. Encryption is an established technique which is a frequently used concealment method. Exploit code which uses encryption for concealing purpose is called *polymorphic exploit code*.

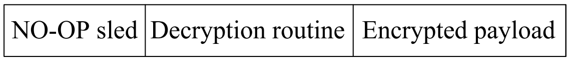


Figure 3.1: A typical structure of polymorphic exploit codes on buffer overflow vulnerabilities

Figure 3.1 shows a typical structure of the polymorphic exploit code. In such a structure, there is encrypted exploit payload which looks like *random data*, and a decryption routine which is normally appended to the encrypted payload such that when the whole exploit is received and executed by the vulnerable program, the encrypted exploit can self-decrypt and fulfill the actual malicious functions. The decryption routine is normally a sequence of manually crafted machine instructions that reverses the effects of the original encryption that is applied to the exploit payload. Since it consists of machine instructions, the decryption routine has characteristic control flow and data flow that distinguish itself from non-code data when disassembled.

We use static and dynamic program analysis to identify and disassemble the executable code (i.e., the decryption routine). There are several research challenges. A polymorphic exploit code is often buried inside the network traffic at a non-obvious starting location and intermingled with non-code data, thereby making detection a difficult job. The non-exploit-code data consist of various parts such as the protocol frame head, the embedded junk data, etc.

The proposed method first locates where the decryption routine is in the network traffic. It then verifies the detected code is a decryption routine by checking whether it satisfies two properties that are typical of such code.

To locate where the decryption routine is, the proposed method first identifies possible starting locations of a decryption routine by looking for a form of *GetPC* code. This is code that the decryption routine uses to get the absolute address of the encrypted payload in the vulnerable program's address space. It then applies recursive traversal from there to find the decryption instructions, looking for a loop in the control flow structure. In addition, the proposed method is enhanced with the capability to dealing with self-modifying decryption routines. It first uses a two-way traversal and backward data-flow analysis to find the functional instructions, i.e. the self-modifying instructions as well as the initialization instructions. It then uses emulated instruction execution of the instructions already found to discover the self-modifying decryption routine.

To verify the detected code is a decryption routine, the proposed approach checks whether it satisfies two properties that we observe from typical decryption routines. These properties have not been used for this purpose previously.

3.2 The Proposed Method

We now describe the method in detail, starting with the decryption routine localization.

3.2.1 Decryption Routine Localization

Generally speaking, a decryption routine is suspected if the control flow structure shows the existence of a loop. Loops are likely to occur in decryption routine for the simple reason that decryption of a sequence of bytes is a very repetitive process.

General Approach

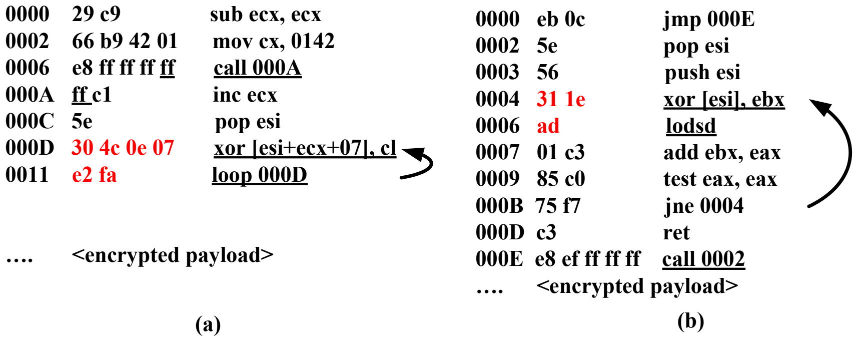
The general approach works by firstly finding the starting point of the decryption routine and then use recursive traversal to find the loop structure of the decryption routine.

Starting Point Localization.
The first step is to find the starting instruction of the decryption routine which is hidden within the network traffic. This routine may be intermingled with bytes of data (non-instructions). The proposed approach is to scan the network packet for candidate *seeding instructions* of *GetPC* code. We now explain seeding instructions and *GetPC* code.

As argued by Polychronakis et al. [34], reliable exploit code should avoid any hard-coded absolute addressing. Therefore, the decryption routine must have some way to dynamically determine the address of the encrypted payload in the vulnerable program's address space, in order to modify it. This is accomplished by *GetPC* code, which computes absolute addresses as offsets from the current value of the program counter. The *GetPC* code should be among the very few instructions that cannot be self-modifying, and it also should be among the very few first functional instructions of a decryption routine. Therefore, detection of *GetPC* code helps localize the start of the decryption routine.

Polychronakis et al. [34] identified two [1] feasible and easy forms of *GetPC* code. One way is through a `call` instruction. Execution of a `call` instruction pushes the return address (the PC) onto the stack. The decryption routine when executed can easily read this return address from the stack. The second way is through a `fnstenv` instruction, which

[1] M. Polychronakis et al. [34] also mentioned a third form of *GetPC* code which is to exploit the structure exception handling(SEH) mechanism of Windows. However they mentioned this technique is not feasible with advanced version of Windows.



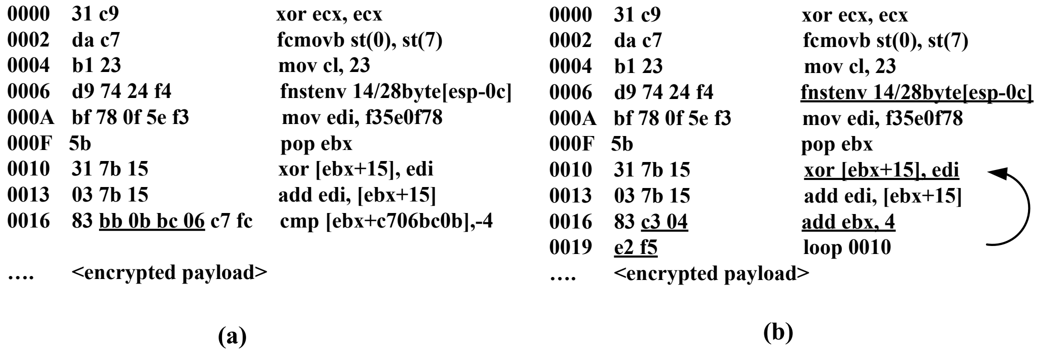
```
0000   29 c9          sub ecx, ecx
0002   66 b9 42 01    mov cx, 0142
0006   e8 ff ff ff ff call 000A
000A   ff c1          inc ecx
000C   5e             pop esi
000D   30 4c 0e 07    xor [esi+ecx+07], cl
0011   e2 fa          loop 000D

....   <encrypted payload>
```

(a)

```
0000  eb 0c          jmp 000E
0002  5e             pop esi
0003  56             push esi
0004  31 1e          xor [esi], ebx
0006  ad             lodsd
0007  01 c3          add ebx, eax
0009  85 c0          test eax, eax
000B  75 f7          jne 0004
000D  c3             ret
000E  e8 ef ff ff ff call 0002
....  <encrypted payload>
```

(b)

Figure 3.2: Disassembly of decryption routines for a) Countdown b) JmpCallAdditive encoders. The underlined instructions are the seeding instruction, the instruction for decrypting the encrypted exploit payload and the instruction for updating the address of encrypted exploit payload. For both examples, a loop structure is presented.



```
0000   31 c9                  xor ecx, ecx
0002   da c7                  fcmovb st(0), st(7)
0004   b1 23                  mov cl, 23
0006   d9 74 24 f4            fnstenv 14/28byte[esp-0c]
000A   bf 78 0f 5e f3         mov edi, f35e0f78
000F   5b                     pop ebx
0010   31 7b 15               xor [ebx+15], edi
0013   03 7b 15               add edi, [ebx+15]
0016   83 bb 0b bc 06 c7 fc   cmp [ebx+c706bc0b],-4
....   <encrypted payload>
```

(a)

```
0000   31 c9            xor ecx, ecx
0002   da c7            fcmovb st(0), st(7)
0004   b1 23            mov cl, 23
0006   d9 74 24 f4      fnstenv 14/28byte[esp-0c]
000A   bf 78 0f 5e f3   mov edi, f35e0f78
000F   5b               pop ebx
0010   31 7b 15         xor [ebx+15], edi
0013   03 7b 15         add edi, [ebx+15]
0016   83 c3 04         add ebx, 4
0019   e2 f5            loop 0010
....   <encrypted payload>
```

(b)

Figure 3.3: Disassembly of Self-Modifying decryption routine for ShikataGaNai encoder. a) Before Execution b) After Execution. The `fnstenv` instruction is the seeding instruction. The `xor [ebx+15], edi]` is the instruction for decrypting the self-modifying decryption routine and encrypted exploit payload. The loop structure is revealed after execution.

stores the current FPU environment, including the preceding FPU instruction pointer, in an area of memory specified by the instruction. This instruction pointer can then be read by a following instruction and, as for the PC, used to compute the absolute address of the encrypted payload. The `call` or `fnstenv`, which are the key instructions for *GetPC* code to work, we term *seeding instructions*. The number of candidate *seeding instructions* in the Intel instruction set is expected to be limited.

By scanning the network packet for the seeding instruction (`call`, `fnstenv` etc.) of *GetPC* code, we can *coarsely* locate where a decryption routine starts.

We scan the network traffic for the occurrence of seeding instructions. Each such instruction found is treated as if it belongs to a decryption routine. We have an enhanced approach to more precisely locate the starting instruction, as well as other decryption routine instructions.

Recursive Traversal To Detect Decryption Loop Structure.
Recursive traversal is a standard disassembly technology. It is robust against data-injection attacks, in which code is interleaved with data. The proposed method uses it to find the control flow structure of the decryption routine. Once a loop is detected during recursive traversal, this is a candidate for a decryption routine. However, a recursive traversal may be hindered by indirect addressing branch instructions, and the loop structure can be hidden by self-modifying code techniques. An enhanced approach can address these two issues. The approach uses (a) two-way traversal and backward data-flow analysis, and (b) a limited instruction emulation.

Enhanced Approach

The enhanced approach deals with the two issues when a self-modifying decryption routine is used and the indirect addressing branch instruction is used.

Two-way Traversal and Backward Data Flow Analysis To Find Decryption Instructions.
The enhanced method uses both forward *and* backward traversal of code from the seeding instruction to find the decryption routine. Backward traversal is needed since the seeding instruction may not be the very first instruction executed by the routine. This analysis step is quick if the seeding instruction is close to the start of the decryption routine. Forward traversal recursively follows the control flow, starting at the seeding instruction, to find the instructions that are data-flow dependent on the *GetPC* code. This includes the instructions directly responsible for data decryption.

Backward traversal uses breadth first search, starting at the seeding instruction, to find the instructions that are not data-flow dependent on the *GetPC* code. This includes the initialization instructions. First the set of instructions that directly reach the seeding instruction at byte offset i of the input network traffic are found. This set will possibly contain branch instructions whose target address is i. The set may also contain non-branching instructions immediately preceding the seeding instruction. Then instructions reaching instructions in this set are found, etc. A branch instruction using indirect addressing is unlikely to appear prior to the seeding instruction in the control flow for the simple reason that the *GetPC* code must be executed first. The same is true for self-modifying instructions.

It must be decided which instruction sequence will actually be executed before the *GetPC* code. Backwards data-flow analysis is used for this purpose. This is a popular technique for program analysis [96]. Backward data-flow analysis is a commonly used technology in program analysis [96]. Here we use it for finding the instructions of the decryption routine. If an instruction of the decryption routine is found, backward slicing is used to find instructions on which it has data-flow dependency (i.e. we follow backwards the $define - use$ chain). The instruction from which the backward data-flow analysis begins is either (a) an instruction that writes to memory, or (b) a branch instruction with indirect addressing. Let this be called the *target instruction*. When the target instruction is (a), it could be an instruction used for decrypting the hidden loop or the encrypted payload. When the target instruction is (b), it could be an instruction to obfuscate the control flow. Either of the instructions is significant. When we use backward slicing, if we find all required variables have been defined till the seeding instruction, then there is no non-*GetPC* decryption routine code exists earlier than the seeding instruction. Otherwise, there must be. To choose which instruction sequence contains these code, we pick one that defines all the rest variables or is the longest of multiple qualified instruction sequences.

Figure 3.3 is an example to illustrate this two-way traversal and backwards data-flow analysis. In 3.3 (a), the instruction at offset `0006` is the seeding instruction. During the forward traversal, the instruction `xor [ebx + 15], edi` is encountered. This is the target instruction, which uses values stored in `ebx` and `edi`. The contents of these two registers are defined by the previous instruction `pop ebx` and `mov edi, f35e0f78`. We find these two instructions by backward data-flow analysis. The instruction `pop ebx` is data-flow dependent on the seeding instruction `fnstenv 14/28byte[esp-0c]` which is data-flow dependent on the previous `fcmovb st(0), st(7)`. The instruction `fcmovb st(0), st(7)` is found by backward traversal, as described.

After constructing a chain of instructions through backward traversal, the execution of instructions in the chain is then emulated, as described below.

Detection of Self-modifying Decryption Routine.

Self-modifying decryption routines are detected by performing emulated execution of the already found decryption instructions. The purpose of this execution is to determine the address to which the target instruction writes a value, or the address to which the target instruction branches, depending on the type of the target instruction. As far as the emulation is concerned, the decryption code of the input network traffic is mapped to a random virtual address space of the vulnerable program that the exploit code targets.

The emulation is limited in the following way. Instruction emulation proceeds until a decryption loop is detected, or an illegal instruction is encountered. If a memory location is modified that is within the emulated address space of the code, this fact is noted. It is evidence for the existence of a decryption routine. If the address of the target instruction branches points to the flow itself, the forward traversal is continued, otherwise it is stopped.

In a favorable situation, instruction emulation only occurs for a small number of instructions. This is because execution ends once a self-modifying decryption loop is uncovered. For a decryption loop not using self-modifying techniques, only one traversal of the loop is needed to stop execution.

3.2.2 Decryption Routine Verification

The previous phase detects the presence of a possible decryption routine by finding the loops in its control flow structure. During the detection of the loop, a form of *GetPC* code should be available to find a pointer to the encrypted payload. The data flow of the detected loop is analyzed to improve the overall accuracy of the method. Two properties of of decryption routines are exploited for this purpose.

The first property is that in a detected loop, there must be a memory store instruction that uses indirect addressing. That is, a register is used to contain an offset that partly identifies the location where data is to be read or written. In addition, the memory address pointers to the input network traffic. IA-32 [2] offers 24 memory addressing modes which can be classified into two categories – the direct and indirect addressing. For direct addressing, the memory operand's address is specified directly in the instruction. For indirect addressing, the memory operand's address is referred through one or two registers w/o a displacement. These registers offer a base address(stored in the base register) w/o an offset value(stored in the index register) w/o relative displacement to them. A memory-store instruction using direct addressing is unlikely to be the instruction that directly modifies the encrypted payload. The hard-coded address easily results a fragile exploit code. (That is why the *GetPC* code is needed). For instance, the instruction at address `000D` in Figure 3.2 (a) is such an example. IA-64 architecture also supports `RIP/EIP`-relative data addressing. The memory address can be referenced through `RIP/EIP` registers.

The second property is that the register holding the address or offset must be updated within the loop. Otherwise the same memory location will be written over and over. In our current prototype, we only look for instructions that will update the register value in predictable and regular ways. For instance, `inc/dec/sub/add` instructions are most favorable for updating the registers. Other instructions, such as string instruction `lods` and loop instruction `loop` may also be used to update the register which holds the address or the offset. Future work will generalize this analysis. A possible way for the attackers to achieve the randomness is using a sequence of `push` instructions to specify the decryption order in the stack. The decryption loop then uses `pop` to get the order and then decode iteratively.

A few implementation details are as follows. Each instruction in a cycle is inspected to determine if it satisfies the first property, according to its opcode and addressing mode. If it is such an instruction, the cycle is cut to create an instruction sequence, with this instruction at the end. Then, other instructions in the sequence are sliced out by checking whether they have a data-flow dependence on the target instruction, using backward slicing.

For example, suppose an unwrapped cycle contains the instruction sequence `inc eax, xchg eax,esi, xor [esi],ebx`. The first two instructions are sliced out because of their effect on register `esi`, used in the final instruction.

For checking the data flow dependency of two instructions, instructions are first converted, through into a semantics-preserving transformation, into an intermediate representation. This is helpful for overcoming code obfuscation techniques used in metamorphic exploits. For instance, a well crafted decryption routine may combine several processing steps into a single instruction. The `loop` and `lodsd` instructions shown in Figure 3.2 are examples.

3.3 Evaluation

A prototype of the proposed method has been implemented, and evaluated under realistic conditions. The results are described below.

3.3.1 Detection Rate

We tested the detection capability of the proposed approach on polymorphic exploits. These exploits were generated by two off-the-shelf polymorphic engines: ADMmutate [23], and Clet [24]. These engines have been used in other research for the same purpose [49, 50, 34, 33]. Also tested were polymorphic exploits generated by the Metasploit Framework [25]. This is a powerful open source framework for the construction and execution of exploits. This framework has also been used in other research [27, 34, 28].

The first experiment was as follows. 10 exploits were downloaded from Milw0rm [97]. For each exploit, 10 polymorphic instances were generated, using the above tools (ADMmutate and Clet). ADMmutate may be the first well-known polymorphic engine. It can generate a simple metamorphic NO-OP sled with one-byte instructions, and a metamorphic decryption routine using several advanced obfuscation techniques. These include using multiple code paths for an operational instruction and inserting non-operational "junk" instructions. Clet can generate a metamorphic NO-OP sled using English words. It also uses "cramming" bytes to make the byte frequency of the resulting polymorphic exploit codes resemble that of normal network traffic.

Each of these exploit instances was then input to the proposed detection method. All 100 instances were successfully identified as exploit code. Both of these polymorphic engines generate encrypted exploit codes with an obvious NO-OP sled of sufficient length, as well as an obvious decryption loop. Previously-proposed detection methods [27, 28, 34, 29, 30] may also be able to detect such exploits. The existence of a sufficiently long NO-OP sled will help them cope with the non-obvious starting location of the decryption routine.

The second experiment simulated remote exploit attacks, using the Metasploit Framework. The target service was an unpacked Windows XP host running the Serv-U ftp server v4.0. Attacks were launched from a Windows host using polymorphic exploits for the following vulnerabilities:

- Serv-U FTPD MDTM Overflow [98]
- Microsoft RPC DCOM MS03-026 [99]
- Microsoft LSASS MSO4-011 Overflow [100]
- Microsoft ASN.1 Library Bitstring Heap Overflow [101]

For each vulnerability, we launched multiple attacks from the Metasploit console interface, using the following encoders (encryption methods):

1. `Pex`
2. `PexFnstenvSub`

3. `PexFnstenvMov`

4. `Countdown`

5. `JmpCallAdditive`

6. `Alpha2`

7. `ShikataGaNai`

These were combined with two NO-OP sled generation methods: `Pex`, and `Opty`. `Pex` generates a NO-OP sled with one-byte instructions. `Opty` generates a NO-OP sled with multiple-byte instructions, as well as a "trampoline" sled, which transfers control using relative addressing directly to the exploit code. The traffic capture tool Ethereal was used to capture the network traffic generated by Metasploit. This traffic was then input to the prototype implementation of the proposed detection method.

The proposed approach successfully detected all of the polymorphic exploits generated using the encoders `Pex`, `PexFnstenvSub`, `PexFnstenvMov`, `Countdown`, and `JmpCallAdditive`. These encoders generate static decryption code with the properties identified in section 3.2.2. They do not employ self-modification of the decryption routine. Figure 3.2, for example, shows the disassembly of the decryption code produced by the `Countdown` and `JmpCallAdditive` encoders. The underlines mark the major functional decryption instructions: the seeding instruction of the *GetPC* code, the memory-writing instruction for decrypting the encoded payload and the instruction for updating the address of encoded byte.

More impressively, the proposed method successfully detected 100% of the exploits generated by the `Alpha2` and `ShikataGaNai` encoders. These methods generate *self-modifying* decryption routines. The decryption loop is changed or patched "on the fly" (during execution) before it is used to decrypt the exploit. For illustration, Figure 3.3 shows the disassembly of the self-modifying decryption code for `ShikataGaNai` encoder. Figure 3.3(a) shows the original decryption routine before execution. Figure 3.3(b) demonstrates the results after execution of the self-modifying decryption routine. The underlined instructions in (b) have the same effects as those shown in Figure 3.2. In addition, the underlined bytes identify the modified instructions before and after execution. In the appendix A, we also present the disassembly results of the self-modifying decryption routine for `Alpha2` encoder to favor interested readers.

The polymorphic exploit code for attacking Serv-U FTPD MDTM Overflow vulnerabilities does not use a NO-OP sled. This has been verified by inspection of the outputs generated under different configurations, and by inspection of the Metasploit source code. The absence of a NO-OP sled will likely defeat several proposed methods which specifically look for the NO-OP sled [29, 30]. Emulation methods (e.g., [34]) are also likely to have problems identifying the start of the decryption routine. One of its heuristic for performance optimization is to skip several bytes (e.g. 50 bytes) after a zero byte is detected at a byte offset. Without the compensation effect of the NO-OP sled, instructions of the decryption routine codes could be missed by the method. Sigfree [28] cannot detect polymorphic exploit codes generated by small-sized decryption routines, such as `Countdown`, as mentioned in [28]. It also cannot detect polymorphic exploits that use self-modifying decryption routines, such as the exploit codes generated by encoder `ShikataGaNai`. The method proposed by Chinchani et al. [27] also cannot detect polymorphic exploits with self-modifying decryption routines.

In summary, the proposed method achieves a 100% detection rate on polymorphic exploit codes, with or without NO-OP sleds, and with or without self-modifying decryption routines. No previous method of static analysis has been able to achieve this. Emulation methods (e.g., [34]) can deal with the polymorphic exploit codes with self-modifying decryption routines. However they are not robust against those without NO-OP sled.

3.3.2 False Positives

We also tested the proposed method on normal (non-exploit) network traffic, and on Windows binary executables. A detection method should indicate in both cases that the traffic does not contain exploits. Indicating otherwise is regarded as a false positive.

We collected network traffic for five days from users in our lab, engaging in normal activities. Most of the traffic was UDP, FTP, HTTP, SSL, and other TCP data packets. Among these packets, the number of FTP and TCP packets containing downloaded executables, video files, and streaming video was significant (>90%). Over 4 million packets were captured, with a total payload size of more than 5 GB.

The data payloads from these packets were extracted and presented to the proposed method for testing, a packet at a time. Most exploits are small (a few tens of bytes) and easily fit within a single packet. (We discuss the limitations of this approach in section

3.4.) A packet incorrectly identified as containing an exploit was a false positive. The false positive rate was calculated as:

(# of falsely identified packets) / (Total # of packets)

Windows executables were also analyzed to determine the ability of the proposed method to distinguish exploit code from non-exploit code. Executables in the C:\windows\system32 directory of a machine running Microsoft Windows XP, service pack 2, were used for this purpose. The total size of these files was around 1 GB. For analysis, we packetized each executable into a sequence of packets, and analyzed each packet separately. The false positive rate was calculated as above.

The results were as follows. The false positive rate was 0.0126% for the case of Windows executables, and 0.0002% for the case of captured network traffic. Only 8 out of more than 4 million packets resulted in false identifications, or alerts. The packet contents were manually inspected to verify that they did not contain exploits.

3.3.3 Processing Cost

We also measured the running time of the core detection algorithm of the proposed approach. The experiments were performed on a machine running Microsoft Windows XP, service pack2, with a Pentium(R)D 3.00GHz CPU, and 2GB of RAM. In these experiments, network packets and Windows executables of various sizes, ranging from several bytes to millions of bytes in length, were analyzed. We used the function `clock`() provided by Microsoft Visual C++ to measure the accumulated running time for core detection procedures and calculate the average value. Figure 3.4 shows the results.

