2009

Detecting Malicious Software By Dynamicexecution

Jianyong Dai
University of Central Florida

Detecting Malicious Software by Dynamic Execution

by

JIANYONG DAI
M.S., University of Central Florida, 2008
B.S., Shanghai Jiaotong University, 1998

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florrida
Orlando, Florida

Summer Term
2009

Major Professor: Ratan K. Guha

ABSTRACT

Traditional way to detect malicious software is based on signature matching. However, signature matching only detects known malicious software. In order to detect unknown malicious software, it is necessary to analyze the software for its impact on the system when the software is executed. In one approach, the software code can be statically analyzed for any malicious patterns. Another approach is to execute the program and determine the nature of the program dynamically. Since the execution of malicious code may have negative impact on the system, the code must be executed in a controlled environment. For that purpose, we have developed a sandbox to protect the system. Potential malicious behavior is intercepted by hooking Win32 system calls.

Using the developed sandbox, we detect unknown virus using dynamic instruction sequences mining techniques. By collecting runtime instruction sequences in basic blocks, we extract instruction sequence patterns based on instruction associations. We build classification models with these patterns. By applying this classification model, we predict the nature of an unknown program. We compare our approach with several other approaches such as simple heuristics, NGram and static instruction sequences.

We have also developed a method to identify a family of malicious software utilizing the system call trace. We construct a structural system call diagram from captured dynamic system call traces. We generate smart system call signature using profile hidden Markov model (PHMM) based on modularized system call block. Smart system call signature weakly identifies a family of malicious software.

ACKNOWLEDGMENTS

A PhD degree is not just a personal achievement. It is a contribution of a group of people.

First, I would like to thank my thesis committee members. In particular, Dr Ratan Guha who advise me to finish my major research accomplishment during my PhD study. Dr Joohan Lee, who advise me for my initial part research and build a solid ground for my continuous work. Dr Cliff Zou, who arouse my interest to work on security. Dr Morgan Wang, who give me lots of suggestions on data mining.

I would also thank for Dr Ning Zhang, without your support, I can hardly finish my PhD study. Dr Jianlin Cheng, who consistently encourages me with my research.

I would like to give thanks to my parents. Without your wholehearted support, I may not be able to make my decision to pursue a PhD degree.

Last but not the least, I would like to thank all my friends in the University of Central Florida who are around me everyday. I can only name a few here, Ping Wang, Shipeng Qiu, Yu Ao, Danzhou Liu, Ruifeng Xu, Qing Chang, etc. You keep me happy and give me the courage and help to get through all the difficulties.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Motivation

Malicious software is becoming a major threat to the computer world. The general availability of the malicious software programming skill and malicious code authoring tools makes it easier to build new malicious codes. In 2008, the number of new malicious code signatures increased by 265 percent over 2007; over 60 percent of all currently detected malicious code threats were detected in 2008 [1]. Moreover, the advent of more sophisticated virus writing techniques such as polymorphism [2] and metamorphism [3] makes it even harder to detect a virus.

The prevailing technique in the anti-virus industry is based on signature matching. The detection mechanism searches for a signature pattern that identifies a particular virus or strain of viruses. Though accurate in detecting known viruses, the technique falls short for detecting new or unknown viruses for which no identifying pattern is present. Whenever a new virus comes into the wild, virus experts extract identifying byte sequences of that virus either manually or automatically [4], then deliver the fingerprint of the new virus through an automatic update process. The end user will finally get the fingerprint and be able to scan for the new viruses.

However, zero-day attacks are not uncommon these days [5]. These zero-day viruses propagate fast and cause catastrophic damage to the computers before the new identifying fingerprint is distributed.

Furthermore, polymorphic and metamorphic viruses change their appearance every time they propagate. The signature generated for one virus copy may not capture other virus copies.

Several approaches have been proposed to detect unknown virus without signatures. These approaches can be further divided into two categories: static approaches and dynamic approaches. Static approaches check executable binary or assembly code derived from the executable without executing it. Detecting virus from binary code is semantic agnostic and may not capture the key component of virus code. Static approaches based on assembly code seems to be promising, however, deriving assembly code from an executable itself is a hard problem. In our experiment, we find that approximately 90% of virus binary code cannot be fully disassembled by state of art disassembler. Dynamic approaches are to actually run the executables and capture the runtime behavior.

1.2 Dynamic Instruction Sequences Approach Overview

Most existing dynamic approaches are based on system calls made by the unknown executable at runtime. The idea behind is that viral behavior of a malicious code is revealed by system calls. However, some malicious code will not reveal itself by making such system calls in every invocation of the virus code. On the other hand, some malicious behaviors such as self-modifying are not revealed through system calls. Based on these observations, we propose to use dynamic instruction sequences instead of system calls to detect virus dynamically.