Our non-optimized implementation demonstrates a modest processing speed. The results show almost a linear relationship between the packet size, and running time. The current implementation achieves a speed of roughly 1.5M/s. This method has not been optimized, and substantial speedups should be possible.

3.3.4 Comparison With Previous Work

This subsection summarizes the comparison with previous research proposals. Previous static-analysis approaches [29, 30, 27, 28] are not robust against the exploits which employ static analysis resilient techniques such as self-modifying and indirect control transfer instructions, partly due to the non-obvious starting location of the exploit code. None

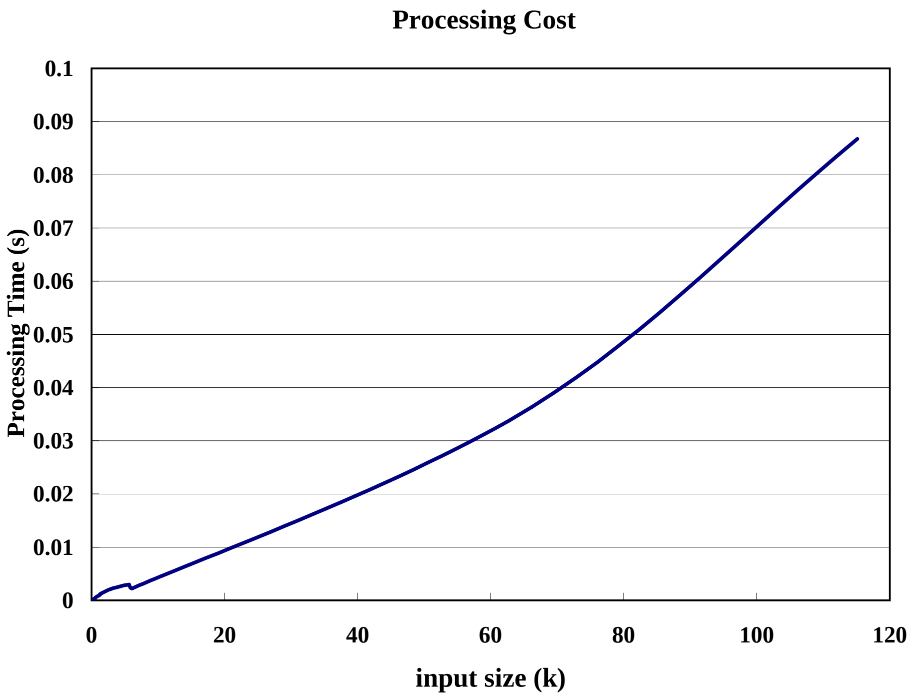


Figure 3.4: Running Time Overhead

of these approaches offer a mechanism to clearly identify the starting location of the polymorphic exploit code. Toth and Kruegel [29] and Akritidis et al. [30] proposed methods that look for a NO-OP sled to detect exploits. However, more advanced exploit code may not need to use a NO-OP sled [32, 33]. The exploit that exploits Serv-U FTPD MDTM Overflow vulnerabilities [98] and is used in our experimental evaluation is such a case.

Chinchani et al. [27]'s approach extracts the control flow of the exploits based on a disassembly technique that is not resilient to data injection attack. It also cannot detect polymorphic exploits with self-modifying decryption routines. Wang et al. [28] proposed a code abstraction method to distill useful instructions from an instruction sequence to detect exploits. However the code abstraction which is based on a data-flow anomaly rule that an instruction referring an undefined variable is deemed useless is easily evaded by some obfuscation techniques. For instance, two instructions referring to the same undefined variable can still be useful, i.e. they can be used to clear the registers for initialization. If the attacker purposely exploits this property, most of their useful instructions would probably not be detected by a chaining effect. This proposal [28] cannot detect polymorphic exploit codes generated by small-sized decryption routines, such as `Countdown`, as mentioned in [28].

It also cannot detect polymorphic exploits that use self-modifying decryption routines, such as the exploit codes generated by encoder `ShikataGaNai`.

The second type of approaches is based on instruction emulation. Polychronakis et al. [34] proposed a method that uses instruction emulation to more effectively identify self-modifying polymorphic exploit code than is possible with static analysis. The execution of potential instruction sequences is emulated to reveal the execution behavior of polymorphic exploit code. Their approach does not provide a comprehensive mechanism to identify the starting location of polymorphic exploit code in network traffic. There will be too much overhead if all potential starting locations are tried, but simple heuristics for narrowing down the possible starting locations may miss some attacks. The polymorphic exploit code which does not use a NO-OP sled for attacking Serv-U FTPD MDTM Overflow vulnerabilities may bypass this emulation approach.

The work presented in this chapter combines both static analysis and instruction emulation techniques to detect the remote polymorphic exploits. The evaluation results demonstrate that the proposed method has a 100% detection rate on realistic exploits of many types, including those that use self-modifying code, and/or that do not have a NO-OP sled. No previous method of static analysis has been able to achieve this. Emulation methods (e.g., [34]) can deal with the polymorphic exploit codes with self-modifying decryption routines. However they are not robust against those without NO-OP sled.

3.4 Attack Analysis

We discuss now the possibility of defeating the the proposed detection method.

Fragmentation. Decryption routines are normally of small size. They can be contained within single packets. However, the attackers may deliberately split exploit traffic across multiple packets. It is trivial to reassemble packets before analysis, at the cost of modest additional processing overhead (i.e., the dependence on payload size is slightly greater than linear, as shown in Figure 3.4.)

No use of looping by the decryption routine. A loop is very likely to be needed for decryption purposes, since in-line coding of a decryption routine will otherwise be much longer (and therefore easier to identify). Interspersed in-line decryption code and the encrypted exploit payload should be highly carefully designed. This is because, after it has run, the decryption code should be bypassed by the decrypted exploit code. Here we do not claim there are no such encryption or decryption methods. Instead, we speculate preventing the use of looping for the decryption methods will raise the bar for the attackers.

Use of values not in the exploit code. Polychronakis et al. [34] have pointed out that attackers can use data from the environment in which the exploit executes. If self-modifying code relies on a key outside the address space of the exploit, this will not be detected by the proposed method at present. However, such exploits will be much more platform specific, and therefore much more sensitive to small system changes and randomization techniques [65].

Long or infinite loops. The analysis time of traversal and execution depends on the length of the derived chain of instructions. If the code contains a lengthy loop, or one which does not terminate, analysis may fail or may require an excessive amount of time. Nevertheless, our approach can still be useful as a first-stage detection method. Polychronakis et al. [34] demonstrated that long loops in normal network traffic are rare.

3.5 Summary

In this chapter, we presented a new method for detection of self-decrypting exploits. The proposed method scans network traffic for the presence of a decryption routine, which is characteristic of such exploits. The proposed method outperforms previous proposals [27, 28, 34, 29, 30] in its capability to identify more precisely the starting location of the decryption routine, with fewer assumptions. The method also can identify the decryption routine even if self-modifying code has been used to conceal its presence.

The evaluation results demonstrate that the proposed method has a 100% detection rate on realistic exploits of many types, including those that use self-modifying code, and/or that do not have a NO-OP sled. On a large collection of network traffic and Windows executables, a very low false positive rate was observed. The non-optimized implementation running time is roughly linear in the amount of data processed. These results indicate the proposed method is likely to be useful as part of an automated network defense again both targeted attacks, and large-scale zero-day worm outbreaks.

The proposed method is not claimed to be a panacea. There are ways that the proposed method can be bypassed such as using lengthy loops or using running-time-environment related values. To do so, attackers need to carefully craft their exploit code. Nevertheless, the proposed approach can still be useful as a first-stage detection method.

Future work will focus on generalizing the method for less obvious sequences of byte decoding. In addition, we will test the method on non-exploit code that uses code obfuscation, code encryption, and self-modification for legitimate purposes (e.g., to prevent reverse-engineering, and to protect license verification). We expect the way these techniques are used to be substantially different than exploit code.

In the next, we are going to introduce our second work which identifies metamorphic malware that uses code obfuscation for concealment.

Chapter 4

Metamorphic Malware Detection

In this chapter, we address the problem of identification of metamorphic malware and malware mutants. Unlike polymorphic malware, metamorphic malware uses code obfuscation techniques instead of encryption to bypass the conventional signature based detection.

4.1 Overview

Metamorphic malware and enormous malware mutants can easily bypass signature based malware detection [35]. Metamorphic malware can obfuscate its entire code in a variety of ways, such as control flow transposition, substitution of equivalent instructions, variable renaming, etc. [56, 26]. This creates an arms race between the metamorphic malware writers (or obfuscation engines such as Mistfall, Win32/Simile, and RPME as pointed out in [26]), the signature writers, and the owners/administrators of the threatened computers or devices, which must be protected. It is also quite common that new malware *mutants* (or variants) rapidly evolve from old malware, to which new functions have been added or existing functionalities have been tweaked [19]. For example, the VX Heavens website [20] provides access to thousands of malware variants in a variety of different categories. For each malware variant, a signature may be identified, packaged, and downloaded to the base of users expecting protection from the new attack. The huge range of possible variants and the speed with which they appear makes this a less and less practical approach.

One essential reason for this vulnerability is that signature based approaches are essentially syntactic and lack insight into the programs' semantics.

To address this problem, this chapter presents a semantic characterization of programs and a method of matching such characterizations as a basis for malware detection that is resilient to many commonly-used obfuscation techniques. Generally, the problem of determining whether a program will exhibit a certain behavior is undecidable. Therefore, this chapter presents an approximation approach that is based on static program analysis to address the problem.

Static program analysis is used for many purposes, such as security vulnerability checking [90, 102], and program behavior modeling for intrusion detection [103, 60]. Static program analysis needs to be done only once and does not require run-time monitoring of program execution, which has substantial overhead. Proving that two programs (for instance, an instance of a virus and a suspected metamorphic variant) are functionally equivalent is an undecidable problem, unfortunately. The goal for static analysis of this chapter is thus less than a full proof of functional equivalence.

Instead, we propose to characterize a program in a way that can combine both structure and function. This characterization is referred to as the *pattern* of a code fragment. Ideally, the pattern should be markedly different for distinct malware, and the obfuscation used by metamorphic engines would not drastically change the pattern derived by static analysis. The challenge is to compute such patterns quickly, and to find a way to compare patterns that yields insight into the similarity between program functions. These patterns then can be used in a way that is similar to the way that signatures are used by conventional virus checkers. The use of patterns must be substantially more resistant to obfuscation than the use of fixed signatures, however.

When two programs are analyzed to produce patterns representing their function, these patterns can be compared to determine how similar the programs are. The process of comparing patterns is termed *pattern matching* in this dissertation. The output of pattern matching is a *similarity score* between 0 and 1, where a value of 0 is interpreted to mean the program functions are very different, and a value of 1 is interpreted to mean the program functions are extremely close or identical. To make a decision whether an unknown program is similar enough to a known malware to require that it be quarantined, this score must be greater than a user-defined threshold.

We propose that patterns are based primarily on the system calls or library executed by the malware. We propose to statically analyze the control and data flow of *call traces*, which are the instructions that prepare the parameters used by a system call plus the corresponding call instruction. System call based modeling has been frequently used to characterize a program's behavior for intrusion detection purposes [103, 60]. It is a reasonable assumption that a compromised application cannot cause much harm unless it interacts with the underlying operating system [60].

As an illustration of this behavior, the Sapphire worm executes the following set of system calls [104]: `LoadLibrary`, `GetProcAddress`, `GetTickCount`, `socket`, `sendto`. Malware which did not make use of such system functions would likely be harder to write, and result in a much larger code size. The use of existing code obfuscation techniques or metamorphic program transformations does not in general remove such system calls from the malware.

The proposed approach differs from a previous research contribution [37] with similar assumptions and goals. That work proposes to use semantic templates of certain malicious behavior (such as the decryption loop in polymorphic malware) to detect malware. The templates are generated by studying the common behavior of a set of collected malware instances and are generated manually. The method in this chapter, in contrast, automatically generates a pattern that characterize a program's semantics and uses this pattern to detect either obfuscated, or mutated, malware.

The proposed method has been implemented and evaluated on actual malware variants, widely-used benchmark programs that have been randomized, and different releases of the GNU binutils programs [40]. The evaluation results demonstrate the proposed method can make a clear distinction between semantically equivalent or related programs, and those that are not. The measured similarity score of an original benchmark program and its randomized version in most cases achieves a value of .95 or greater. The measured similarity scores of different releases of the GNU binutil programs can achieve .75 or better. The measured similarity scores for different malware variants vary, but there is a very clear distinction between malware variants and non-variants. To apply the proposed approach in practice, a reasonable threshold can be set by the user to determine the sensitivity of malware detection.

4.2 The Proposed Method

In this section we present a new method of static analysis of executables. This method disassembles two executables and then computes the degree of similarity between them. The essential characteristics used for this comparison are the system or library function calls made by the two programs. The method is intended to be used for recognition (and subsequent isolation) of metamorphic malware and malware variants.

4.2.1 Pattern Generation Based on Static Analysis

System calls and library function calls are proposed to be used as the basis for generating a pattern that characterizes a target malware. Control flow and data flow analysis are used for this purpose.

To generate a pattern for code fragment p, p is disassembled first. There are a number of methods and tools designed to disassemble obfuscated binary code. The method of Kruegel et al. [41] was adapted for this purpose.

Once the code is disassembled, the control flow is easily obtained through static analysis. The result of such an analysis is a set of basic blocks and the transfers of control between those blocks. The call instructions that branch to system or library functions are then identified. Let I_i denote such a library call instruction in block i, and let the total number of library call instructions in the program be denoted as N.

The next task is to identify instructions that affect the parameters (values in memory or registers) used by the system functions when they are called by the program. While there can be many such parameters, the only such parameter used at present by the proposed method is the *target address* of the `call` instruction. Finding the instructions that affect the target address can be accomplished by data flow analysis.

The data flow analysis is initially given a single block i, and includes the system call or library function call I_i contained in that block. In block i, the instructions affecting the parameters of I_i are determined and sliced. Essentially, they refer to the instructions with definitions reaching [1] this call. For each of these instructions, the blocks affecting their input operands are determined by data flow analysis. This process of backwards data flow analysis continues until either (a) the target block i is again reached (in which case a cycle has been discovered), or (b) there are no more instructions remaining for which backwards data flow analysis must be performed. The dependency or data flow relationship between instructions can then be represented as a graph, with a vertex for each instruction in the program, a directed edge from u to v if the instruction corresponding to vertex u affects the operand(s) of the instruction corresponding to vertex v, and the vertex representing I_i

[1] Please refer to a textbook on compiler theory [96] for the explanation of reaching definition in data flow analysis.

as the sink of the graph. A *maximal instruction trace* B in this graph is a path from a vertex having no predecessor in the graph to the vertex representing I_i. The above process is performed for each system or library function call encountered in the disassembled code. To implement the backwards data flow analysis for finding maximal instruction traces, we use a *depth first search* algorithm to traversal the control flow graph. There is a limit on the depth of traversal (i.e., the number of visited control flow blocks in one pass). The limit is configured as 8 in our experimentation to reach an acceptable runtime performance. For instance, the results can be quickly obtained within minutes.

Following this data flow analysis, the instructions of each maximal instruction trace are processed to generate a subpattern for the trace. For this purpose, each instruction is first converted into an intermediate representation, based on the semantics of the instruction. This intermediate representation is convenient for processing and allows functionally equivalent instructions to be represented in the same way. It also allows the method to be applied regardless of the instruction set architecture, although at present only the Intel x86 architecture [2] is targeted, due to its popularity.

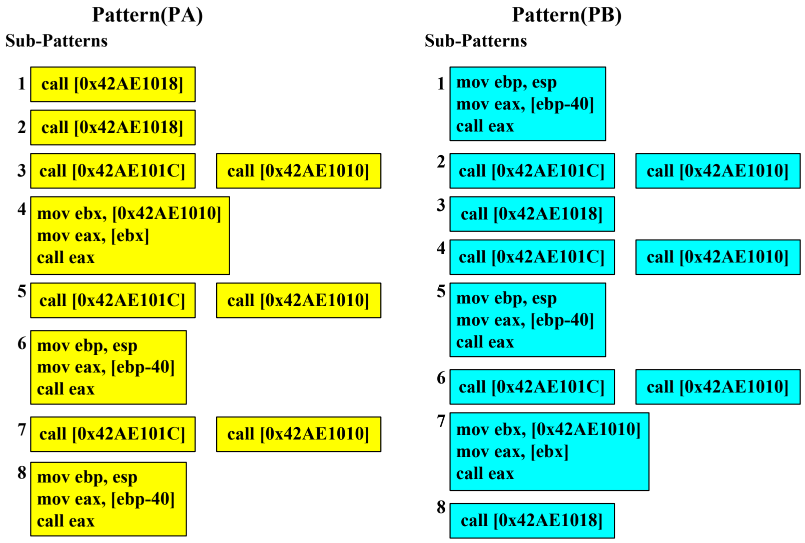
The intermediate representation consists of the *operation type*, the *operands*, and the *operand addressing modes* (i.e., immediate data, register, or memory addressing). The operation types for the x86 architecture are classified into eight major categories (e.g., data transfer, arithmetic, logical, control transfer) and within each category multiple subcategories may be defined. For instance, the `loop` and `jcc` instructions both transfer control and therefore are assigned to the same operation type. Operands are classified as being of type source (read only) or destination (write only or read/write), and the addressing mode and associated register, if any, are recorded. The conversion to intermediate form allows many instructions that are functionally equivalent to be identified to a limited degree. For instance, using intermediate representations, the instructions `sub ecx, ecx` and `xor ecx, ecx` are identified as functionally equivalent to `mov ecx, 0`, and the instruction `push eax` is identified as being equivalent to `dec esp, 4` followed by `mov [esp], eax`. Section 4.2.3 provides details on the intermediate instruction representation.

After conversion to the intermediate representation, the instructions in each maximal trace are symbolically executed in a very limited way. Currently, this symbolic execution is simply the propagation of constant values. Suppose the first (earliest) instruction in a trace assigns a constant value `c` to a register or memory location, and this constant value can be propagated to the target system call I_i as the address to which flow of control will occur. This `call` instruction and the (constant) target address then form an *element* of a subpattern for a single maximal trace ending at I_i. Note that this symbolic execution is not sophisticated enough to recognize all target addresses unambiguously. For instance, some addresses may be computed from information that is not available until runtime. Therefore, when an instruction cannot be symbolically executed, the propagated constants are bound to it and recorded. All such instructions in their intermediate representation are recorded in order and form an element of the subpattern for a single maximal trace ending at I_i. An executable instruction in a maximal trace is not included in the element.

The *subpattern* for I_i, denoted U_i, is the set of all such elements for all maximal traces ending at I_i. The set of all such subpatterns U_i for all the system or function calls in code fragment p is called the pattern of p, and is denoted as P^p. The intuition behind this definition of a program pattern is that a malware program will normally make use of some well-defined system services, whose addresses must be found (so that they can be accessed) in a well defined way. Attempts to obfuscate the program function, without changing the set of system or library function calls, can still leave this behavior visible to inspection. Even obfuscation of the target of a system call may leave the true target exposed as one possibility. Since the proposed method uses all possible traces, this true target will remain part of the pattern of the code fragment. The use of both symbolic execution and control flow analysis for disassembly will also overcome many known methods of obfuscation, as will be shown in section 5.3.

Figure 4.1 shows an example of the patterns generated for code fragments of the Sapphire worm and for a metamorphic variant of this worm. For purposes of illustration, each sub-pattern is presented in Intel x86 assembly language form, and is a result of data flow analysis and symbolic execution. For instance, subpattern 1 of pattern PA results from symbolic execution of the instruction trace `mov esi, [0x42AE1018]` || `call [esi]`, in which the second instruction operand depends on the first instruction. Subpattern 3 of pattern PA has two elements which result from two traces whose target is the same library function call. In this example, there happens to be multiple subpatterns which are identical. This is because some of the library functions are called in multiple places, with different parameters.

The next section explains a method of pattern matching to compute the similarity between two binaries. The input to this process is the patterns derived from the binaries in the way just described. The pattern matching algorithm is intended to overcome the



```
1 call [0x42AE1018]
2 call [0x42AE1018]
3 call [0x42AE101C]    call [0x42AE1010]
4 mov ebx, [0x42AE1010]
  mov eax, [ebx]
  call eax
5 call [0x42AE101C]    call [0x42AE1010]
6 mov ebp, esp
  mov eax, [ebp-40]
  call eax
7 call [0x42AE101C]    call [0x42AE1010]
8 mov ebp, esp
  mov eax, [ebp-40]
  call eax
```

```
1 mov ebp, esp
  mov eax, [ebp-40]
  call eax
2 call [0x42AE101C]    call [0x42AE1010]
3 call [0x42AE1018]
4 call [0x42AE101C]    call [0x42AE1010]
5 mov ebp, esp
  mov eax, [ebp-40]
  call eax
6 call [0x42AE101C]    call [0x42AE1010]
7 mov ebx, [0x42AE1010]
  mov eax, [ebx]
  call eax
8 call [0x42AE1018]
```

Figure 4.1: The patterns of code fragments of Sapphire worm and its metamorphic version.

differences between two variants of the same malware.

4.2.2 Pattern Matching

The purpose of pattern matching is to determine if two code fragments are similar enough to exhibit functional equivalency. The proposed method does not produce a formal proof of equivalence. Not only is that undecidable, but malware variants may in fact compute somewhat different results. Rather, we consider similarity in system or function call behavior to be strong evidence that programs have a similar purpose. The two requirements for defining patterns and the resulting pattern matching algorithm are:

1. The pattern derived from one malware program should be very different from patterns derived from other programs, whether benign or malware of another type.

2. Patterns derived from metamorphic variants of a single malware program should be very similar.

The matching algorithm is defined as follows. Two code fragments k and l are given, where k may be, for instance, a known instance of malware. The pattern for k has been computed and is represented as $P^k = \{U_1^k\ , ..., U_{N_k}^k\}$. The pattern for l has been computed and is represented as $P^l = \{U_1^l, ..., U_{N_l}^l\}$.

Let similarity scores be real values between 0 (minimum similarity) and 1 (maximum similarity). Suppose similarity scores between all pairs of subpatterns, where one subpattern is taken from P^k and one subpattern is taken from P^l, have been computed.

A *pattern matching* of k and l is a one-to-one assignment from the set of subpatterns of k to the set of subpatterns of l. A maximum matching is one that includes all of the subpatterns of k, and/or all of the subpatterns of l. A maximum weighted matching is one that maximizes the sum of the similarity scores of the pairs of subpatterns that are matched. The value or score produced by a maximum weighted matching W is equal to the mean of the similarity scores of pairs of subpatterns that are present in that matching:

$$M(P^k, P^l) = \frac{\sum\limits_{\langle U_i^k, U_j^l \rangle \in W} score(U_i^k, U_j^l)}{max(N_k, N_l)} \tag{4.1}$$

A maximum weighted matching is an optimistic approach to computing the similarity between two code fragments. The process of deriving and matching patterns should not be greatly affected by small errors in disassembly and data flow analysis, or by current program obfuscation techniques. These claims are evaluated in section 5.3.

Pattern matching is performed after similarity scores are computed for all pairs of sub-patterns. For each such pair of sub-patterns, the similarity score of all pairs of elements is computed, where one element is taken from the first sub-pattern and the other element is taken from the second sub-pattern. From this, a maximum weighted matching of the elements of the two sub-patterns is computed in the same way as mentioned before. The similarity score of this pair of sub-patterns is then the mean of the similarity scores of pairs of elements that are matched.

Finally, computing the similarity of two elements involves comparison of the instructions or instruction sequences (still in their intermediate form) in the two elements. This step finds a maximum weighted matching between the instructions in the two elements. To do this requires computing the similarity between any two instructions, using as input their intermediate forms. The computation is only an estimate of the similarity between instructions. Therefore, a heuristic method is used. This method first computes the similarity between operation types. As an example, `add` and `subtract` operations are deemed to be similar, while `add` and `call` are not. The comparison of operands checks for each operand pair whether the addressing mode and the operand values (when they can be determined) are the same, and scores them based on closeness. Closeness of operands is weighted more heavily than closeness of operation types when computing a final similarity

	PB							
	0.1	0.2	🟩 1	0.2	0.1	0.2	0.1	1
	0.1	0.2	1	0.2	0.1	0.2	0.1	🟩 1
	0.05	🟩 1	0.2	1	0.05	1	0.05	0.2
PA	0.73	0.05	0.1	0.05	0.73	0.05	🟩 1	0.1
	0.05	1	0.2	🟩 1	0.05	1	0.05	0.2
	🟩 1	0.05	0.1	0.05	1	0.05	0.73	0.1
	0.05	1	0.2	1	0.05	🟩 1	0.05	0.2
	1	0.05	0.1	0.05	🟩 1	0.05	0.73	0.1

Figure 4.2: The maximum weighted pattern matching of code fragments of the Sapphire Worm and a metamorphic variant, whose patterns are shown in Figure 4.1. Each Cell is a similarity score of two subpatterns, one from pattern PA, and one from pattern PB. The marked cells show the maximum weighted matching. The score of pattern matching $M(PA, PB)$=8/8=1.

score for two instructions. This computation is designed to be accurate enough to capture most obfuscations used in practice. Implementation details on the instruction comparisons are provided in next subsection.

Figure 4.2 shows an example of the maximum weighted matching process for the two patterns shown in Figure 4.1.

A software prototype of the proposed method has been implemented, based on the ideas described above. The Hungarian algorithm [55] is a well known method for solving weighted matching problems and was used in the implementation. The complexity of this algorithm is approximately $O(\max^2(N_k,N_l))$. Although the Hungarian algorithm has a polynomial running time, this could still be undesirably slow. For instance, a large program whose code size is measured in MB can easily produce thousands of subpatterns. Therefore, when the number of subpatterns exceeds a threshold, an approximate version of maximum weighted maching is used.

4.2.3 Implementation Details

This subsection presents a few details that are critical for repeating our implementation.

Disassembly

The implementation is targeted at the Intel [2] $x86$ instruction set which contains variable length instructions. A brief introduction is given at Appendix A.2. The disassembly engine is built on and modified from 1) the $x86$ instruction set parsing capability of an

```
typedef struct {
        BYTE state;
        BYTE length;

        BYTE type;
        BYTE subtype;

        BYTE desMode;
        BYTE srcMode;

        BYTE rBit;
        BYTE immBit;
        BYTE dSeg;
        BYTE sSeg;
        BYTE dBase;
        BYTE dIndex;
        BYTE sBase;
        BYTE sIndex;
        BYTE scaler;
        UN imm;
        int offset;

        DWORD next1;
        DWORD next2;
        DWORD status;
        DWORD preInstr;
        UN refValue;
        DWORD addr;
} _instruction;
```

(a)

```
typedef struct {
        BYTE type;
        BYTE subtype;

        BYTE desMode;
        BYTE srcMode;

        BYTE rBit;
        BYTE immBit;
        BYTE dBase;
        BYTE dIndex;
        BYTE sBase;
        BYTE sIndex;
        BYTE scaler;
        UN imm;
        int offset;

        DWORD addr;
        UN dBvalue;
        UN dIvalue;
        UN sBvalue;
        UN sIvalue;
        UN refvalue;
        char *fname;
} instructionStr;
```

(b)

Figure 4.3: The data structures of intermediate instruction representation.

existing disassembler [105]; 2) the Windows PE program header processing capability of `pedump.c` supplied in MSDN [106]; 3) the Linux ELF program header processing capability of `rand_elf3.c` supplied in a binary randomization tool [107]; and 4) the disassembly algorithm of obfuscated executables according to the description of [41].

Intermediate Instruction Representation

Each binary instruction is converted to an intermediate representation for the convenience of analysis. This preserves its information generated during the analysis and the instruction semantics. There are two forms of intermediate representation structures (as shown in Figure 4.3) which are used in different phases. The structure shown in Figure 4.3(a) is used in the disassembly phase and the control flow analysis phase, where raw instruction information is recorded. The structure shown in Figure 4.3 (b) is used in the data flow analysis phase, where the processed information (such as interim results of propagated data) is recorded.

There are four categories of fields in the intermediate representation structures. The first category is the instruction's *operation type* as indicated by the field `type` and subtype. Using the operation type instead of the opcode[2] as a way to record the actual operation has two functions. It allows many instructions that are functionally equivalent to be identified to a limited degree as previously discussed, and it assigns them to the same type and subtype. It also reduces the coding complexity and improves the runtime performance. Otherwise, there would be many functions that have to be coded in such a way that resembles the instruction parsing. These functions would be designed for control flow analysis, data flow analysis, and instruction comparison upon two patterns matching. In our implementation, the instructions are classified into eight major categories and within each category multiple sub-categories are defined. The eight major categories are introduced in Appendix A.2.

The second category is the instruction's *operand addressing modes* as indicated by the field desMode and srcMode. The third category is the instruction's *operands* that include fields rBit, immBit, dSeg, sSeg, dBase, dIndex, sBase, sIndex, scaler, imm and offset. Operands are classified as being of type source (read only) or destination (write only or read/write). The addressing modes indicate how the operands are addressed. The IA-32 [2] offers 24 memory addressing modes with flexible combinations of registers (i.e., with the base, with or without index register, and with or without a scaler to the index register), displacement or immediate number. The field rBit and field immBit indicate the bytes of the corresponding register(s) and the immediate number (1, 2 or 4 bytes). The information reflected by those two categories is obtained during the instruction parsing phase which is the first phase of disassembly. Sometimes, the information is *explicit* through the instruction encoding. Sometimes the information is *implicit* and will require an examination of the instruction's semantics as specified by the manual [2]. For instance, instruction loop, IMUL, MOVS, RCL, and many others belong to the latter case.

The fourth category is the *accessorial information fields*, that include the remaining fields that are generated during the disassembly, control flow analysis, and data flow analysis phases. The fields dBvalue, dIvalue, sBvalue, and sIvalue record the values of the corresponding registers as indicated by dBase, dIndex, sBase, and sIndex. If an instruction has an indirect addressing of operands, the fields addr and refValue will record the address and the referenced value if they are available.

There are five data types (BYTE, DWORD, UN, int, and char *) in the two data

[2] opcode is a term used in [2] to denote an instruction's operation. The opcode table can be looked up in [2].

structures shown in Figure 4.3. In a 32-bit machine, the latter four types have 4-byte lengths. Type `BYTE` has a 1-byte length. Type `UN` is a union type that is a union of `char`, `short`, and `int`. In an immediate addressing mode, an immediate number could have 1, 2, or 4 bytes. Type `UN` is designed to deal with such issues.

The order arrangement of the fields in the data structures of intermediate instruction representation considers optimization of data structure allocation in memory to minimize the gap occurred in alignment.

Heuristic Instruction Comparison

Instruction comparison is the basis of pattern matching. To compute the similarity between any two instructions, their intermediate representations are used as input. The computation is only an estimate of the similarity between instructions. Therefore, a heuristic method is used.

Figure 4.4 shows the high level procedure to process the comparison. The instruction comparison involves four components, including *operand type*, *first source operand*, *second source operand*, and *destination operand* comparisons. In Figure 4.4, these are indicated as `cmpType`, `cmp1stSrcOperand`, `cmp2ndSrcOperand`, and `cmpDesOperand` procedures. These comparison can deal with many obfuscations of a single instruction. This heuristic instruction comparison is not intended or capable of dealing with multiple instructions obfuscation, such as using two or more instructions that are semantically equivalent to substitute an instruction, However, the data flow analysis, which does simple symbolic execution, can handle or mitigate this problem of multiple instructions obfuscation to a limited degree.

The similarity between operation types is computed first. This process resembles a simple look up operation. The closeness of the operation types that are represented by the fields `type` and `subtype` are empirically pre-determined.

The comparison of operands checks for each operand pair whether the addressing mode, and their operand values (when they can be determined) are the same and scores them based on closeness. For most frequently used instructions, there are at most three operands. (An exceptional example is instruction `pushad` which pushes all general-purpose registers into stack. Therefore it has more than 3 operands. In this case, special treatment can be adopted.)

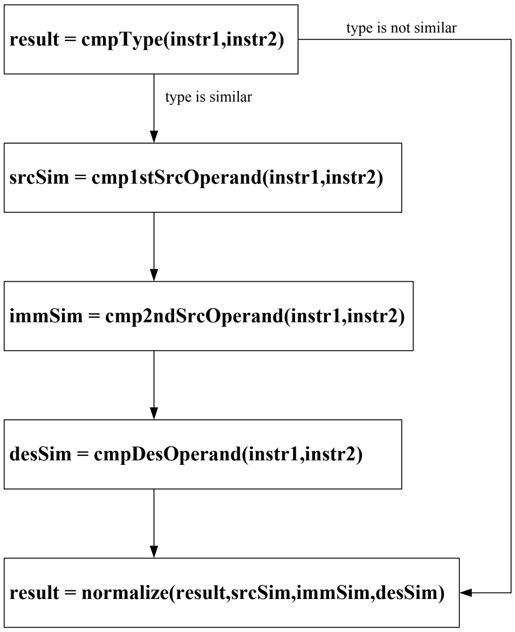

result = cmpType(instr1,instr2)
type is not similar
type is similar
srcSim = cmp1stSrcOperand(instr1,instr2)
immSim = cmp2ndSrcOperand(instr1,instr2)
desSim = cmpDesOperand(instr1,instr2)
result = normalize(result,srcSim,immSim,desSim)

Figure 4.4: Instruction Comparison Flow Chart.

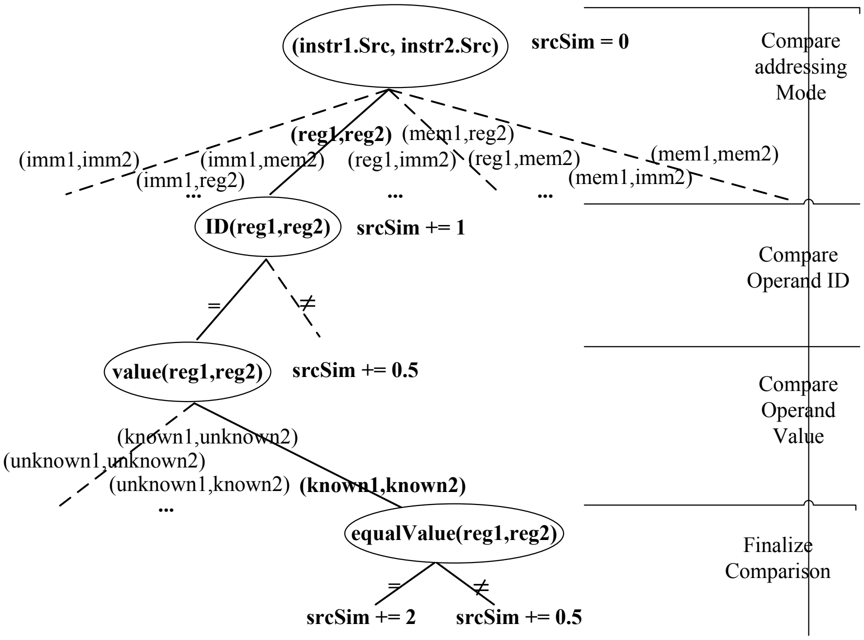


Figure 4.5: Partial Instruction Source Operand Comparison Decision Tree.

Figure 4.5 abstracts the comparison of first source operand pair as a decision tree. After each decision is made, a partial score is given. The partial score is an empirical value which is set to reflect the importance of the corresponding equation. For instance, in two malware programs, one of which is a metamorphic version of the other, there are two instructions `mov [eax + 01h], edx` and `mov [ebx + 01h], edx`. The two instructions could be a case of obfuscation using the *register reassignment* technique. (For more details, please refer to Appendix A.3.) In this example, the source operands are both register addressing and they are `edx`. According to the decision tree in Figure 4.5, if the values of `edx` can be determined as a result of data flow analysis, and they are equal, then the similarity score for this source operand comparison is 3.5. If the values of `edx` can be determined but they are different, then the similarity score is only 2 instead. A similar approach is applied in comparison of the second source operand pair and the destination operand pair. The final instruction comparison score will be normalized to 1.

Closeness of operands is weighted more heavily than closeness of operation types when computing a final similarity score for two instructions. This is due to the observation that it is less likely for two unrelated instructions to have the same operands than to have the same operation types. The ratio is empirically set at 3 over 7 for closeness of operand type to closeness of operands. The closeness of different operands are weighted slightly differently. The closeness of destination operands weights slightly heavier. If there are two operands, one being source and the other being destination, the ratio is 3 over 4. If there are three operands, two being source and one being destination, the ratio is 2:2:3.

In the next section, the preliminary results from testing of this software are described.

4.3 Evaluation

The proposed method computes the similarity between two binary executables based on the characteristics described above. If one executable is derived from another (i.e., is a variant or version of another), the computed similarity should be very high. Otherwise, the computed similarity should be low, with a large gap allowing these two cases to be easily distinguished.

The proposed method has been fully implemented. This implementation can analyze executables for both the Linux and Windows operating systems, compiled for the Intel x86 instruction set architecture.

Three sets of inputs were used to test this hypothesis experimentally. The first set of inputs consisted of benchmark programs (compiled for Linux) that were processed using a tool for fine-grained randomization of commodity software [107]. The second set of inputs consisted of variants of known Windows viruses, downloaded from the VX Heavens [20] website. The third set of inputs consisted of various releases of the GNU binutils programs, compiled for the Linux platform. For each set, the similarities of known variants or versions were computed, and when it made sense to do so, the similarities of unrelated programs (neither derived from the other) were computed as well. The experiments and the results are described in more detail below.

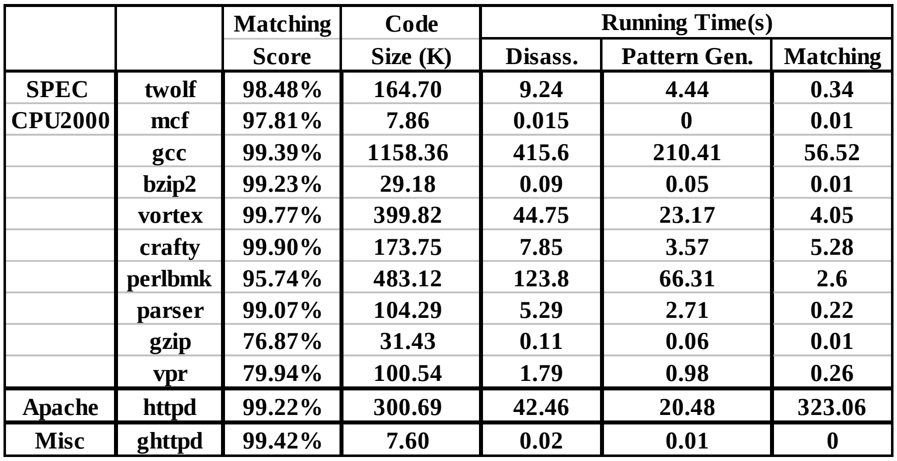
We were not able to use any existing tools designed to produce an obfuscated version of an arbitrary executable program that contained all common obfuscation techniques. Previous work [37] manually generated the obfuscated test cases with simple obfuscation techniques. The work [35] generated obfuscated test cases on programs written in visual basic language to test the resilience of commercial anti-virus software to metamorphic malware.

4.3.1 Randomized Executables

To test resilience to obfuscation, a set of programs was randomized using the ASLP tool [107]. This tool uses binary rewriting to rearrange the static code and data segments of an executable file in a random way. It performs fine-grained permutation of the procedure bodies in a code segment and of data structures in the data segment. This randomization technique can invalidate the use of static signatures for recognition of malicious code. This experiment was performed on a machine running Fedora Core 1, with a Pentium 4 CPU of 2.26GHz and 512M of RAM.

ASLP was applied to programs taken from two well-known benchmark suites: the SPEC CPU2000 programs [108] and two web server programs (the Apache web server httpd [109] and and the GazTek web server ghttpd [110]). The similarity between the randomized and original versions of each program was then computed using the proposed method. The results are shown in Figure 4.6.

Of these 12 test cases, 10 have similarity scores above 95%. A perfect matching tool should return similarity scores that are exactly 100% since the randomized programs



		Matching	Code	Running Time(s)		
		Score	Size (K)	Disass.	Pattern Gen.	Matching
SPEC CPU2000	twolf	98.48%	164.70	9.24	4.44	0.34
	mcf	97.81%	7.86	0.015	0	0.01
	gcc	99.39%	1158.36	415.6	210.41	56.52
	bzip2	99.23%	29.18	0.09	0.05	0.01
	vortex	99.77%	399.82	44.75	23.17	4.05
	crafty	99.90%	173.75	7.85	3.57	5.28
	perlbmk	95.74%	483.12	123.8	66.31	2.6
	parser	99.07%	104.29	5.29	2.71	0.22
	gzip	76.87%	31.43	0.11	0.06	0.01
	vpr	79.94%	100.54	1.79	0.98	0.26
Apache	httpd	99.22%	300.69	42.46	20.48	323.06
Misc	ghttpd	99.42%	7.60	0.02	0.01	0

Figure 4.6: Pattern matching between randomized and original programs. Programs are from SPEC CPU2000 benchmark, the Apache web server httpd, and the GazTek web server ghttpd. Code size refers only to the code segment size, not the size of the entire executable program.

and the original ones do have the same functionalities. However, our pattern matching is a heuristic approach that can only achieve approximations in program analysis. Nevertheless, the evaluation results shown here demonstrate that program changes due to randomization do not affect the ability of the proposed method to recognize the similarity in function between normal and randomized versions. The proposed method of analysis, using system calls as a point of reference, is robust to such changes in program structure.

4.3.2 Variant Detection Evaluation

Virus and malware writers have manually created many variations of common exploits, in an attempt to evade virus-checking tools. Hundreds of such examples can be found at popular hacking web sites. Such examples make use of a wide range of obfuscation techniques.

One such website is VX Heavens [20], which identifies programs that are variants of the same malware. More than 200 pairs of malware mutants were downloaded from this website. These program instances were selected from multiple malware categories. Among these instances, 36.6% were worms, 18.3% were viruses, 20.8% were backdoor programs, and 16.3% were trojan programs. The remainder of the programs included flooders and exploits. The tested malware programs had sizes ranging from 8K to 1M bytes. We observed some malware programs and their mutants employ multiple commonly used obfuscation

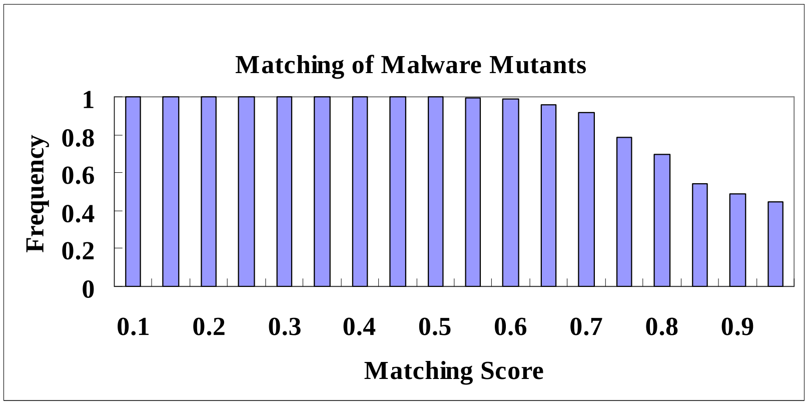


Figure 4.7: Malware Variants Pattern Matching. A y-axis value is an accumulative value.

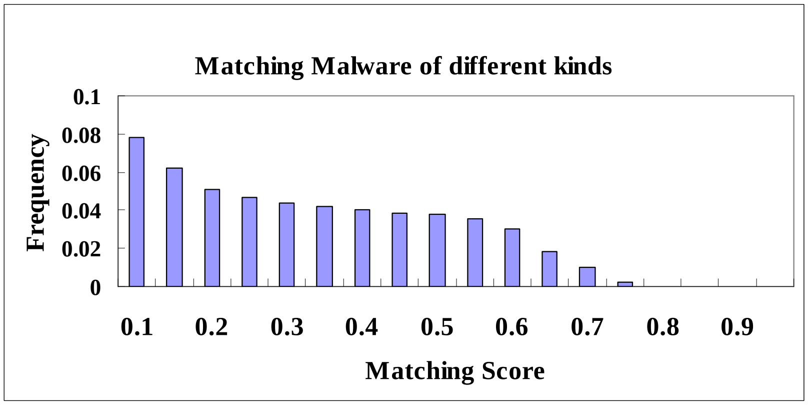


Figure 4.8: Malware Variants Pattern Matching. A y-axis value is an accumulative value.

techniques. For instance, the simplest obfuscation technique used is register renaming. A more complicated obfuscation technique is using functionally equivalent instruction substitution. The code addresses of the mutants can be very different. The relative offset in corresponding function `call` instructions are frequently adjusted. Sometimes, even within a single program, the same library functions may be imported multiple times at different addresses, although at runtime, these may be reloaded to the same address. Recognizing that two `call` instructions refer to the same target address requires data flow analysis techniques. A much more common case is that new functions or behaviors are added or revised in mutants. There is another obfuscation that is not considered here but is often encountered when downloading the malware programs. A non-trivial number of malware programs use encryption. The limitation of the proposed approach will be discussed in the next section.

In the first test, similarity scores were computed for pairs of executables that represented the same attack. A histogram of the resulting similarity scores is shown in Figure 4.7. The great majority of pairs are easily recognized as variants of the same function. For example, over 90% of the pairs have a similarity score of .7 or greater. These mutants represent the state of the art in mutation and obfuscation of malware, and thus are a worthwhile test case for any program attempting to recognize metamorphic variants in an automated way. The results indicate that comparison with a previous version of malware will, with high probability, identify a new version of the malware.

It is also important to measure the similarities computed between programs which are *not* variants of the same malware. In the second test, the malware programs were randomly paired with each other, excluding all instances that were identified on VX Heavens as being variants of the same malware. Similarities for the resulting pairs were then computed using the proposed method. Figure 4.8 shows the results. The computed similarities are very low, with less than 10% have a similarity score of 0.1 or greater (or to say, with over 90% having a similarity score of below .1). Approximately 1% have a similarity score of .7 or greater. It may be that malware even in different families are derived from a common code base, explaining these results. Further investigation is required.

For these programs, a similarity score of .7 would be an optimal threshold for concluding whether two programs, one of which is known to be malicious, are functionally equivalent. While this threshold does not perfectly distinguish malware variants from non-variants, keep in mind that this is a tough test case: identifying hand-crafted metamorphic malware, and distinguishing it from other malware, rather than distinguishing it from non-malicious code.

4.3.3 Version Difference Evaluation

The third experiment tested how well the proposed method recognized variations between different versions or releases of a program. Such versions are not intentionally obfuscated, but represent another case of software that is derived from a previous version of a program. They are therefore a useful test of the proposed method.

Releases 2.10 through 2.17 of the GNU project “binutils” binary tools [40] were used for this purpose. These tools are used for compiling, linking, and debugging programs. They make use of several common libraries for low level, shared functions. Different releases

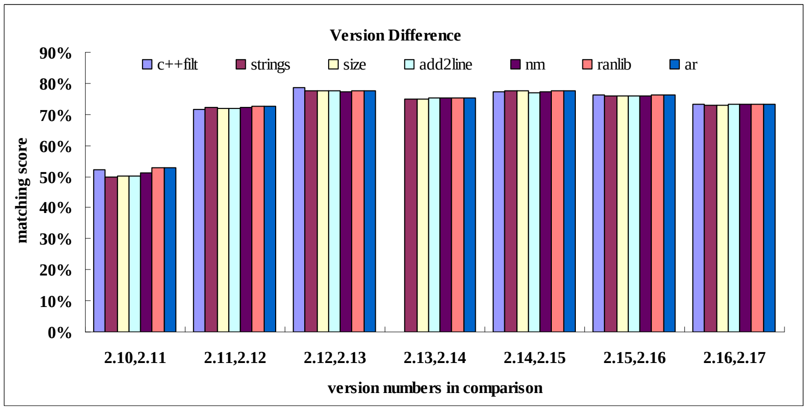


Figure 4.9: Pattern Matching of GNU Binutils programs. Each pattern matching is performed between two consecutive versions of a GNU Binutils program.

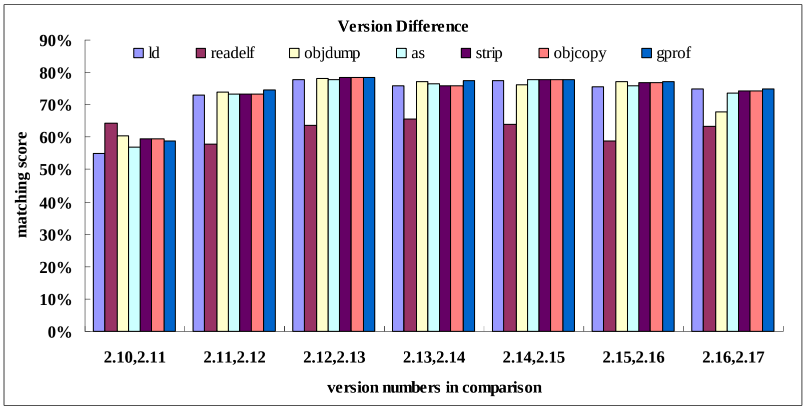


Figure 4.10: Pattern Matching of GNU Binutils programs. Each pattern matching is performed between two consecutive versions of a GNU Binutils program.

will represent varying degrees of modification to the original program code.

Figure 4.9 and Figure 4.10 show the result of computing the similarity between consecutive releases of each program, using the proposed method. For the great majority of cases, the computed similarity was greater than .7. The cases where this was *not* true are instructive to examine (see Figure 4.11 and Figure 4.12. Between release 2.10 and 2.11, code sizes of the utilities increased by approximately 50%, indicating a major revision, and the computed similarity scores were correspondingly lower (around .5). Figures 4.13 and 4.14 compare both release 2.10 and release 2.11 to release 2.17. It is clear that release 2.11 is much closer (in size and similarity) to 2.17 than release 2.10 is to 2.17. Also, between release 2.13 and 2.14, the size of `c++filt` grew 10-fold, indicating essentially a replacement by a new program; the computed similarity in this case was close to 0.

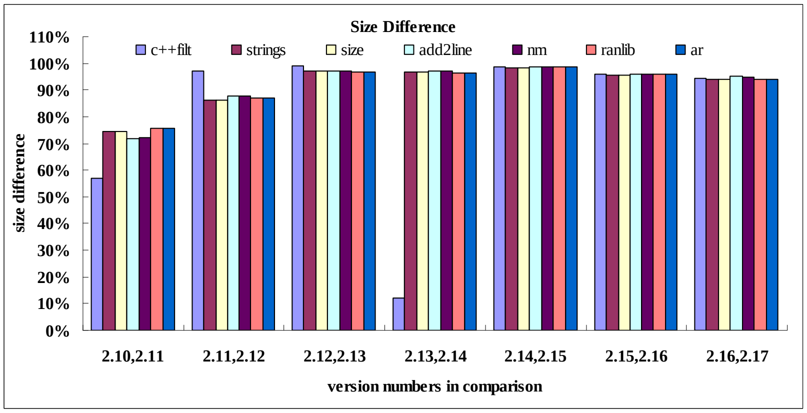


Figure 4.11: Size comparison of GNU Binutils programs. Each size comparison is performed between two consecutive versions of a GNU Binutils program.

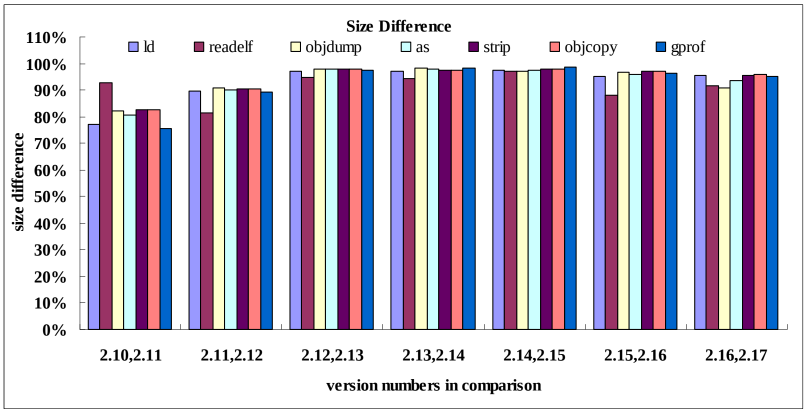


Figure 4.12: Size comparison of GNU Binutils programs. Each size comparison is performed between two consecutive versions of a GNU Binutils program.

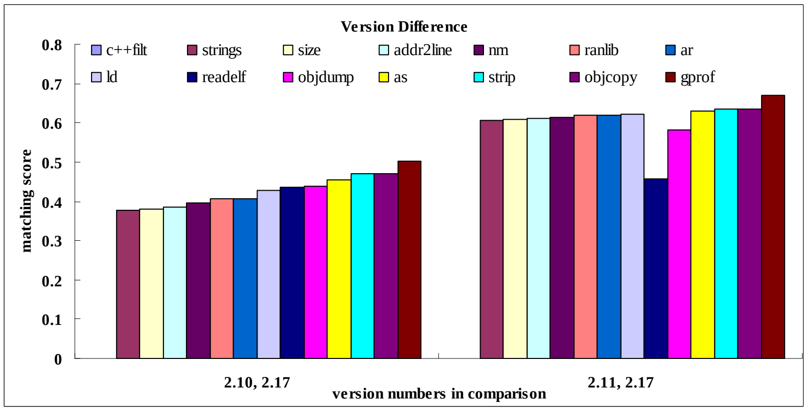


Figure 4.13: Pattern Matching of GNU Binutils programs. Each pattern matching is performed between version 2.10 and 2.17, or version 2.11 and 2.17 of a GNU Binutils program.

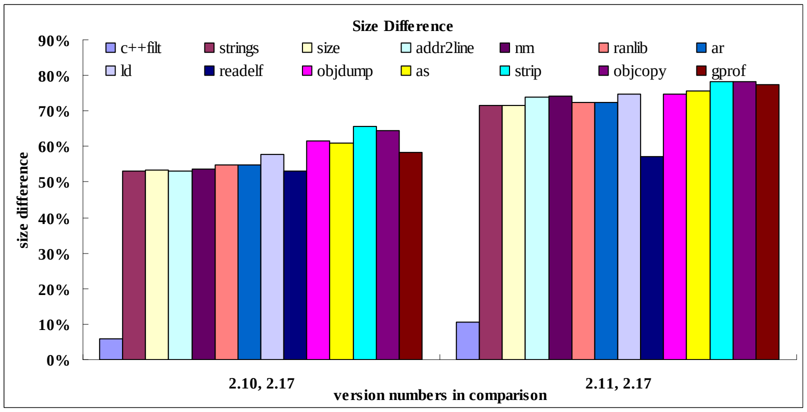


Figure 4.14: Size comparison of GNU Binutils programs. Each size comparison is performed between version 2.10 and 2.17, or version 2.11 and 2.17 of of a GNU Binutils program.

These results indicate that the proposed method is effective at computing the degree of similarity between programs in a way that is meaningful and that is not sensitive to modifications that preserve a program's function.

4.4 Discussion

4.4.1 Comparison with Previous Work

This section compares the current work with three previous methods [56, 37, 58] which statically analyze program semantics to detect metamorphic malware.

As pointed out in [37], the control flow graph comparison method of [56] can only handle very simple program obfuscations. For example, the detection algorithm only allows noop instructions to appear between matching instructions. By comparison, the method proposed in this chapter can handle a much wider range of obfuscations than this. It can also detect program mutants that have similar but not identical behaviors.

The method of [37] uses semantic templates to detect malware that has certain common behaviors. A template is manually generated by studying the common behavior of a set of collected malware instances; how to generate a general semantic template is not addressed. Our method, by contrast, proposes an automatic pattern generation method that characterizes a program's semantics. Furthermore, we argue that the proposed method is harder to bypass. The proposed method uses maximum weighted matching to be tolerant to inaccurate program disassembly and static analysis.

Chouchane and Lakhotia [58] use "engine signatures" to assist in detecting metamorphic malware. That work, however, can only deal with known instruction-substituting metamorphic engines. There are many ways to create metamorphic engines, by no means limited to instruction substitution. Moreover, their technique can be defeated by shrinking substitution methods. Our proposed method does not rely on specific engines. It characterizes and compares a program's semantics more generally. It uses control flow and data flow analysis and is more robust against complex metamorphism.

4.4.2 Limitations

There are two main limitations that can cause the failure of the proposed method. The major limitation is due to the use of static analysis. Since static analysis does not execute the program, run-time information is not available to derive a more accurate pattern. A case in point is static disassembly, which is not guaranteed to be 100% accurate [41]. Various techniques, such as indirect addressing, self-modifying code, and dynamic code loading can lower the accuracy of static disassembly and result in an inaccurate control flow analysis. Many techniques to improve disassembly accuracy have also been proposed [42], but are not currently implemented in our prototype. We also adopt various heuristics approaches in program analysis, which are sources that result in inaccurate pattern generation and matching. The set of heuristics approaches used in this work includes the instruction comparison, limited symbolic execution which only performs constant value propagation, setting a limit on the depth of control flow graph traversal to compute maximal instruction traces, and using only target addresses of system calls for pattern generation.

The second major limitation results from code evolution. Similarity of a mutant to an original code version largely depend on how the new instance evolves. If the majority of functionalities (i.e. the malware payload) of a mutant is replaced, only a small part will be matched, resulting in a low similarity score. In addition, the malware writer can purposely insert random "junk" functionalities *in terms of actual system calls made* to lower the matching score. Theoretically, if the number of "junk" functionalities goes unbounded or infinite, the proposed method will likely fail. To address this issue, it may be necessary for the program pattern to be focused on specific functions of the malware, rather than on the entire program's function. For instance, a whitelist of system calls or library calls can be built to filter out common functionalities that are usually unimportant such as `printf`.

4.5 Summary

This chapter presented a new approach to characterize and compare program semantics. A direct application of the proposed method is to recognize metamorphic malware programs, which conventional signature-based detection methods are less successful at detecting. The proposed method has been prototyped and evaluated using randomized benchmark programs, various types of real malware programs, and multiple releases of the GNU binutils programs. The evaluation results demonstrate three important capabilities of the proposed method: (`a`) it shows great promise in identifying metamorphic variants of common malware; (`b`) it distinguishes easily between programs that are not related; and, (`c`) it can identify and detect program variations, or code reuse. Such variations can be due to insertion of malware (such as viruses) into the executable of a host program, or programs revision. Thus an indirect application of the proposed work is to help localize an occurrence of one fragment of code inside another program using the maximum matching.

Future work will consider more accurate analysis of the parameters passed to library or system functions. We also believe the method's ability to identify similarities between binary executables will be useful for code attribution and other reverse engineering purposes.

In the next chapter, we are going to introduce the third work which automatically generates common malware behavior patterns for the detection of the metamorphic malware or new malware instances. It addresses the limitation of the proposed work of this chapter.

Chapter 5

Automated Generic Malware Pattern Generation

The chapter aims to propose an *automated* approach to discover common behavior patterns from a set of malware samples that can be used to detect metamorphic malware or new malware instances. This approach will improve the applicability of the semantic malware detector which is the second technique proposed in this dissertation. It could result in an about 80% reduction in semantic pattern population to detect known and new malware instances. Moreover, it will be more robust in the event of a junk behavior pollution attack than the malware detector is. This new approach combines static analysis and data-mining techniques.

5.1 Overview

It is a challenging and non-trivial problem to detect metamorphic malware and new malware instances, which pose great challenges to the current signature-based detection methods [35, 36, 37, 38]. Advanced detection techniques such as [57, 37, 69, 70], in contrast, focus on the specification and identification of malicious behaviors demonstrated by different families of malware. Examples of malicious behaviors include self-unpacking, hooking into web browsers and modifying critical data at load time. Since these approaches are not tailored to specific malware instances but specify the general behaviors of an entire family of malware, they are more robust against the mutated attacks that preserve these behaviors. However, the development of these behavior specifications is an onerous manual process which is considerably slower than the occurrence of new types of malware.

The goal of the present work is to propose an *automated* approach to discover a common behavior pattern from a set of malware samples that can be used to detect metamorphic malware or new malware instances. This approach will improve the applicability of a semantic malware detector [111]. First, given the overwhelming number of malware types and the malware mutants, the feature which uses common behavior patterns instead of a pattern per malware type increases the appeal and deployment of such metamorphic malware detectors by reducing the need for semantic malware pattern generation. Second, the feature which uses common behavior patterns instead of a pattern per malware type increases the malware detector's robustness to a junk behavior pollution attack as pointed out in [111].

The proposed method abstracts malicious program behaviors in the same way as the second proposed work of this dissertation [111]. It embraces static analysis techniques to analyze malware binary programs, and characterizes the program behaviors based on system calls invoked by the malware. Different from [111], this chapter focuses on the problem of how to find common behaviors among an entire family of malware instead of generating a pattern per single malware instance and its mutants. To find a common malware behavior pattern, a hierarchical clustering data-mining technique is used to mine the common malicious behaviors among the disassembled malware patterns that are outputted after the static analysis.

We made an observation that there could be common behaviors among an entire malware family that are unique enough for identification of the family. This observation starts with manually analyzing the source code of seven bot programs found through the Internet including the website in [39]. A brief description of bot software has been given in Chapter 1. The seven bot refers to `sdbot`, `agobot`, `spybot`, `jrbot`, `rxbot`, `ciscobot`, and `forbot`. Consistent with a previous study [10], these bot programs show similar structures and functionalities. Examples include a) providing the capability to disable antivirus software; b) registering the bot program in the system service registry to make it restartable; c) looping structure to detect the network connectivity in order to join a bot communication channel and wait for control commands; d) upgrade/uninstallation of the bot program; e) a sequence of `if-else` statements for processing attack commands received from external botmasters, etc [10]. Although different bot programs may vary in attacking abilities, such as installation of backdoor and keylogger programs which can be conveniently customized

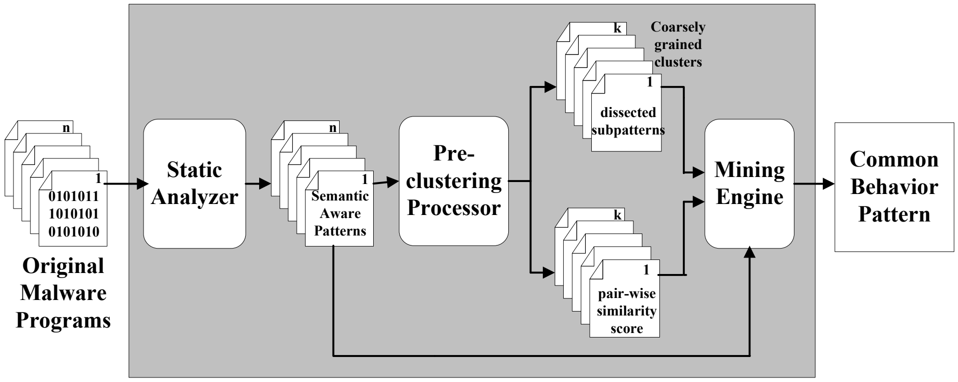


Figure 5.1: Overview of procedure for common behavior pattern generation.

by different malware writers, the bot family members share basic behaviors. Some of these behaviors are easily reflected through a set of system calls in their binary images and can be discovered. A reflection of these commonalities is that the evolution of malware development is derivative in nature. It is not a surprise that the source code of many malware is available and easily obtained from web sites such as [39].

This chapter uses static analysis and data-mining techniques to automatically find these shared program behaviors that can be unique enough for identifying the malicious software. Data mining methods are frequently used to detect patterns in a large set of data and have already been applied for mining benign behavior models for intrusion detection [59, 60, 61, 62] or malicious behaviors for malware detection [63]. The key to a data mining based behavior modeling is the extraction of behavior features. In this work, we resort to static analysis to extract program behavior features. Figure 5.1 illustrates the procedure of the proposed method. Input to the procedure is a set of malware binary programs from the same malware family such as the malicious bot software. Output from the procedure is the common behavior pattern of these malware samples. The outputted pattern can be directly used by a semantic-aware malware detector such as [111] for detection of this malware family. There are three notable functional components in the procedure. A ***static analyzer*** is the entry functional component which is responsible for processing the primary malware samples to derive their behavior patterns. A set of corresponding program behavior patterns are generated thereafter and dumped into separate files. They are the intermediate results for the other components to further process. The static analyzer is implemented according to the work in [111] which can effectively and statically analyze obfuscated malware program behaviors based on system calls. A ***pre-clustering processor*** is the intermediate functional component which coarsely clusters the program behavior patterns to prepare a classified data set for improving the performance and accuracy of the final functional component, a ***mining engine***. The pre-clustering processor outputs two types of file sets as the intermediate results with which the mining engine will work. One set is clusters of dissected sub-patterns. The other set is files of pair-wise similarity scores computed to measure the similarity between a pair of sub-patterns from a same cluster. The mining engine takes these results along with the primary program patterns and mines popular behavior patterns.

The proposed method has three notable features. First, it is fully automated. Given a set of malware programs, the proposed method discovers the common malicious behavior without any human intervention. Second, it works completely on binary code and requires no source code of the malware. Finally, it looks for similar behaviors in addition to the completely identical ones. Focusing only on identical behaviors makes the mining procedure easy, but potentially misses common malicious behaviors that involve intentional or non-intentional program obfuscation. This feature is reflected in the similarity score computation for each coarsely grained cluster.

Building program behavior models using various mining techniques for intrusion detection purpose is not a new topic [59, 60, 61, 62]. However, these methods construct benign behavior models and detect deviated behaviors via run-time monitoring. This chapter targets a different aspect. It uses data mining techniques for mining malicious behavior patterns. Moreover, the detection of malware using derived malicious behavior patterns is a static pattern matching process. A recent paper [46] developed a novel mining technique to specify malicious program behaviors by mining differences of execution traces between a malware sample and a set of benign programs. However, a general limitation to this dynamic execution based approach is the difficulty of simulating the malicious execution environment to obtain all possible malicious execution traces. The work by Schultz et al. [63] is closest to our work. It built a framework that uses three different data-mining algorithms to train multiple classifiers on a set of malicious and benign executables to detect new malware instances. The training binaries are statically analyzed to extract the properties being used, such as Dynamic Linked Libraries (*DLL*s), strings, or byte sequences of the binaries. Our approach uses a different feature extraction and a different data-mining approach. Moreover, the mined common behavior patterns are a byproduct that can be utilized in malware analysis. To summarize, compared with other mining approaches for analyzing program behaviors, the proposed new approach is entirely static analysis based, fault-tolerant, and easy to deploy. Therefore, it can be an effective complementary approach to malware defense.

The proposed method has been implemented and evaluated on real world malware programs and benign programs (i.e, Windows XP Professional System programs and several application programs). The malware samples are obtained from two trusted sources: Cyber-TA project [18] and Johns Hopkins University Botnet research project [112]. A set of experiments was performed to test the quality of the common behavior patterns which were generated with different parameter configurations. The evaluated results under an optimized parameter configuration in terms of detection rate and false positive rate are 94% and 8.3% respectively. The evaluated results under a second optimized parameter configuration have a *much lower* false positive rate which is 0.32% and a detection rate of 78%. In addition, this method results in an about 80% reduction in semantic pattern population to detect known and new malware instances when it is compared with the second technique proposed in this dissertation.

In the next section, the proposed method and the three functional components will be introduced in detail.

5.2 The Proposed Work

In this section, we present the various components of the proposed work (e.g., the static analyzer, pre-clustering processor, and mining engine) that automatically generate common behavior patterns for an entire family of malicious software. The generated patterns can be used by a semantic-aware metamorphic malware detector to detect such malware families and provide forward detection of new malware mutants.

5.2.1 Static Analyzer - extracting behaviors from binary code

To mine the common malicious behaviors the first step is to extract the program behavior features. The feature extraction decides the quality of a data mining based approach. In this work, we resort to static analysis to extract program behaviors. As pointed out by M. Christodorescu et al. [37], to analyze malicious behaviors we need to look into program semantics rather than just its syntactic features in order to better deal with obfuscated malware or numerous malware mutants. We adopt the second technique of this dissertation which is presented in Chapter 4, which is fully based on automated static analysis of executables, to summarize and compare program behaviors semantically and effectively. We adopt this approach for extracting behaviors from binary code in the present work. Next, the highlights of the adopted static analysis method [111] are briefly introduced to lay the necessary foundation.

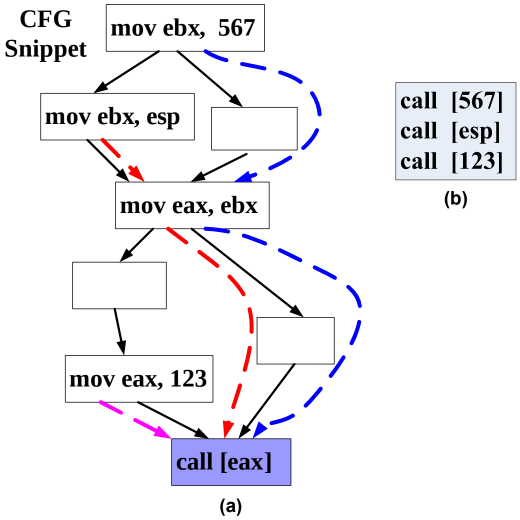


Figure 5.2: Pattern Generation Illustration. (a) A snippet of the control flow graph after disassembly of a program. Only the instructions that have data flow dependency to the `call` instruction are shown in each block. Three *maximal instruction traces* for the `call` instruction are shown as dashed lines. (b)The generated sub-pattern. It is the result of the simple symbolic execution of the maximal instruction traces and is stored in an intermediate representation format.

A program *pattern* is derived based on the system calls it makes. It characterizes the program's behaviors combining its structure and functionality. Essentially, the goal is to extract the system call instructions and the instructions that prepare the parameters used by them. This is done through control and data flow analysis of the disassembled program. Currently, the only type of parameter considered is the target address of a system call. To derive a pattern, a control flow graph (CFG) is derived first. Based on the CFG, all the system call instructions are found. Starting from a found system call, a set of *maximal instruction traces* are found by backward data flow analysis. A maximal instruction trace is the longest sequence of instructions that have data-flow dependency on a `call` instruction. Simply put, a maximal instruction trace forms a *def-use* chain to a system call instruction. A *sub-pattern* is generated for each found system call after performing a simple symbolic execution to aggregate the maximal instruction traces and propagate the possible values of `call` target addresses. Eventually, all the sub-patterns consist of a complete program *pattern*. Figure 5.2 shows a simple example to illustrate this pattern generation.

There are several outstanding advantages of such feature extraction. First, it improves resilience against dedicated program obfuscation. According to the experimental evaluation of [111], the adopted method can effectively recognize metamorphic malware (through the comparison of randomized and original benchmark programs) and malware mutants (through the comparison of real world malware examples). Second, it enhances the accuracy of evaluating the similarity of two behaviors. It is more than a simple count of the number of system calls a program issues, or a straight comparison of instruction sequences that can be carefully manipulated to demonstrate differences. By computing the maximal instruction traces for a system call, a generated pattern preserves some characteristics of the way that a program interacts with the underlying operating system. In this way, the adopted method is able to measure the similarities among malware mutants or different releases of benign software. Third, it implies a feasible way to dissect program behaviors for comparison. A sub-pattern is generated for a single system call and characterizes how the system call is issued to interact with the operating system. It is the smallest and relatively complete entity that is found to have such a capability. Dissection of program behaviors based on sub-patterns for comparison of program similarity is feasible and has demonstrated effectiveness through the evaluation done by [111].

The static analyzer is an important component of the proposed work and it is the key to the success of the data mining approach which is to be discussed next. However, the focus of the proposed work is the data mining approach that utilizes static analysis as its feature extraction procedure to find the distinctive and common characteristic behaviors among a set of malware programs.

5.2.2 Pre-clustering Processor - preparing classified data sets

Using a classified data set is expected to improve the performance of a data mining tool by reducing unnecessary computing and lead to more accurate results. This subsection discusses the approach of the pre-clustering processor, the intermediate component which is responsible for preprocessing the raw data sets to prepare classified data sets. The raw data sets consist of the semantic program patterns extracted by the static analyzer. The pre-clustering processor executes two actions. First, it performs classification on semantic program patterns in the raw data set. This results in *coarse-grained* clusters that have similar types of dissected sub-patterns and potentially have common behaviors. The finer classification is performed by the mining engine in the final stage. Second, the pre-clustering processor performs computations of entity similarities within each cluster. A mining tool needs numerical measurement of the relationship or distance between a pair of data items.

Classification

A classification is made based on the granularity of sub-pattern level. A sub-pattern is generated according to a single system call issued and is the smallest relatively complete entity which roughly characterizes a program behavior. A single instruction rarely possesses this capability. There are basically four types of sub-patterns as shown in Figure 5.3. They are all relevant to *absolute addressing* `call`s. *Relative addressing* `call`s are usually compilation results of a program's self-functions. A sub-pattern of type (a) involves a system call that can be clearly identified with a known DLL name and a function name. A sub-pattern of type (b) involves a system call whose target address is the sole identifiable information. Sub-patterns of type (c) and (d) correspond to the cases where system calls cannot be exactly identified due to various reasons (e.g., limitations of static analysis where the target address can only be determined at run-time, or some implementation deficiency). A classification based on type (c) or (d) does not mean that two behaviors in two different clusters represent distinct behaviors. Proving equivalence of program behaviors is an undecidable problem. Instead, according to the way they are issued, they are speculated to represent different behaviors.

A classification begins with dissecting program patterns into discrete sub-patterns and categorizing the sub-patterns based on their types as described in Figure 5.3. Three classification schemes were investigated and Scheme 2 shows advantages over the other two schemes. Each scheme reflects a different level of granularity in categorizing the sub-patterns. Figure 5.4 visualizes the three schemes.

Scheme 1 (Low): All sub-patterns are *indiscriminately* categorized into one cluster. This scheme actually involves no clustering. This scheme is slower than the other two schemes and is limited to smaller data sets.

Scheme 2 (Medium): Sub-patterns of type (a) are categorized into discrete

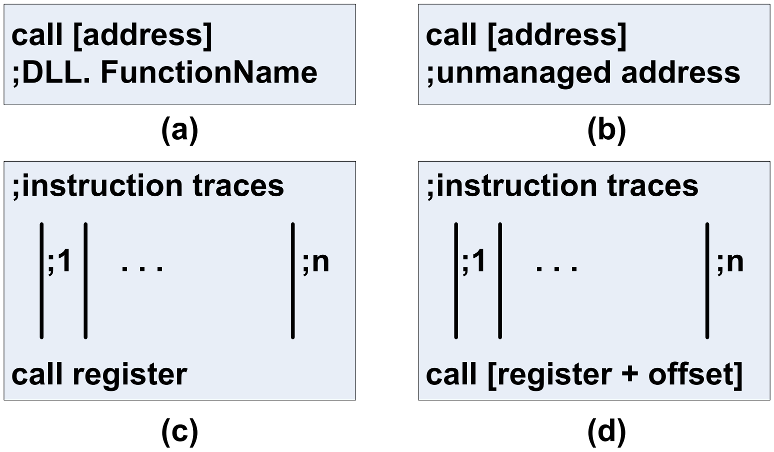


Figure 5.3: Four types of sub-patterns relevant to (a) direct (known) system calls; (b) direct (unknown) system calls; (c) indirect system calls with call address stored in register; (d) indirect system calls with call address stored in memory as specified. This way usually represents a method dispatch.

multiple clusters, each of which contain sub-patterns that relate to the system calls with same DLL name and function name. Sub-patterns of other types are categorized into single clusters. An assumption in the classification of behaviors is that a different system call (i.e., one that is imported from a different DLL or has a different function name) represents a different behavior. This assumption can be relaxed if the proposed method is fed with pre-determined classification information telling which groups of system calls can possibly represent similar behaviors. For instance, unicode and non-unicode versions of Windows library functions can be regarded as implementing the same behaviors. In our prototype implementation we simply hold this assumption.

Scheme 3 (High): This is a finer classification compared to Scheme 2. The difference is in the classification of sub-patterns of type (b). System calls casted at different addresses are considered "different." After the experimental evaluation, this scheme has shown to lead to slightly poorer performance of the proposed work. One explanation for this result is that the same system calls can be casted at different addresses.

Figure 5.17 and Figure 5.18 highlight the comparisons of the above pre-clustering schemes under the same test conditions. The comparison results show that overly specific (Scheme 3) or overly general (Scheme 1) classification will lead to an inaccurate pattern that causes a higher true false positive rate of flagging benign programs as being malicious or a lower detection rate of detection of malicious programs.

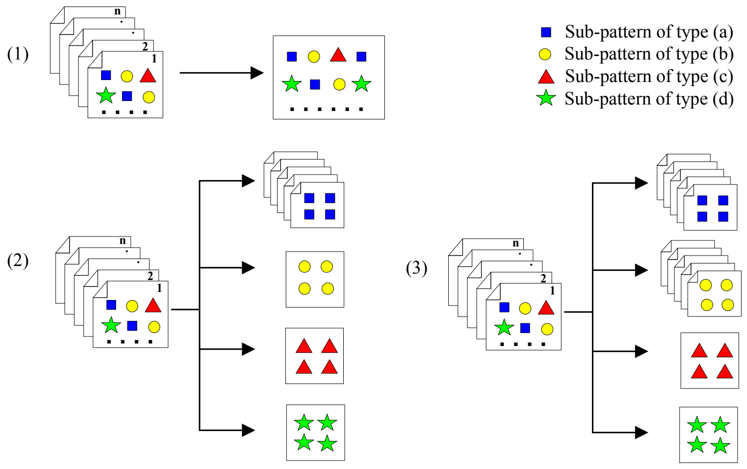

(1)
n
2
1
(2)
(3)
■ Sub-pattern of type (a)
○ Sub-pattern of type (b)
▲ Sub-pattern of type (c)
★ Sub-pattern of type (d)

Figure 5.4: Three schemes of pre-classification.

Similarity Measurement

Paralleling the classification process, similarities among sub-patterns that are from the same clusters are measured. One similarity score matrix is produced for one cluster. A non-diagonal entry in this matrix is a numerical value between 0 and 1 that estimates the similarity degree between two different sub-patterns in this cluster. 1 stands for the maximum similarity degree.

The function that computes the similarity of a pair of subpatterns is a method of semantic pattern matching proposed in the second work of this dissertation [111]. It uses a maximum weighted matching algorithm to find an optimistic match between the elements of the two sub-patterns. An element of a sub-pattern corresponds to a processed maximal instruction trace of the subpattern. A heuristic score function is used to calculate the similarity of two elements. This score function considers several factors including instruction operation type, operand addressing mode, and operand value and scores them based on closeness. For more details, please refer to Chapter 4.

5.2.3 Mining-Engine - mining common behavior patterns

The mining-engine is the final component of the proposed work. It mines common behavior patterns from malware samples. There are three inputs from previous phases, including the coarse-grained clusters of sub-patterns, the files of similarity score matrixes, and the original semantic malware patterns. The mining-engine executes two steps. First, it finds fine-grained clusters from the coarse-grained clusters. This step involves a data-mining algorithm. Second, it finds popular fine-grained clusters and chooses their representative sub-patterns to construct the common behavior patterns.

We use a well-known clustering method, *hierarchical clustering*, as the underlying data-mining algorithm to discover fine-grained clusters of sub-patterns. It enables grouping in data to be discovered simultaneously over a variety of scales by constructing a multilevel hierarchical cluster tree where clusters at one level are merged into clusters at the next higher level [113]. There are two reasons to choose a hierarchical clustering algorithm instead of other methods. First, a hierarchical clustering method fits in our problem setting perfectly. One essential function of the proposed work is comparison of program behaviors based on system calls. It looks for similarity of two behaviors instead of determining the equation of them. Hierarchical clustering algorithm takes the input of distance vector of objects and returns similar objects in clusters. There are other data-mining approaches such as *maximal frequent item sets mining* [114] and *k-means cluster analysis* [113]. The frequent item set mining algorithm is more suitable for a data set with distinct items. There is no overlap (or similarity) between any pair of distinct items. This algorithm usually looks for the frequency of unique items occurring in a data set or the frequency of a sequence of unique item sets. The k-means clustering algorithm is a partitioning method and it outputs a *single* level of clusters [113]. Irrelevant objects can still be grouped into one cluster [113]. Second, hierarchical clustering allows users to decide what level or scale of clustering is most appropriate in their applications. Therefore, it enables a customized configuration. In this proposed work, the variable `cutoff` is used. The value of `cutoff` decides the scale of clustering of similar sub-patterns.

Hierarchical Clustering

Let a coarse-grained cluster which is generated during the pre-clustering phase be denoted C and the corresponding similarity score matrix be denoted S. The similarity of a pair of sub-patterns i and j in C can be accessed through the similarity score matrix S. The corresponding entry is denoted $S_{i,j}$.

The objective of hierarchical clustering in this work is to find the groups of similar sub-patterns in C. A group of similar sub-patterns in C is a fine-grained cluster and is denoted F. The k^{th} such group is denoted F_k. We choose and implement the *average-*

linkage agglomerative hierarchical clustering algorithm [115]. There are single-linkage and complete-linkage clustering algorithms, which evaluate cluster quality based on a single pair of data items (sub-patterns). The average-linkage clustering algorithm, in contrast, evaluates cluster quality based on all similarities among data items. It can avoid the pitfalls of the single-linkage and complete-linkage algorithms [115]. (A detailed introduction to hierarchical clustering algorithms can be found at [115].)

A clustering process starts to find pairs of sub-patterns i and j in C that have the shortest distances measured as 1 - $S_{i,j}$ and merges them into binary clusters. It then iteratively merges these newly formed clusters or new sub-patterns to form bigger clusters. Each time, a pair of clusters, or a pair of sub-patterns, or a pair of a cluster and a sub-pattern, are merged when they have the shortest distance among all pairs of none merged clusters and single sub-patterns. The distance between two clusters is a group distance which is measured as the average of distances among all pairs of data items, where each pair is made up of one data item from each group. This iterative clustering process stops when the shortest distance of all pairs of none merged clusters and single sub-patterns is greater than a threshold, or there is only one fine-grained cluster that remains. We use the variable `cutoff` to represent this threshold and term it the ***clustering parameter***, which is customizable.

Upon the conclusion of the clustering process, there are single non-clustered sub-patterns and a set of fine-grained clusters (F) which consist of similar sub-patterns. Only these fine-grained clusters can be used for constructing the common behavior pattern. Single non-clustered sub-patterns are discarded.

If a fine-grained cluster F_k is eventually chosen for constructing the common behavior pattern, a sub-pattern i in F_k, denoted U_i^k, will be chosen to represent F_k. The representative sub-pattern U_i^k has the minimum average of distances with the other sub-patterns in F_k.

Constructing Common Behavior Patterns.

To construct the common behavior pattern of the sample malware instances, popular fine-grained clusters are found first, and then the representative sub-patterns of these fine-grained clusters are used to compose the common behavior pattern.

Each fine-grained cluster has a *popularity*. Popular fine-grained clusters are found if the values of their popularity exceed a threshold. We use the variable `po` to represent this threshold and term it the ***popularity parameter***. The popularity parameter `po` is customizable.

Taking the mined fine-grained clusters and the original semantic program patterns as input, the *popularity* of a fine-grained cluster is measured as the number of **unique** program patterns which contain the sub-patterns of the fine-grained cluster, compared to the total number of the program patterns.

Malware Detection Using a Common Behavior Pattern

This subsection briefly discusses how a common behavior pattern is used to detect a malware instance. Such detection is a pattern matching process whose approach is the same as the pattern matching approach of [111], except that the calculation of a pattern matching score is slightly different.

A code fragment k is given, where k may be a malware instance. The pattern for k has been computed by the static analyzer and is represented as $P^k = \{U_1^k$, ..., $U_{N_k}^k\}$, where the i^{th} sub-pattern is represented as U_i^k. A common behavior pattern is given and is represented as $P^l = \{U_1^l$, ..., $U_{N_l}^l\}$.

To recapitulate, a *pattern matching* of P^k and P^l is a one-to-one assignment from the set of sub-patterns of P^k to the set of sub-patterns of P^l. Among all possible matchings, we find the maximum weighted matching which is one that maximizes the sum of the similarity scores of the pairs of sub-patterns that are matched.

$$M(P^k, P^l) = \frac{\sum_{\langle U_i^k, U_j^l \rangle \in W} score(U_i^k, U_j^l)}{N_l} \tag{5.1}$$

Our technique uses Formula 5.1 to calculate the value or score produced by a maximum weighted matching W. Formula 5.1 is slightly different from Formula 4.1 in Chapter 4. In a calculation using Formula 5.1, the value or score produced by W is still equal to the mean of the similarity scores of pairs of sub-patterns that are present in that matching. However, the denominator is N_l instead of $max(N_k, N_l)$. The justification is that the detection of a malware instance using a common behavior pattern is an estimation of how much the given common behavior pattern is matched to the malware instance.

In the next section, a set of experiments are described to evaluate the proposed work.

5.3 Evaluation

A set of experiments on real world malware and benign executables were performed to evaluate the proposed approach.

5.3.1 Data Collection

A total of 1234 bot programs were obtained from Cyber-TA project's website [18] and Jhu bot research project [112]. These programs were detected as malicious bot programs by Symantec anti-virus software [116]. These programs include different types of bot programs like `spybot`, `ircbot`, `ircbot.gen`, `linkbot`, `gobot`, etc., as indicated by the Symantec anti-virus tool [116]. Different brands of anti-virus software may have different names or classifications. (For example, `jrbot` is indicated as `ircbot.gen` by the Symantec anti-virus tool [116].) Among these bot programs, 340 programs could not be processed due to file access permission being denied. The blocking of file access is a self-defense technique as pointed out by [1]. In addition, there are 24 programs that have corrupted program headers and 600 programs that have obfuscated program entries. These include basically three types - the program entry virtual address is 0, it is located outside of code section, or the calculated code size is 0. These 600 programs can be accessed and processed. However, the results are expected to be inaccurate. In this sense, more advanced disassembly techniques need to be investigated. The other 270 programs could be processed. However, only 167 of them have non-zero patterns. The reasons for the bot programs having zero-patterns will be investigated. Our suspicion is that they use encryption. This technique does not handle encryption or any self-modifying code.

Twenty five bot programs were chosen as the data set for common pattern generation. The quality of generated patterns was investigated, and the results will be presented in the next subsection. Two set of tests were performed. The first test was to evaluate the detection capability of a generated common malware pattern to identify known or new malware instances that are from the same or similar malware categories. The second test was to to evaluate the true false positive rate, or how often the generated common behavior falsely flags benign programs as malicious. In the first test, an additional 25 bot programs were used. Therefore, a total of 50 bot programs were used for detection capability evaluation. In the second test, 313 benign windows programs and applications that are processable executables were used as the test data set. The 313 benign programs include executables (in PE format) under C: \ windows \system32 directory of our test machine running Windows XP professional and application executables.

Out of the 167 processable bot programs, the 25 programs are probably the maximum processable data set for pattern generation. This is due to a system restriction on file size. During the pattern generation process using 167 processable bot programs, the files of similarity scores for several coarse-grained clusters reached 4G bytes. No further similarity score values could be inserted to a file that exceeds 4G bytes. If we use a different data set with the average pattern size being smaller, the size limit on the number of programs for pattern generation could be higher.

5.3.2 Quality of Generic Pattern

A common behavior pattern is generated from the 25 bot binaries using the described method. Then this common behavior pattern is applied to detect the 50 bot binaries and 313 benign programs to evaluate its detection capability and false positive rate.

In total, 45 patterns were generated under different parameter configurations, using the described method. The pre-clustering approach followed Scheme 2. Two such parameters were the clustering parameter `cutoff` and the popularity parameter `po`. The parameter `cutoff` is used in hierarchical clustering process to determine the groups of similar objects (sub-patterns). The distance of two sub-patterns in a fine-grained cluster is less than the value of `cutoff`, or the similarity score value of two sub-patterns in a fine-grained cluster is greater than 1 - the value of `cutoff`. The parameter `po` is used in the selection of popular fine-grained clusters.

The 45 patterns were generated under different combinations of `cutoff`, whose value was chosen from set {0.1, 0.2, 0.3, 0.4, 0.5}, and `po`, whose value was chosen from {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}. Each generated pattern was compared to the tested data sets of the 50 bot programs and the 313 benign executables in Windows PE format, using the maximum weighted matching approach as discussed in the previous section. For each comparison a similarity score was computed. Then the third parameter `d` was used. The parameter `d` is called ***detection threshold parameter***. If the value of a similarity score

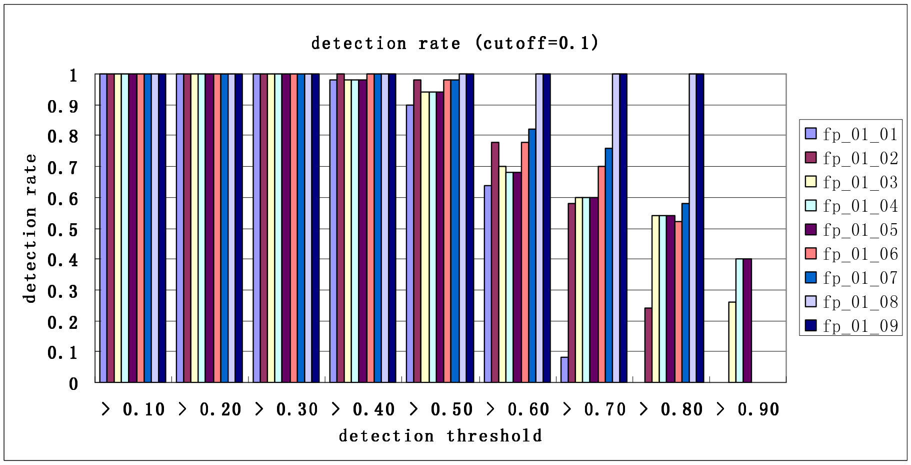


Figure 5.5: Detection rate for `cutoff` = 0.1.

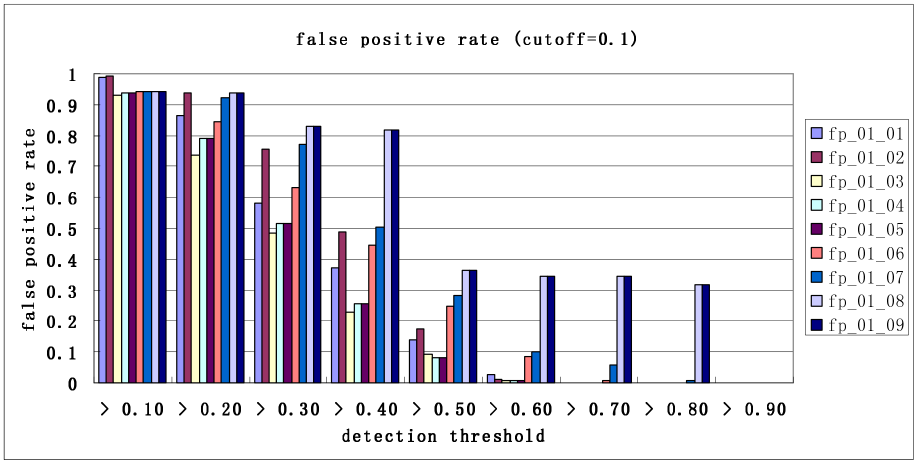


Figure 5.6: False positive rate for `cutoff` = 0.1.

is above the value of `d`, then the corresponding tested program was regarded as a successful match to the generated pattern. Otherwise, it was not a successful match. Therefore, the detection rate of a generated pattern was calculated as the number of successful matches of the 50 bot programs to the generated pattern. The false positive rate of a generated pattern was calculated as the number of successful matches of the 313 benign programs to the generated pattern.

Figure 5.5 to Figure 5.14 plot the results. An optimized result, under a configuration of `cutoff` =0.1, `d` =0.5, and `po` =0.4∼0.5, has a detection rate and false positive rate of 94% and 8.3% respectively. The second optimized result, under a configuration of `cutoff` =0.2, `d` =0.6, and `po` =0.4, has a detection rate and false positive rate of 78% and 0.32%

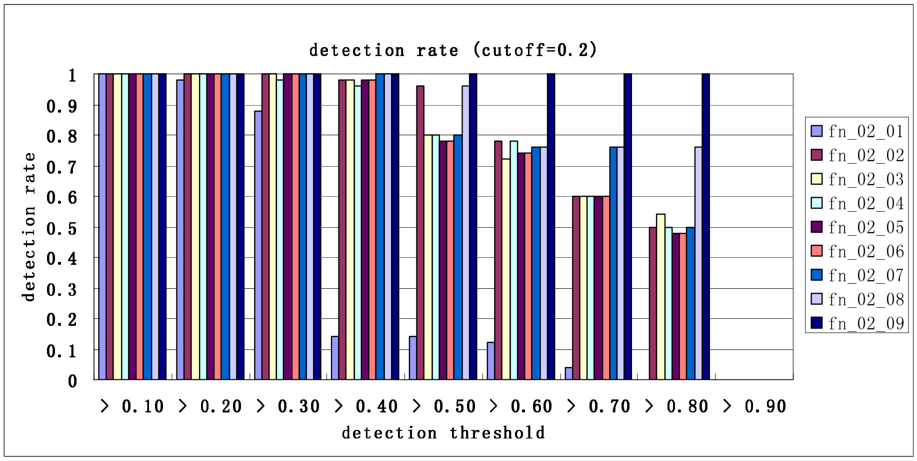


Figure 5.7: Detection rate for `cutoff` = 0.2.

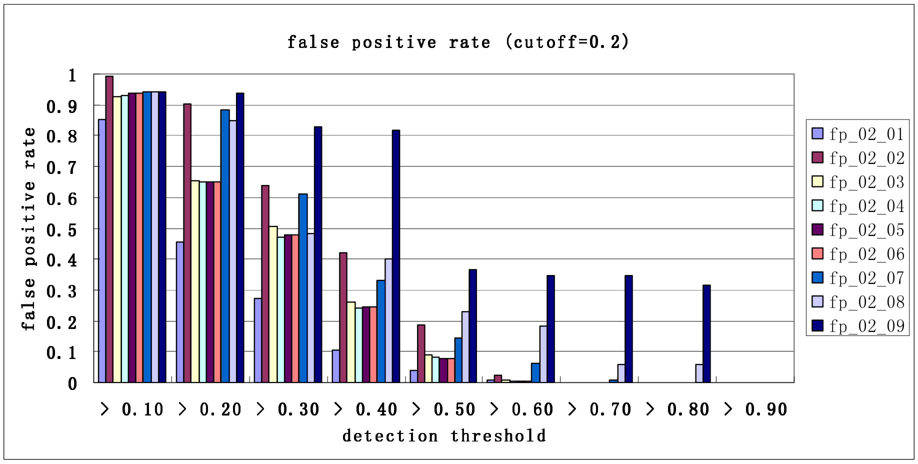


Figure 5.8: False positive rate for `cutoff` = 0.2.

respectively. An optimized result should have a maximal detection rate and a minimal false positive rate.

The following is a simple description of how to find the parameter configuration that produced the two optimized results in this experiment. A more general discussion is presented in the next section. First, a target performance was set. In this experiment, an optimized result was expected to have the false positive rate of less than 10% and the detection rate of over 70%. Second, unqualified results were discarded from consideration. This shrinks the search space. One quick examination of the results plotted in Figures 5.5 to 5.14 finds that only the results with detection threshold `d` no smaller than 0.5 could possibly be optimized. Then the search starts. The basic idea is to find "locally" optimized results

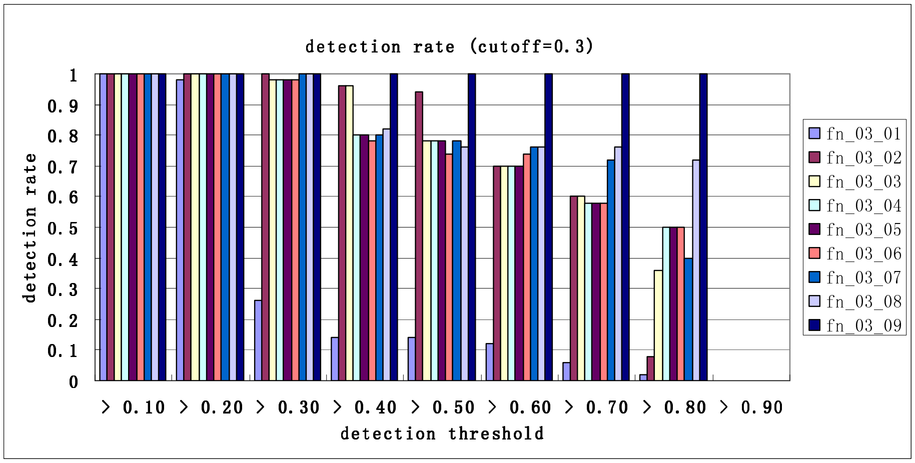


Figure 5.9: Detection rate for `cutoff` = 0.3.

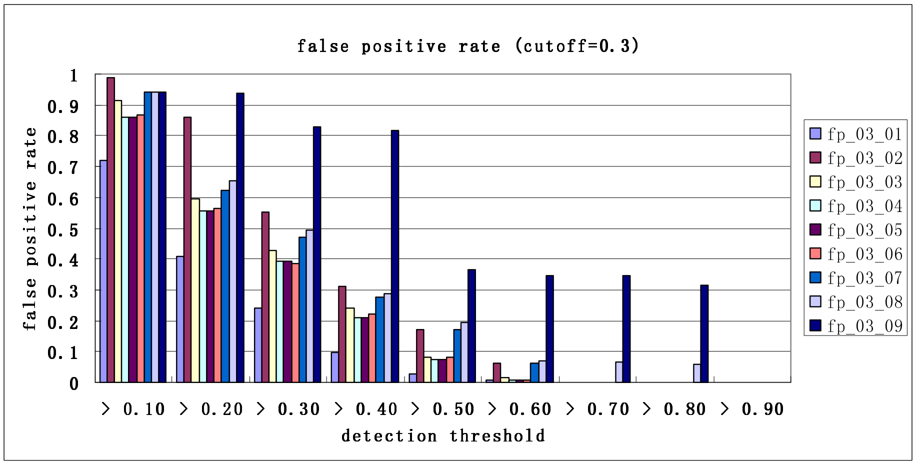


Figure 5.10: False positive rate for `cutoff` = 0.3.

and then compare these results to get the "globally" optimized ones. The local and global scopes are relative concepts. A global scope of results includes the results that are generated under all combinations of parameter configurations. A local scope of results refers to those that are generated with at least one identical parameter configuration. If such a parameter is x, then a local scope is based on x. In this example, a local scope is based on the parameter `cutoff`. The initial search examined the local scope of results with `cutoff`=0.1, which are plotted in Figure 5.5 and Figure 5.6. The qualified results in that scope are shown in Figure 5.15. The locally optimized results and their configurations are shown as the shadowed entries in Figure 5.15. The process repeated to examine the local scope of results with `cutoff`=0.2, which are plotted in Figure 5.7 and Figure 5.8. Likewise, the

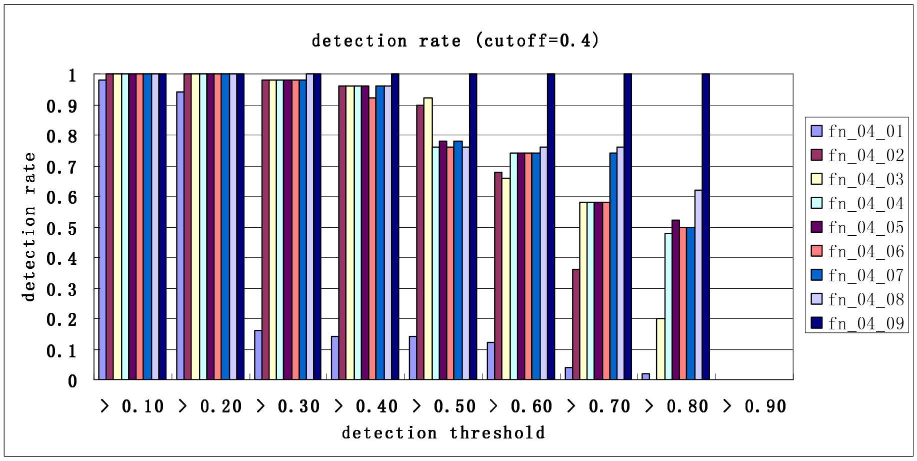


Figure 5.11: Detection rate for `cutoff` = 0.4.

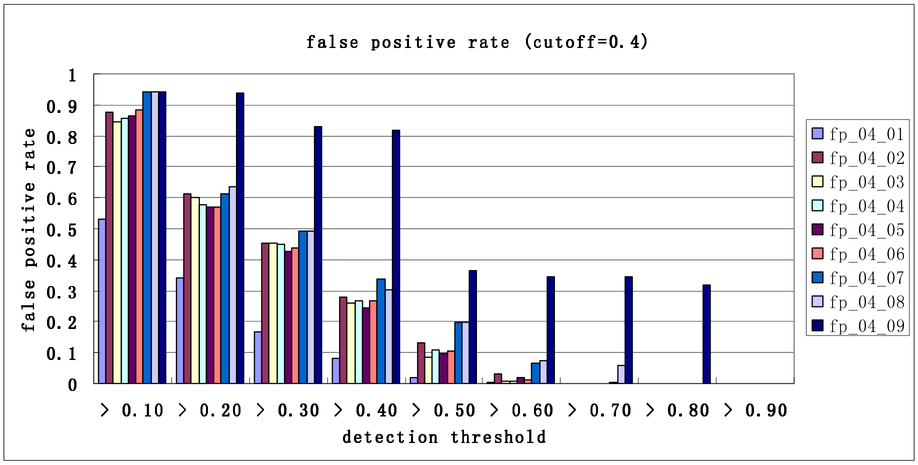


Figure 5.12: False positive rate for `cutoff` = 0.4.

qualified results are shown in figure 5.16. The locally optimized results with `cutoff` being 0.2 and their configurations are shown as the shadowed entries in figure 5.16. Eventually, among the five distinct locally optimized results which are shown as the shadowed entries in Figure 5.15 and Figure 5.16, two optimized results were found. The detection rates and false positive rates are 94%, 8.3%, and 78%, 0.32% respectively. The configurations are `cutoff` =0.1, `d` =0.5, `po` =0.4∼0.5, and `cutoff` =0.2, `d` =0.6, `po` =0.4, respectively.

5.3.3 Comparison of the Three Pre-clustering Schemes

The three pre-clustering schemes as discussed in section 5.2.2 are now compared. A similar experimental approach to the one as discussed in the previous subsection is taken.

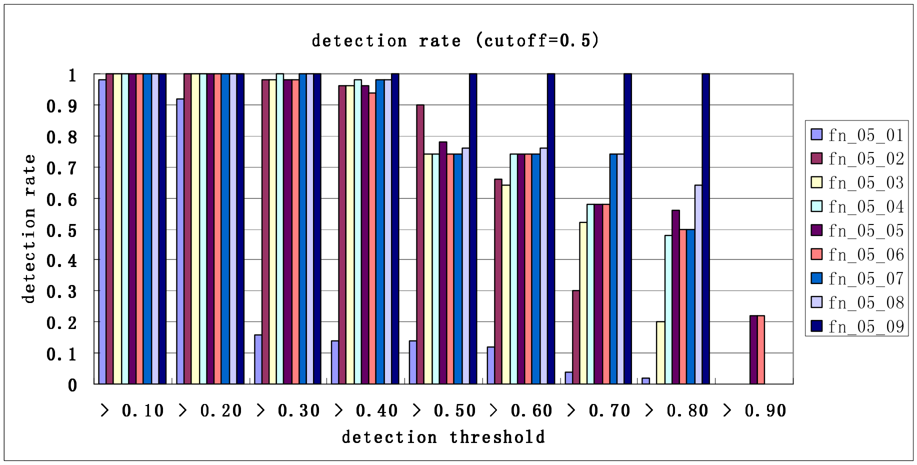


Figure 5.13: Detection rate for `cutoff` = 0.5.

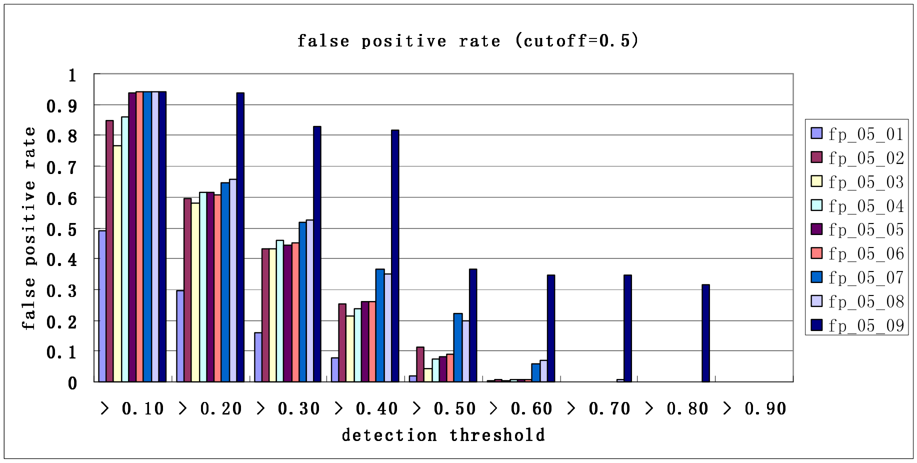
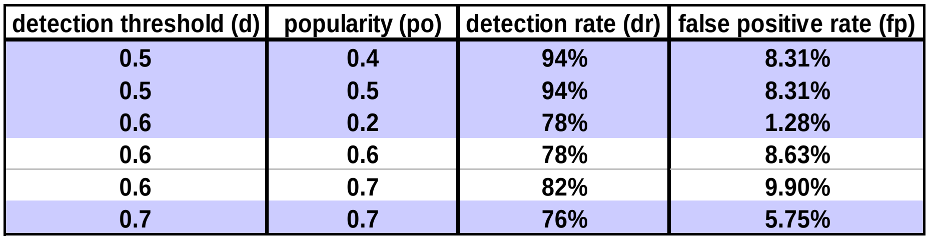


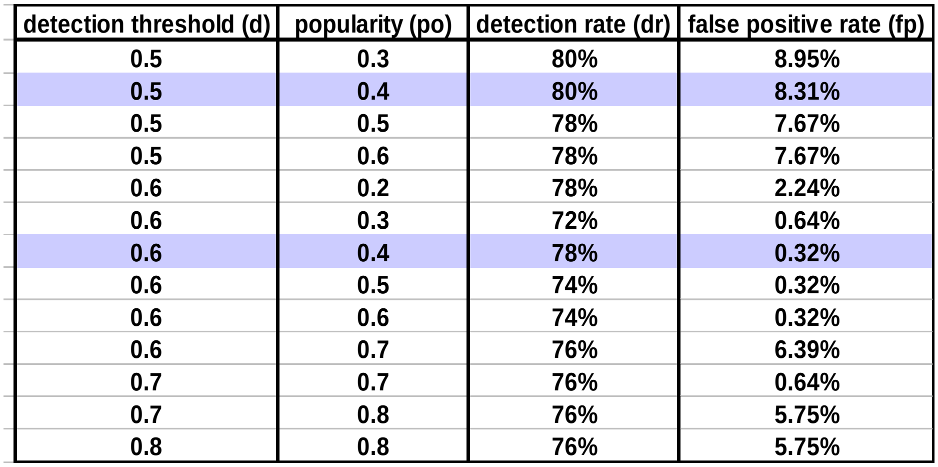
Figure 5.14: False positive rate for `cutoff` = 0.5.

The three pre-clustering schemes are compared under the configuration of `cutoff` =0.2 and `po` =0.4. An optimized result was derived under this configuration in the experiment as presented in the previous subsection. The data set that was used to generate common behavior patterns is different from the one used in the previous experiment. The number of the data items (i.e. the bot binaries) is reduced from 25 to 16. One reason for this is due to the system restriction on file size that should not exceed 4`G` as explained in a previous subsection. The other reason is that the use of pre-clustering Scheme 1 produces unnecessary computations that quickly explode the files of similarity scores. Hence, with the current implementation the number of the data items has to be reduced in order to compare the effectiveness of the three pre-clustering schemes under the same test conditions. The



detection threshold (d)	popularity (po)	detection rate (dr)	false positive rate (fp)
0.5	0.4	94%	8.31%
0.5	0.5	94%	8.31%
0.6	0.2	78%	1.28%
0.6	0.6	78%	8.63%
0.6	0.7	82%	9.90%
0.7	0.7	76%	5.75%

Figure 5.15: Qualified results with `cutoff`=0.1 and their configurations. The entries for locally optimized results are shadowed.



detection threshold (d)	popularity (po)	detection rate (dr)	false positive rate (fp)
0.5	0.3	80%	8.95%
0.5	0.4	80%	8.31%
0.5	0.5	78%	7.67%
0.5	0.6	78%	7.67%
0.6	0.2	78%	2.24%
0.6	0.3	72%	0.64%
0.6	0.4	78%	0.32%
0.6	0.5	74%	0.32%
0.6	0.6	74%	0.32%
0.6	0.7	76%	6.39%
0.7	0.7	76%	0.64%
0.7	0.8	76%	5.75%
0.8	0.8	76%	5.75%

Figure 5.16: Qualified results with `cutoff`=0.2 and their configurations. The entries for locally optimized results are shadowed.

comparison results are shown in Figure 5.17 and Figure 5.18. The optimized results using the three pre-clustering schemes are achieved under $\mathtt{d} = 0.6$ and $\mathtt{d} = 0.7$. In either configuration, the proposed method using the pre-clustering Scheme 2 produces better results in terms of a higher detection rate and a lower false positive rate. As discussed in section 5.2.2, overly generic or overly specific pre-clustering schemes could lead to poorer performance.

5.3.4 Comparison with our Previous Work

In this subsection, we compare the technique proposed in this chapter with our previous work that is presented in Chapter 4. We first highlight the difference between the two techniques and then introduce the experiment which was performed to quantify the difference.

The technique presented in Chapter 4 addresses the problem of how to summarize and compare program semantics between *a pair* of programs (or executables). A semantic malware pattern generation and matching method was proposed in Chapter 4. This tech-

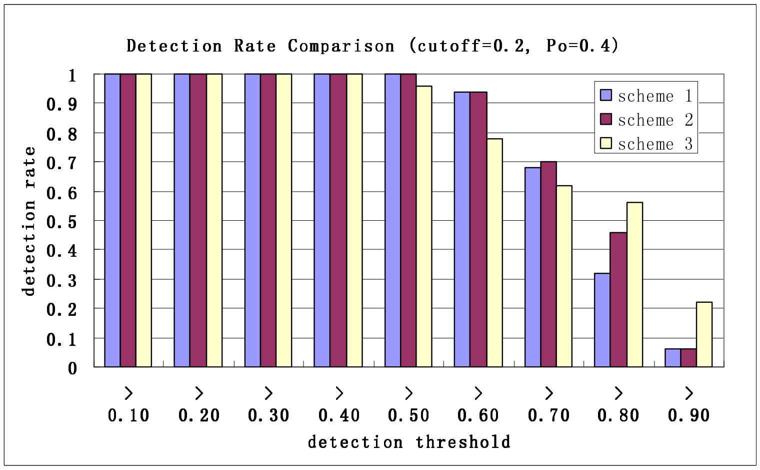


Figure 5.17: Detection rate using the generated common pattern from 16 bot executables.

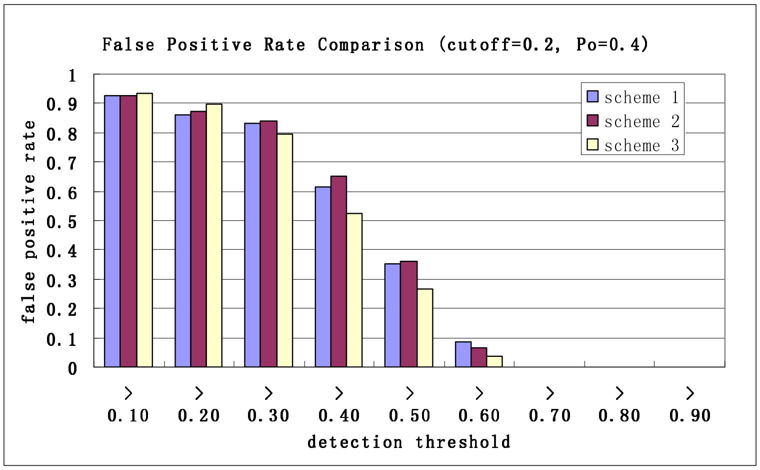


Figure 5.18: False positive rate using the generated common pattern to test on 313 benign executables.

nique can serve as a basis for the detection of metamorphic malware which has equivalent or updated functionalities. This technique works by using one malware instance's pattern to detect the variants of that malware. However, this technique is not intended to detect other malware instances from a totally different malware category, although sometimes it is able to do so when the two different types of malware have behaviors in common.

In contrast, the technique presented in this chapter focuses on generalizing the semantic pattern generation and matching approach. It aims to use a single *common* malware behavior pattern to detect an entire category of malware as well as those are from a different category.

Therefore, the technique presented in this chapter can potentially result in great reduction in semantic pattern population for the detection of known and new malware instances. We performed a new experiment to quantify this amount. In the previous subsection, the experiment results show using ***one*** common behavior pattern which was generated from 25 training bot programs can achieve **78%** detection rate on the 50 testing bot programs with the detection threshold `d` $= 0.6$.

For comparison, the new experiment measures the number of semantic program patterns that are needed for the technique of Chapter 4 to achieve *the same detection rate* (i.e., 78%) using the same data set (i.e., the 25 training bot programs and the 50 testing bot programs) under the same testing condition (i.e., `d` $= 0.6$).

The experiment was performed as follows. First, for each of the 25 training bot programs, a pattern was generated using the method presented in Chapter 4. Second, the detection capability of each pattern was measured. For instance, we recorded the successful matches between the pattern and the patterns of the 50 testing bot programs. Two patterns are matched, or a pattern is detected by the other, if their similarity score is above the value of `d`, which is 0.6. Third, we found the *minimum* number of patterns that can *cumulatively* detect 78% of the 50 testing bot programs. Here are the findings. Using one pattern (the best one) only 32% of the testing malware programs were detectable. Using two patterns 62% were detectable. Using three patterns 72% were detectable. Using four patterns 76% were detectable. Once we used ***five*** patterns, the target detection rate 78% was achieved. In real practice, there may be problems to measure the minimum number of patterns that can cumulatively achieve a target detection rate. Then a solution is to find the average situation. For instance, we can measure the detection rate of using x number of patterns out of a total of n available patterns and calculate the average detection rate of using all such combinations. The comparison result shown here is conservative. The average number of patterns that are needed to achieve the same detection rate as that of using a common behavior pattern under the same test condition should be higher than the minimum one.

To summarize, through the simple experimental comparison, the technique presented in this chapter results in about 80% reduction in semantic pattern population to detect known and new malware instances.

5.4 Discussion

5.4.1 Data Set Selection

There are practical issues in deploying the proposed work for the generation of good quality malware patterns. The data set selection is one of them. If there are overwhelming numbers of irrelevant samples in the training data set that is used for the generation of a common pattern to detect new malware instances, the quality of the generated pattern is expected to be much lower. These irrelevant samples are background "noise." Data set selection is a meaningful and open problem challenging many data-mining related research. For different application domains, there could be context dependent solutions. A generic popular solution to this problem is applying *genetic algorithms* [117] or evolutionary algorithms with domain specific individual selection functions to improve the overall fitness of the population. A recent paper has developed an evolutionary algorithm in the vulnerability analysis testing domain [118]. For the present work, we can design domain specific genetic algorithms to choose fitted data sets with the existing subpattern matching approach as the basis of individual case selection. This, however, is left as our future work.

5.4.2 Optimized Configuration Selection

The second practical issue is how to configure multiple parameters to produce an optimized result. Again, designing genetic algorithms is a possible solution. Genetic algorithms are most commonly applied in *multiparameter function optimization* which can be formulated as a search for an optimal value where the value is a complicated function of some input parameters [117]. With regard to our specific problem, there is a simple solution which can be described as *objective performance directed parameters selection.* Regarding the pattern generation, there are two parameters - the clustering parameter `cutoff` for determining the scope of a fine-grained cluster which holds similar sub-patterns, and the popularity parameter `po` for choosing popular fine-grained clusters. There is also the detection threshold parameter **d**.

To find an optimized configuration, two phases are involved including the *pattern generation*, and *pattern selection.* The pattern generation phase creates patterns that could be of good quality. The pattern selection phase performs the actual search within the set of patterns to find the ones that produce optimized results in terms of maximal detection rate and minimal false positive rate.

A set of patterns are generated first under different configurations of `cutoff` and `po`. A pattern generated under the configuration of `cutoff` $= i$ and `po` $= j$, is denoted $P_{i,j}$. The detection rate and false positive rate of $P_{i,j}$ with the detection threshold `d` $= k$ are denoted $dr_{i,j,k}$ and $fp_{i,j,k}$ respectively. The values of `cutoff`, `po`, and `d` are recommended to be selected from a *finite* set of floating numbers ranging from 0.1 to 0.9.

The problem of optimized configuration selection then can be formalized as a search to find a pattern $P_{i,j}$ such that $dr_{i,j,k}$ is maximal and $fp_{i,j,k}$ is minimal, where i,j, and $k \in 0.1 \sim 0.9$. A pattern $P_{i1,j1}$ is an equal or better choice than pattern $P_{i2,j2}$ when their detection rates ($dr_{i1,j1,k1} \geq dr_{i2,j2,k2}$) and their false positive rates ($fp_{i1,j1,k1} \leq fp_{i2,j2,k2}$) show those relationships. However two patterns are incomparable when their detection rates ($dr_{i1,j1,k1} > dr_{i2,j2,k2}$) and their false positive rates ($fp_{i1,j1,k1} > fp_{i2,j2,k2}$, or $dr_{i1,j1,k1} < dr_{i2,j2,k2}$ and $fp_{i1,j1,k1} < fp_{i2,j2,k2}$) show those relationships.

To minimize the number of incomparable patterns to find the optimized configuration, a target performance can guide the selection. A generated pattern is considered valid when the detection rate it produces is equal to or greater than the targeted detection rate, and the false positive rate it produces is less than or equal to the targeted false positive rate. An example to illustrate this general idea has been given in section 5.3.2.

5.4.3 Special Cases

The previous subsection discusses the impact of parameter configuration on the quality of a generated pattern. This subsection discusses the impact of the nature of the training data set on the quality of a generated pattern.

There are several special cases of the data set that is used for pattern generation. The quality of a generated pattern is affected greatly by the nature of these data sets.

The data set consisting of totally irrelevant binaries

In this case, no matter how we choose to configure the three parameters, there are expected to be no non-trivial patterns generated. For instance, a generated pattern may consist of only one behavior that is common to the sample programs, i.e., a single call to `printf`. However, the awareness of the fact that a generated pattern is trivial can be achieved through the *pattern selection* procedure as described in the previous subsection.

Upon the conclusion of a pattern selection, no optimized patterns could be found. Then a better data set should be selected.

The data set consisting of completely functionally equivalent binaries

This special case resembles the application scenario of our second proposed work. The generation of a pattern from the data set is actually the generation of a pattern of metamorphic malware. No matter how we choose to configure the three parameters, a generated pattern is expected to summarize the entire program behaviors of these binaries. The false detection rate of a generated pattern will be very low, which is good. However, the detection rate of a generated pattern on functionally similar binaries will be low, which is not good.

The data set consisting of benign programs with embedded malicious functions

This case depicts a new application scenario of the proposed work. If the data set consists of benign programs with *common* embedded malicious functions, the proposed method works normally. As long as the embedded malicious functions have popular occurrence in the benign programs. The proposed method could still generate an accurate common behavior patterns.

The data set consisting of malicious binaries that have common behaviors generated by popular software tools

These common behaviors are not necessary being “malicious”. They can also occur in benign programs. One such case is that a lot of programs are developed using SDKs or SDEs (e.g., MFC). These non-malicious behaviors of the malware samples can be found to compose a common malicious behavior pattern. The “non-malicious” and “common” behaviors shared by both malicious and benign programs will not result in a significant reduction of the detection capability of the generated common malicious behavior pattern. However, it would probably increase the true false positive rate.

A solution to this problem is to mine the common behavior of benign programs, and remove that from use in malware analysis. This leaves to our future work.

5.4.4 Limitations

There are several limitations to the proposed work.

Limitation of static analysis

The first limitation stems from the limitations of static analysis. Malware self-defense techniques can challenge and thwart the static analysis. There are several such techniques as pointed out by an interesting article [1]. Some of them, including using packed malware binaries and blocking access to files, are encountered in our experimental evaluation. Other program obfuscation techniques like delicate system call obfuscation are not handled by our static analyzer, but a solution has been proposed by A. Lakhotia et al. [119]. Therefore, a more sophisticated technique, probably integrating different approaches designed for specific obfuscation or behaviors, is needed to strengthen the state-of-art malware analysis and defense.

Limitation of data-mining based approach

The second limitation stems from the data-mining approach whose accuracy depends on the training data set. Theoretically, there is no formal method that is able to determine the semantic equivalence of two abstract behaviors. This is an undecidable problem. Therefore, it is always theoretically possible to confuse, bypass, or poison the data-mining approach that is based on an approximated measurement or estimation of the equivalence of program behaviors by exploiting its similarity evaluation algorithm. In the current case, an attacker can insert junk functions or behaviors in each malware instance in the sample malware pool to increase the percentage that these behaviors can be discovered and included in a pattern. The attacker later produces new malware instances without incorporating these junk behaviors. Therefore, the new malware instances will not be matched to the generated pattern in the old training set.

Fundamentally, there is no solution to defeat this attack if we assume attackers can control the training data set selection. However, there is a practical issue for the attacker - how he can control or influence the data set choice. Furthermore, a mitigation approach can be proposed, using as *large* and *diversified* while *related* malware data set as possible to lower the percentage of the background noise. In addition, research to develop better techniques (i.e, combining dynamic analysis) to recognize the necessary conditions for the malware to succeed can help develop a more accurate pattern [120]. This is left as an open question.

5.5 Summary

This chapter presented an automated approach to discover a common behavior pattern from a set of malware for detection of metamorphic malware or new malware instances. This approach can improve the applicability of a semantic malware detector [111]. The proposed approach combines static analysis and data-mining techniques. It has been prototyped and evaluated using real world malicious bot software and benign Windows programs.

Through the experimental comparison with the metamorphic malware detector, this method results in an about 80% reduction in semantic pattern population to detect known and new malware instances. It is more robust to a junk behavior pollution attack than the malware detector is. A set of experiments was performed to test the quality of the common behavior patterns which were generated with different parameter configurations. Two criteria were used to evaluate the quality of a common behavior pattern. One was the detection capability of the generated common behavior pattern to identify known or new malware instances that are from the same or similar malware categories. The other was the true false positive rate, or how often the generated common behavior falsely flags benign programs as malicious. Two optimized common behavior patterns were obtained. The corresponding detection rates and true false positive rates are 94%, 8.3%, and 78%, 0.32% respectively.

Future work will study how to automatically choose suitable training sets to improve the common behavior pattern generation approach. It will further study to how to remove the common benign program behavior from use in the malware analysis. Moreover, it will study how to improve the program behavior extraction process that is the key to the work proposed in this chapter. More advanced techniques will be researched in addition to the static analysis used in this work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation addresses the problems of how to detect polymorphic and metamorphic malicious software. The attacks caused by these malware cause serious problems and they can evade anti-malware software. This dissertation makes the following three contributions:

1. *A new approach for recognizing polymorphic exploits that are encrypted and that self-decrypt before launching the attacks in network traffic*: Remotely-launched exploits are a common way for attackers to intrude into vulnerable systems and gain control of them. As remote exploitation techniques evolve, polymorphic remote exploits that are encrypted and that self-decrypt before launching the intrusion pose a great challenge to existing malware detection techniques, partly due to the non-obvious starting location of the exploit code in the network payload. To detect them, the proposed method scans network traffic for the presence of a decryption routine which is characteristic of such exploits. It does this by combining static analysis and instruction emulation techniques. The proposed method outperforms previous proposals [27, 28, 34, 29, 30] in its capability to identify more precisely the starting location of the decryption routine, with fewer assumptions. The method also can identify the decryption routine even if self-modifying code has been used to conceal its presence. This method has been implemented and tested on current polymorphic exploits, including ones generated by state-of-the-art polymorphic engines. All exploits have been detected (i.e., a

100% detection rate), including those for which the decryption routine is dynamically coded, or self-modifying. The false positive rate is close to 0%. Running time is approximately linear in the size of the network payload being analyzed.

2. *A new approach for recognizing metamorphic malware*: Metamorphic malware uses various code obfuscation techniques to transform its program image from its early version. It has equivalent or updated functionalities and it can easily evade the conventional signature based anti-malware software. We propose a new approach that uses fully automated static analysis of executables to summarize and compare program semantics, based primarily on the pattern of library or system functions which are called. This method has been prototyped and evaluated using randomized benchmark programs, instances of known malware program variants, and utility software available in multiple releases. The evaluation results demonstrate three important capabilities of the proposed method: (a) it has great promises in identifying metamorphic variants of common malware: the measured similarity score of an original benchmark program and its randomized version in most cases achieves a value of .95 or greater; (b) it distinguishes easily between programs that are not related and, (c) it can identify and detect program variations, or code reuse: the measured similarity scores of different releases of the GNU binutil programs can achieve .75 or better. Such variations can be due to insertion of malware (such as viruses) into the executable of a host program or program revision.

3. *An automated approach that generates common malware behavior patterns for recognizing metamorphic malware or new malware instances*: It is a challenging and non-trivial problem to effectively apply a metamorphic malware detector like the second work of this dissertation. Given the overwhelming number of malware types and the malware mutants, it would have had to generate a semantic pattern for each type of malware mutants. This would be a heavy burden for maintenance. We propose an automated approach to generate common malware behavior patterns for detection of metamorphic malware or new malware instances. This method combines static analysis and data-mining techniques. This method has been prototyped and evaluated on real world malicious bot software and benign Windows programs. Through the experimental comparison with the metamorphic malware detector, this method results in an about 80% reduction in semantic pattern population to detect known

and new malware instances. It is more robust to a junk behavior pollution attack than the malware detector is. A set of experiments was performed to test the quality of the common behavior patterns which were generated with different parameter configurations. Two optimized common behavior patterns were obtained. The corresponding detection rates and true false positive rates are 94%, 8.3%, and 78%, 0.32% respectively.

6.2 Future Work

Our future work will address the limitations of the three works proposed in this dissertation.

1. *Study new techniques to detect new polymorphic malware attacks*: There are ways that the proposed method can be bypassed such as using lengthy loops or using running-time-environment related values in a polymorphic exploit. To do so, attackers need to carefully craft their exploit code. Nevertheless, it is worth further study of new techniques to detect these polymorphic exploits. We will focus on generalizing the method for less obvious sequences of byte decoding.

2. *Study new techniques to enhance the accuracy of static analysis*: Static analysis is the basis of the three works proposed in this dissertation. In particular, it is the key to successfully characterizing program behaviors for metamorphic malware detection. Therefore, it is important to enhance its accuracy. Here we have several ideas to improve our static analysis approach that characterizes program behaviors based on system calls. (There could be same or similar ideas that have been proposed. We need to investigate them.)

    - *Combine localized dynamic analysis*: Many techniques can thwart static analysis such as using self-modifying code or indirect control transfer instructions. Making obfuscated system calls is a technique that specifically bypass our static analysis approach. Using dynamic analysis techniques can address these problems. For instance, we can design techniques to "locally" emulate instruction execution to deal with self-contained behaviors such as self-modifying and the obfuscated system calls. Static analysis can be used to develop information to determine a "local" scope for a round of emulated instruction execution.

- *Incorporating more types of system call parameters into pattern generation*: This will improve the accuracy of pattern generation. Besides the target address of a system call, the function argument(s) of a system call could also be used in pattern generation. The information on what function argument(s) a system call is using can possibly be obtained in two ways: 1) from system call specifications; 2) from instruction compilation heuristics: a parameter is usually passed through *specific* registers, i.e., `eax`.
- *Investigate new approaches that characterize malicious program behaviors beside the system call based ones*: Looping structure for processing various control and commands in malicious bot binaries is a characteristic behavior of bot software. We will study new techniques to automatically learn what instruction sequences can be characteristic of a particular malware type.

3. *Address the deployment issues of the third proposed work*: How to choose a suitable training set is a practical issue to a data mining based research. We will study algorithms that automatically choose suitable training sets to improve the common behavior pattern generation approach. According to [117], studying genetic algorithms with our domain specific individual selection function will be a good direction. We will also study to how to remove common benign program behavior from use in the malware analysis.

Bibliography

[1] Alisa Shevchenko. The Evolution of Self-Defense Technologies in Malware. July 2007. http://www.net-security.org/article.php?id=1028.

[2] Intel Architecture Software Developers Manual. Volume 2: Instruction Set Reference.

[3] G. McGraw. *Software Security: Building Security In.* Addison-Wesley Professional.

[4] Computer Economics. http://www.computereconomics.com.

[5] G. McGraw and G. Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, Sept./Oct. 2000.

[6] Antivirus Defense-in-Depth Guide. August 2004. http://www.microsoft.com/technet/security/guidance/serversecurity/avdind_0.mspx.

[7] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware on the Web. In *Proceedings of the 13th Network and Distributed System Security (NDSS'06) Symposium*, February 2006.

[8] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 33–42, October 2004.

[9] R. Lemos. Study: Unpatched PCs compromised in 20 minutes. August 2004. http://news.zdnet.com/2100-1009_22-5313402.html.

[10] P. Barford and V. Yegneswaran. An Inside Look at Botnets. 2006.

[11] Taxonomy of Botnet Threats, November 2006. http://us.trendmicro.com.

[12] N. Daswani, M. Stoppelman, the Google Click Quality, and Security Teams. The Anatomy of Clickbot.A. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, April 2007.

[13] The Honeynet Project. Know your Enemy: Tracking Botnets, March 2005. http://www.honeynet.org/papers/bots.

[14] K. Chiang and L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, April 2007.

[15] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *ESORICS*, pages 319–335, 2005.

[16] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *Proceedings of the 7^{th} International Symposium on Recent Advances in Intrusion Detection (RAID'04)*, pages 59–81, September 2004.

[17] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of 2006 ACM SIGCOMM Conference*, pages 291–302, September 2006.

[18] Cyber-Threat Analytics (Cyber-TA) Project. http://www.cyber-ta.org/releases/malware/.

[19] J. Gordon. Lessons from Virus Developers: The Beagle Worm History Through April 24, 2004. May 2004.

[20] VX heavens. http://vx.netlux.org.

[21] Snort: an open source network intrusion prevention and detection system, 2005. http://www.snort.org.

[22] Bro Intrusion Detection System, 2003. http://www.bro-ids.org.

[23] The ADMmutate polymorphic engine. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[24] The CLET polymorphism engine. http://www.phrack.org/show.php?p=61&a=9.

[25] Metasploit project. http://www.metasploit.org.

[26] P. Szor. *The Art of Computer: Virus Research and Defense.* Symantec Press, NJ, USA, first edition, 2005.

[27] R. Chinchani and E. Berg. A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 284–308, September 2005.

[28] X. Wang, C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, pages 225–240, July 2006.

[29] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5thInternational Symposium on Recent Advances in Intrusion Detection (RAID'02)*, pages 274–291, October 2002.

[30] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis. In *Proceedings of the 20th IFIP International Information Security Conference (SEC'05)*, pages 375–392, June 2005.

[31] Common vulnerabilities and exposures. http://cve.mitre.org/cve/downloads/full-cve.csv.

[32] J. C Foster and M. Price. *Sockets, Shellcode, Porting, & Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Syngress Publishing, USA, 2005.

[33] U. Payer, M. Lamberger, and P. Teufl. Hybrid engine for polymorphic code detection. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA'05)*, pages 19–31, July 2005.

[34] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA'06)*, July 2006.

[35] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, 2004.

[36] G. Vigna, W. K. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security(CCS)*, pages 21–30, October 2004.

[37] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, May 2005.

[38] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting Signature Learning By Training Maliciously. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, September 2006.

[39] http://www.offensivecomputing.net/.

[40] GNU Binutils. http://www.gnu.org/software/binutils/.

[41] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, pages 255–270, Auguest 2004.

[42] C. Cifuentes, M. Van Emmerik, D. Simon D.Ung, and T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Proceedings of the Workshop on Binary Translation*, pages 12–22, October 1999.

[43] C. Collberg and C. Thomborson. Watermarking, Tamper-Proof, and Obfuscation - Tools for Software Protection. In *IEEE Transactions on Software Engineering*, pages 735–746, Auguest 2002.

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

[45] M. Weber, M. Schmid, M. Schatz, and D. Geyer. A Toolkit for Detecting and Analyzing Malicious Software. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)*, pages 423–431, 2002.

[46] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the the 6th ESEC/FSE*, pages 5–14, September 2007.

[47] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://invisiblethings.org/papers/redpill.html.

[48] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th conference on USENIX Security Symposium (Security'00)*, pages 10–10, August 2000.

[49] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 226–241, May 2005.

[50] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 32–47, May 2006.

[51] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. In *Proceedings of the 14th USENIX Security Symposium (Security'05)*, pages 97–112, August 2005.

[52] X. Wang, Z. Li, J. Xu, M. Reiter, C. Kil, and J. Choi. Packet Vaccine: Black-box Exploit Detection and Signature Generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security(CCS)*, October 2006.

[53] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID05)*, pages 53–64, September 2005.

[54] L. Babai and E. Luks. Canonical Labeling of Graphs. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, 1983.

[55] D. B. West. *Introduction to Graph Theory.* Prentice-Hall, NJ, USA, second edition, 2001.

[56] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186, Auguest 2003.

[57] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[58] M. R. Chouchane and A. Lakhotia. Using Engine Signature to Detect Metamorphic Malware. In *Proceedings of the 4th ACM Workshop on Rapid Malcode*, November 2006.

[59] W. Lee and S. J. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th conference on USENIX Security Symposium (Security'98)*, January 1998.

[60] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of 2001 IEEE Symposium on Security and Privacy (S&P'01)*, pages 156–169, May 2001.

[61] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, 2004.

[62] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of 2001 IEEE Symposium on Security and Privacy (S&P'01)*, pages 144–155, May 2001.

[63] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P'01)*, page 38, May 2001.

[64] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security (NDSS'03) Symposium*, February 2003.

[65] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, October 2003.

[66] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307, October 2004.

[67] S. Sidiroglou and A. Keromytis. Countering Network Worms Through Automatic Patch Generation. In *Research Report*, 2003.

[68] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A Behavioral Approach to Worm Detection. In *Proceedings of the 2nd ACM Workshop on Rapid Malcode*, October 2004.

[69] E. Kirda and C. Kruegel. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, pages 273–288, Auguest 2006.

[70] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, December 2004.

[71] H. Wang, S. Jha, and V. Ganapathy. NetSpy: Automatic Generation of Spyware Signatures for NIDS. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[72] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[73] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium (Security'02)*, pages 191–206, August 2002.

[74] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security(CCS)*, pages 340–353, November 2005.

[75] M. Prasad and T. Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 211–224, June 2003.

[76] W. Li, L. Lam, and T. Chiueh. How to Automatically and Accurately Sandbox Microsoft IIS. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[77] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of 2004 ACM SIGCOMM Conference*, pages 193–204, Auguest 2004.

[78] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'06)*, pages 2–16, May 2006.

[79] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the 13th Network and Distributed System Security (NDSS'06) Symposium*, February 2006.

[80] 2000. http://www.sans.org/resources/idfaq/honeypot3.php.

[81] C. Kreibich and J. Crowcrofto. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (Hotnets II)*, November 2003.

[82] M. Cost, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM symposium on Operating systems principles (SOSP'05)*, pages 133–147, October 2005.

[83] S. Staniford, J. Hoagland, and J. McAlerney. Practical Automated Detection of Stealthy Portscans. In *Journal of Computer Security 10*, Janurary 2002.

[84] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.

[85] H. Kim and B. Karp. Autograph: Toward Automated,Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, August 2004.

[86] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous Payload-based Worm detection and Signature Generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID05)*, September 2005.

[87] O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. 2004.

[88] S. Garriss, M. kaminsky, M. Freedman, B. Karp, D. Mazieres, and H. Yu. RE: Reliable Email. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2005.

[89] M. Xie, H. Yin, and H. Wang. An Effective Defense Against Email Spam Laundering. In *Proceedings of the 13th ACM Conference on Computer and Communications Security(CCS)*, November 2006.

[90] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security (NDSS'00) Symposium*, February 2000.

[91] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[92] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[93] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security(CCS)*, November 2006.

[94] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies–A Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM symposium on Operating systems principles (SOSP'05)*, pages 133–147, October 2005.

[95] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the 12th Network and Distributed System Security (NDSS'05) Symposium*, February 2005.

[96] S. S. Muchnick. *Advanced Comiler Design Implementation.* Morgan Kaufmann Publisher, CA, USA, 1997.

[97] http://www.milw0rm.com/.

[98] Serv-U FTP Server MDTM Command Overflow, Feburary 2004. http://www.osvdb.org/4073.

[99] Microsoft Windows RPC DCOM Interface Overflow, July 2003. http://www.osvdb.org/2100.

[100] Microsoft Windows LSASS Remote Overflow, April 2004. http://www.osvdb.org/5248.

[101] Microsoft ASN.1 Library Bitstring Heap Overflow, Feburary 2004. http://www.microsoft.com/technet/security/bulletin/MS04-007.mspx.

[102] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security(CCS)*, pages 345–354, October 2003.

[103] A. Somayaji S. Forrest, S. Hofmeyr and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[104] eEye Digital Security company. http://www.eeye.com.

[105] Windows Disassembler, 1998. http://www.geocities.com/s̃angcho/.

[106] R. Kath. The Portable Executable File Format from Top to Bottom, 1993. Microsoft Developer Network Technology Group.

[107] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[108] SPEC CPU2000. http://www.spec.org/cpu/.

[109] Apache Web Server. http://httpd.apache.org/.

[110] GazTek Web Server. http://gaztek.sourceforge.net/ghttpd/.

[111] Q. Zhang and D. Reeves. MetaAware: Identifying Metamorphic Malware. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*, pages 411–420, December 2007.

[112] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC '06)*, pages 41–52, October 2006.

[113] Matlab statistics toolbox. http://www.mathworks.com.

[114] Mining Maximal Frequent Itemsets. http://himalaya-tools.sourceforge.net/Mafia/.

[115] Hierarchical clustering. http://nlp.stanford.edu/IR-book/html/htmledition/hierarchical-clustering-1.html.

[116] Symantec anti-virus software. http://www.symantec.com.

[117] Stephanie Forrest. Genetic algorithms. *ACM Computing Surveys*, 28(1):77–80, 1996.

[118] S. Sparks, S. Embleton, R. Cunningham, and C. Zhou. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*, pages 477–486, December 2007.

[119] A. Lakhotia, E. Uday Kumar, and M. Venable. A Method for Detecting Obfuscated Calls in Malicious Binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968, 2005.

[120] M. V. Gundy, H. Chen, Z. Su, and G. Vigna. Feature Omission Vulnerabilities: Thwarting Signature Generation for Polymorphic Worms. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*, pages 74–85, December 2007.

[121] T. Swan. *Mastering Turbo Assembler*. SAMS Publishing, IN, USA, second edition, 1995.

[122] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. In *Technical Report 148, Department of Computer Science, University of Auckland, U. C. Berkeley*, July 1997.

[123] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Lecture Notes in Computer Science*, 2001.

Appendices

Appendix A

Appendix

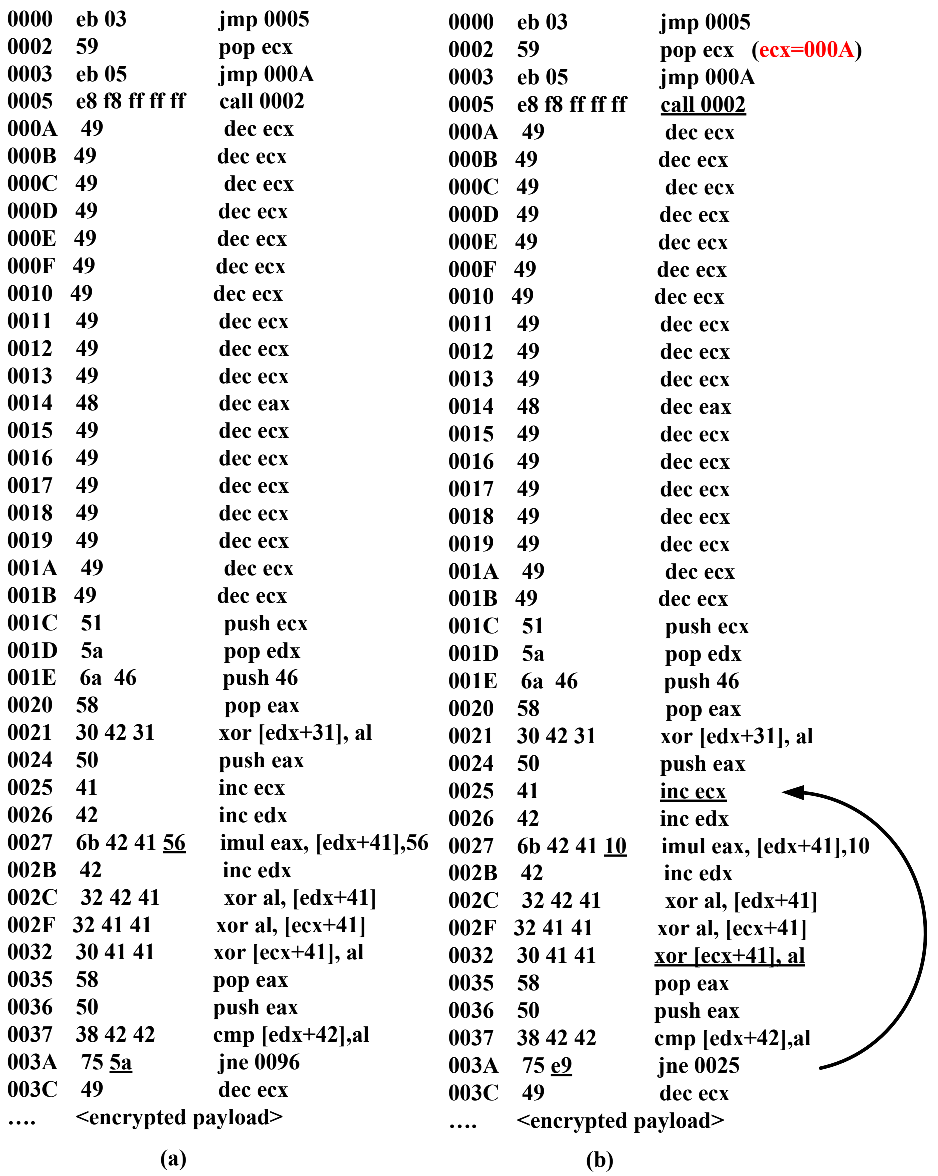
A.1 Disassembly of the Self-Modifying Decryption Routine for `Alpha2` Encoder

In this section, the disassembly result of the self-modifying decryption routine for `Alpha2` encoder is provided. The underlined instructions in figure A.1 (b) are the seeding instruction of the *GetPC* code, the memory-writing instruction for decrypting the encoded payload and the instruction for updating the address of encoded bytes. The underlined bytes in (a) and (b) highlight the contrast between modified instructions before and after execution.

A.2 Binary Code Analysis

In this section, the relevant background in the present thesis work is briefly introduced in the order of *instruction format*, *instruction operands and types*, and *static program analysis*.

Instruction Format Many binary code analysis techniques involve instruction disassembly from its binary representation. Figure A.2 shows the general instruction format for all Intel Architecture instruction encodings [2]. "Instructions consist of optional instruction *prefixes* (in any order), one or two primary *opcode* bytes, an addressing-form specifier (if required) consisting of the *ModR/M* byte and sometimes the *SIB* (Scale-Index-Base) byte, a *displacement* (if required), and an *immediate data* field (if required). [2]" The length of an instruction depends on its opcode as well as the directives indicated by its prefixes,



```
0000   eb 03            jmp 0005
0002   59               pop ecx
0003   eb 05            jmp 000A
0005   e8 f8 ff ff ff   call 0002
000A   49               dec ecx
000B   49               dec ecx
000C   49               dec ecx
000D   49               dec ecx
000E   49               dec ecx
000F   49               dec ecx
0010   49               dec ecx
0011   49               dec ecx
0012   49               dec ecx
0013   49               dec ecx
0014   48               dec eax
0015   49               dec ecx
0016   49               dec ecx
0017   49               dec ecx
0018   49               dec ecx
0019   49               dec ecx
001A   49               dec ecx
001B   49               dec ecx
001C   51               push ecx
001D   5a               pop edx
001E   6a 46            push 46
0020   58               pop eax
0021   30 42 31         xor [edx+31], al
0024   50               push eax
0025   41               inc ecx
0026   42               inc edx
0027   6b 42 41 56      imul eax, [edx+41],56
002B   42               inc edx
002C   32 42 41         xor al, [edx+41]
002F   32 41 41         xor al, [ecx+41]
0032   30 41 41         xor [ecx+41], al
0035   58               pop eax
0036   50               push eax
0037   38 42 42         cmp [edx+42],al
003A   75 5a            jne 0096
003C   49               dec ecx
....   <encrypted payload>
```

(a)

```
0000   eb 03            jmp 0005
0002   59               pop ecx   (ecx=000A)
0003   eb 05            jmp 000A
0005   e8 f8 ff ff ff   call 0002
000A   49               dec ecx
000B   49               dec ecx
000C   49               dec ecx
000D   49               dec ecx
000E   49               dec ecx
000F   49               dec ecx
0010   49               dec ecx
0011   49               dec ecx
0012   49               dec ecx
0013   49               dec ecx
0014   48               dec eax
0015   49               dec ecx
0016   49               dec ecx
0017   49               dec ecx
0018   49               dec ecx
0019   49               dec ecx
001A   49               dec ecx
001B   49               dec ecx
001C   51               push ecx
001D   5a               pop edx
001E   6a 46            push 46
0020   58               pop eax
0021   30 42 31         xor [edx+31], al
0024   50               push eax
0025   41               inc ecx
0026   42               inc edx
0027   6b 42 41 10      imul eax, [edx+41],10
002B   42               inc edx
002C   32 42 41         xor al, [edx+41]
002F   32 41 41         xor al, [ecx+41]
0032   30 41 41         xor [ecx+41], al
0035   58               pop eax
0036   50               push eax
0037   38 42 42         cmp [edx+42],al
003A   75 e9            jne 0025
003C   49               dec ecx
....   <encrypted payload>
```

(b)

Figure A.1: Disassembly of self-modifying decryption routine for Alpha2 encoder. a) Before execution b) After execution

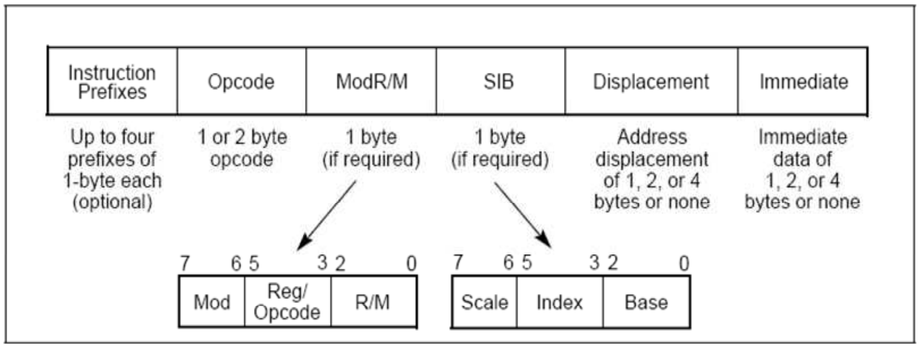


Figure A.2: Intel Architecture Instruction Format [2].

ModR/M and *SIB* bytes when applicable. The following description is based on the instruction manual [2]. Primary opcode can contain some smaller encoding fields which define the direction of operation, the size of displacement, the register encoding, condition codes, or sign extension. An additional 3-bit opcode field is sometimes included in the *ModR/M* byte which specifies how an operand in memory that is referenced by the instruction is addressed and contains 3 fields. The *Mod* field along with the *R/M* field forms 32 possible values, 8 general purpose registers and 24 addressing modes. The *R/M* field specifies a register as an operand or an addressing mode. The *Reg/Opcode* field specifies either a register as an operand or three additional bits of opcode, depending on the primary code. Sometimes a second addressing byte, the *SIB* byte is used to fully specify the addressing form and is required through certain encodings of the *ModR/M* byte. For more detail on the instruction format, types, and semantics, please refer to [2].

Instruction operands and types An operand of an instruction can be a register or a memory location specified by one or several combination of registers or an immediate data. Not all CPU registers can be explicitly referenced as operands in the instruction encoding. The *Mod* and *R/M* fields of the *ModR/M* byte can specify the general-purpose registers, the MMX^{TX} technology register $MM0$, or $SIMD$ floating-point register $XMM0$. The common Intel Architecture 32-bit registers are listed as follows. The instruction pointer register can not be directly referenced by instruction encoding.

- General Purpose Registers : `EAX (AH,AL)`, `EBX (BH,BL)`, `ECX (CH,CL)`, `EDX (DH,DL)`
- Pointer and Index Registers : `ESP, EBP, ESI, EDI`
- Segment Registers : `CS, DS, ES, SS, FS, GS`

- Instruction Pointer : `IP`
- Flag Register : `of, df, if, tf, sf, zf` etc.

According to [121], all 8086 Intel instructions are divided into six categories according to their function:

- Data Transfer Instructions : `mov, pop, push, xchg, in, out, lds, lea, lahf, popf, pushf`, etc.
- Arithmetic Instructions : `aaa, add, inc, cmp, dec, neg, sbb, sub, imul, mul, div, idiv, cbw`, etc.
- Logic Instructions : `add, not, or, test, xor, rcl, rcr, sar, shl, shr`, etc.
- Flow-control Instructions: `call, jmp, ret, retn, retf, int, into, loop, loope, loopne` and `jcc` (a set of condition jump instructions) , etc.
- Processor Control Instructions: `clc, sti, stc, esc, hlt, lock, wait, nop`
- String Instructions : lods, movs, stos, cmps, cmpsw, scas, rep, repe, etc.

In our implementation, we add two more categories. The first additional category contains the floating-point instructions or rarely used instructions such as `mmx` instructions and zero effect instructions. A zero effect instruction means that the instruction has no significant effect on the control flow or data flow of the program. This type of instruction includes `nop` and `cmp` etc. Someone would argue that a `cmp` instruction will affect a program's control flow since the condition evaluated by the `cmp` instruction will decide the branch taken by a control transfer instruction. In our analysis, we simplify the processing of branch condition evaluation and always generate two branches for the conditional control transfer instructions `jcc`. The second additional category contains a special set of instructions `xchg`, `cmpxchg` which is a conditional exchange instruction, and `xadd` which is an exchange and add instruction.

Static program analysis. We reference the concepts of static program analysis from an advanced compiler textbook [96].

- Control Flow Analysis: A control flow graph is normally constructed to achieve a global understanding of how program transfers control within procedures. A control flow graph consists of basic blocks of code of the program and edges connecting these basic blocks. An edge stands for a control transfer between the two connected basic blocks.

- Basic Block. A basic block is a sequence of code with only one entrance at the beginning and only one exit at the end.

- Data Flow Analysis: A data flow analysis of a program is a characterization of how it manipulates the data.

- Reaching Definition: "A definition is an assignment of some value to a variable. A particular definition of a variable is said to reach a given point in a procedure if there is an execution path from the definition to that point such that the variable may have, at that point, the value assigned by the definition. [96]" A definition is *killed* at a particular point if the variable is redefined at that point.

- Dependence: "A dependence between two statements in a program is a relation that constrains their execution order. A control dependence is a constraint that arise from the control flow of the program. A data dependence is a constraint that arises from the flow of data between statements. [96]" There are four kinds of data dependence. Figure A.3 shows an example reflecting these four kinds dependence. First, a *flow dependence* $\langle S1, S2\rangle$, is that the former ($S1$) sets a value (eax) that the latter ($S2$) uses. Second, an *anti-dependence* $\langle S2, S3\rangle$, is that the former ($S2$) uses some variable (eax)'s value that the latter ($S3$) sets. Third, an *output dependence* $\langle S3, S4\rangle$, is when both statements ($S3, S4$) set the value of some variable (eax). The last is *input dependence* $\langle S2, S5\rangle$, in which both statements ($S2, S5$) read the value of some variable (eax). "An input dependence does not constrain the execution order of the two statements. [96]"

- Copy Propagation: It "is a transformation that, given an assignment `x` ← `y` for some variables `x` and `y`, replaces later uses of `x` with uses of `y`, as long as intervening instructions have not changed the value of either `x` or `y`. [96]"

```
-------------------------
S1        pop    eax
S2        call   [eax]
S3        add    eax, 3
S4        xor    eax, eax
S5        mov    ecx, eax
-------------------------
```

Figure A.3: Example of control and data dependence in assembly code

A.3 Code Obfuscation

Acknowledgement. The following parts are from the spring $CSC591R$ class term paper which was a joint work with Young June Pyun.

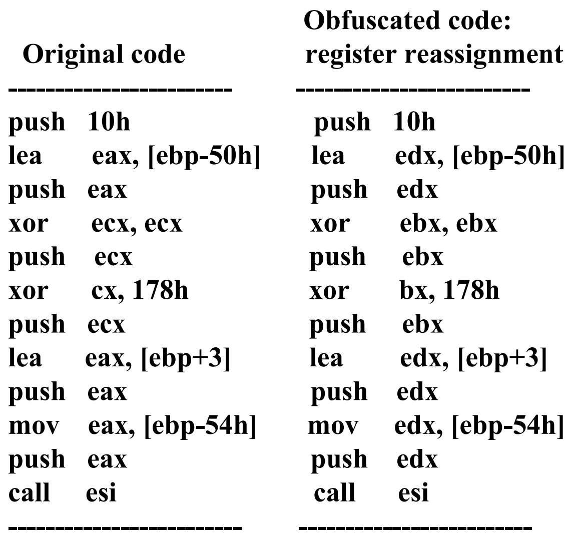
Code obfuscation is a transformation that does not change the behaviors of the original program and hence retains the core functionalities. It changes the existing code or generates new behaviors without affecting the overall result of the original program [122]. Code obfuscation is mainly used to generate metamorphic versions of malicious code, such as worms and viruses that are intended to subvert detection systems using signatures [26, 56]. It can also be used to protect software content from malicious reverse engineering [43]. Although code obfuscation has been proven to be computationally bounded and thus cannot completely hide the malicious behavior [123], some techniques are potentially plausible due to their simplicity and efficacy [56]. Some common code obfuscation techniques are listed below and the corresponding examples are appended in the end.

Register Reassignment. Code obfuscation by means of register reassignment replaces usage of one register with another within a specific live scope. It does nothing but exchange register names and will not affect program behavior. There is no real obfuscatory value gained other then avoiding naive signature matching detection systems.

Dead-Code/Junk-Code Insertion. Dead-code or junk-code insertion is another code obfuscation technique that adds code to a program without affecting the original functionality. This includes `nop` instruction, dead-code, and junk-code insertion. Dead-codes are a sequence of instructions that cancel out, whereas junk-codes are irrelevant instructions that do not affect the intended result of the program. The objective of this obfuscation is to disguise sophisticated signature matching detection systems.

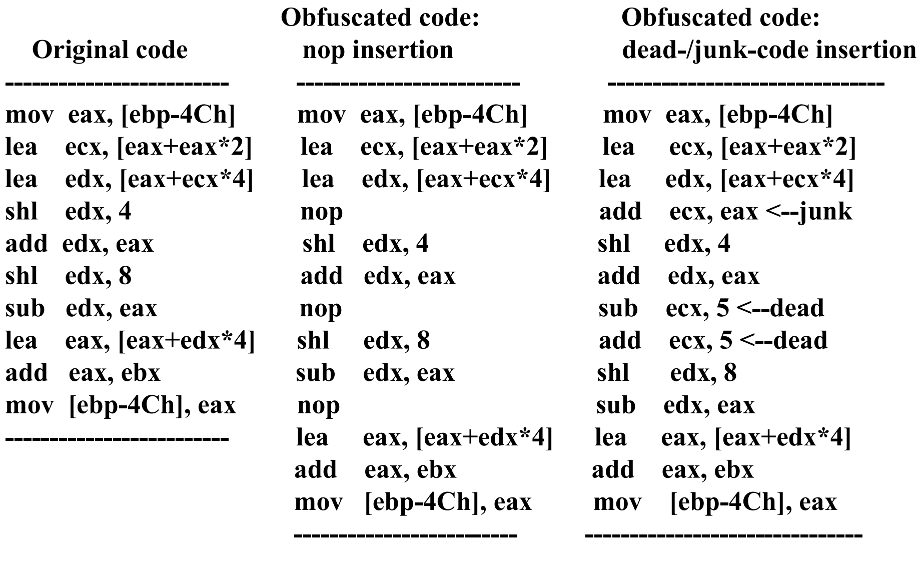
Code Transposition. Code transposition permutes the order of the instructions in such a way that maintains the original program flow. It includes random reordering of instructions with unconditional branches and swapping of two sequences of instructions that are independent. The objective of this method is to transform the original program image into an unexpected instruction order intended to fool the detection systems relying on fixed patterns of instruction orders.

Instruction Substitution. Instruction substitution replaces an instruction sequence with one or more semantically equivalent instructions. This technique usually requires a pre-generated dictionary of equivalent instruction sequences for implementation.



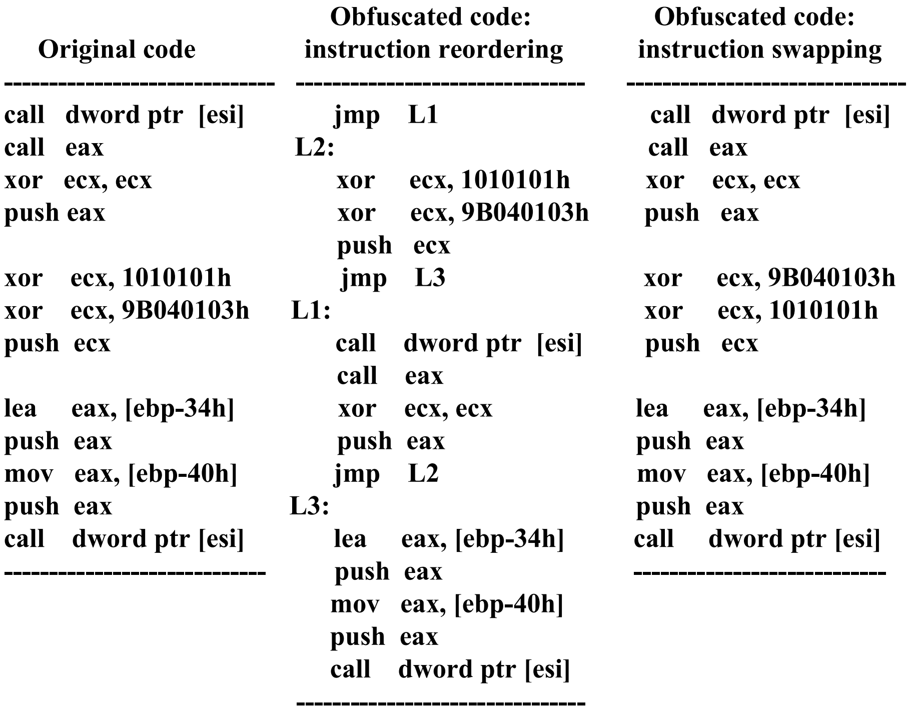
```
                            Obfuscated code:
  Original code             register reassignment
-----------------------    -----------------------
push   10h                  push   10h
lea     eax, [ebp-50h]      lea     edx, [ebp-50h]
push   eax                  push   edx
xor     ecx, ecx            xor     ebx, ebx
push    ecx                 push    ebx
xor     cx, 178h            xor     bx, 178h
push   ecx                  push    ebx
lea     eax, [ebp+3]        lea     edx, [ebp+3]
push   eax                  push   edx
mov    eax, [ebp-54h]       mov    edx, [ebp-54h]
push   eax                  push   edx
call    esi                 call    esi
------------------------    ------------------------
```

Figure A.4: Example of code obfuscation with register reassignment.



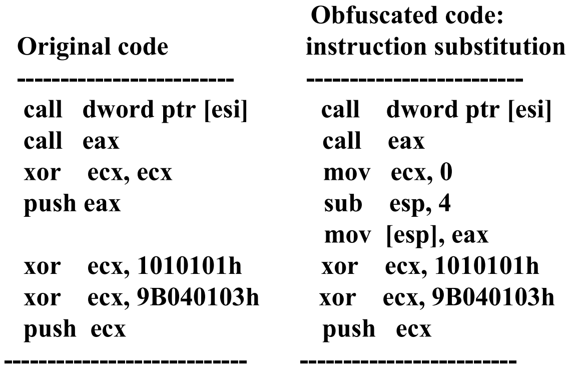
```
                          Obfuscated code:            Obfuscated code:
  Original code             nop insertion             dead-/junk-code insertion
-----------------------   -----------------------    ------------------------------
mov  eax, [ebp-4Ch]       mov  eax, [ebp-4Ch]        mov  eax, [ebp-4Ch]
lea    ecx, [eax+eax*2]   lea    ecx, [eax+eax*2]    lea    ecx, [eax+eax*2]
lea    edx, [eax+ecx*4]   lea    edx, [eax+ecx*4]    lea    edx, [eax+ecx*4]
shl    edx, 4             nop                        add    ecx, eax <--junk
add   edx, eax            shl    edx, 4              shl    edx, 4
shl    edx, 8             add    edx, eax            add    edx, eax
sub   edx, eax            nop                        sub    ecx, 5 <--dead
lea    eax, [eax+edx*4]   shl    edx, 8              add    ecx, 5 <--dead
add   eax, ebx            sub    edx, eax            shl    edx, 8
mov  [ebp-4Ch], eax       nop                        sub    edx, eax
-----------------------   lea    eax, [eax+edx*4]    lea    eax, [eax+edx*4]
                          add    eax, ebx            add    eax, ebx
                          mov   [ebp-4Ch], eax       mov    [ebp-4Ch], eax
                          -----------------------    ------------------------------
```

Figure A.5: Example of code obfuscation with nop, dead-code, and junk-code insertion.



```
                            Obfuscated code:            Obfuscated code:
    Original code          instruction reordering      instruction swapping
------------------------  -------------------------   -------------------------
call   dword ptr  [esi]        jmp    L1                call   dword ptr  [esi]
call   eax                L2:                           call   eax
xor   ecx, ecx                 xor    ecx, 1010101h     xor    ecx, ecx
push eax                       xor    ecx, 9B040103h    push   eax
                               push   ecx
xor    ecx, 1010101h           jmp    L3                xor     ecx, 9B040103h
xor    ecx, 9B040103h     L1:                           xor     ecx, 1010101h
push  ecx                      call   dword ptr  [esi]  push   ecx
                               call   eax
lea     eax, [ebp-34h]         xor    ecx, ecx          lea     eax, [ebp-34h]
push  eax                      push  eax                push  eax
mov   eax, [ebp-40h]           jmp    L2                mov   eax, [ebp-40h]
push  eax                 L3:                           push  eax
call    dword ptr [esi]        lea     eax, [ebp-34h]   call    dword ptr [esi]
------------------------       push  eax                ------------------------
                               mov   eax, [ebp-40h]
                               push  eax
                               call    dword ptr [esi]
                          -------------------------
```

Figure A.6: Example of code obfuscation with instruction reordering and instruction swapping.



```
                           Obfuscated code:
Original code             instruction substitution
---------------------     ---------------------
 call   dword ptr [esi]     call    dword ptr [esi]
 call   eax                 call    eax
 xor    ecx, ecx            mov   ecx, 0
 push eax                   sub    esp, 4
                            mov  [esp], eax
 xor    ecx, 1010101h       xor    ecx, 1010101h
 xor    ecx, 9B040103h      xor   ecx, 9B040103h
 push  ecx                  push   ecx
----------------------    ---------------------
```

Figure A.7: Example of code obfuscation with instruction substitution.