One approach to detect unknown virus automatically is to use heuristic rules based on expert knowledge [6]. Although this approach is widely adopted by commercial anti-virus software, it is easier to be evaded by the virus writers once the rules are known to them. The other approach is data mining. Data mining refers to the process to prepare, transform, and discover useful knowledge from data automatically [7]. In our context, it is a classification problem to determine whether a program can be classified into either malicious or benign.

The key problem for this classification problem is how to extract features from captured runtime instruction sequences. We use the information how instructions group together to capture the nature of malicious behavior. To this end, we devise a notion called instruction association.

In the first step, we organize instructions into logic assembly. Logic assembly is a reconstructed program assembly using available runtime instruction sequences. It may have incomplete code coverage, but logic assembly will keep the structure of the executable code. In addition, the process of logic assembly construction handles the self-modifying code in a normal way.

The second step is to extract frequent instruction groups inside basic blocks within logic assembly. We call these instruction groups "instruction associations". We use three variations of instruction associations. The first is the instruction association that observes which instructions appear together in a block but does not consider the order. Second, we consider the order of the instructions in a block but not necessarily consecutive. Third, we consider the exact consecutive order of instructions in a block.

We use the relative frequency of instruction association as features of our dataset. We then build classification models based on the dataset.

While accuracy is the main focus for virus detection, efficiency is another concern. No matter how accurate the detection mechanism is, if it takes long time to determine if an executable is a virus or not, it is not useful in practice. We log the first several thousand dynamic instructions of an unknown program. We believe the initial part of a virus and a benign code is quite different. Our analysis shows that compared to system calls, our approach takes less time to collect enough data for the classification model, and the subsequent calculation time is affordable.

1.3 Structural System Call Diagram Approach Overview

Most existing dynamic malicious software detection approaches are using system call traces. System call trace contains important information about the behavior of a program, however, system call trace is chronological logs with repetitions. It is hard and time consuming to read.

Structural system call diagram is an approach to address this problem. We construct a control flow graph like structure to visualize the execution flow of the system call trace. With the help of structural system call diagram, it is much easier to analyze the behavior of the program.

Structural system call diagram also provides a natural boundary of system calls. Using this system call boundary, we are able to identify a distinctive system call signature to capture a family of malicious software. We can derive the system call signature either manually or automatically. We do not require exact match, we allow insertion, deletion and mutation of any system calls within this system call signature.

1.4 Dissertation Organization

In the rest of dissertation, we first provide background of this research in chapter 2, then describe dynamic execution environment in chapter 3, present dynamic instruction sequence approach in chapter 4, structural system call diagram approach in chapter 5, and present our conclusion in chapter 6.

CHAPTER 2: BACKGROUND

2.1 Malicious Software

Malicious software is a software written purposefully to perform unwanted actions to the authorized computer users. The exact action malicious software performing evolves over time. In the early years, the unwanted actions could be some simple pranks, such as displaying message "catch me if you can!". Later on, many malicious softwares are designed to destroy end user's computer, such as formatting the hard drive or deleting crucial system files. Current trend for malicious code is to perform criminal actions such as steal user's private information or use one's computer as a zombie to attack other computers. Some software are written in good intention in mind, such as patch the computer against certain system vulnerability. But as long as it is not the intention of the end user, it is generally considered as malicious software as well.

Malicious software is a collective notion. It can be further divided into several categories. Different categories may overlap each other.

Computer virus is the first kind of malicious software ever written. Computer virus is characterized by the ability to infect other executable files. Computer virus do not propagate automatically. Typically virus code propagates when user execute a malicious code inadvertently. Nowadays, these malicious codes typically come with email attachments.

Malicious software can also propagate totally unnoticed by the user by exploiting system vulnerabilities. We call this type of malicious software worm. Network service vulnerabilities are the most common threats. "Code Red" [8] exploits a vulnerability in IIS web server. "Klez" worm [9] propagates via email. Different from email virus which requires user interaction, "Klez" worm exploits a vulnerability [10] in Outlook Express and will execute automatically without explicitly opening the email attachment by the user. "Nimaya" [11] is able to propagate by simply inserting an infected flash memory into the victim computer without any other additional actions by the user.

Trojan horse [6] is memory resident malicious software. Once infected, the victim computer opens up one or several control ports for intruders to control certain part of the computer behavior. Trojan horse can either passively listens to the open ports and waits for the instructions from the intruder (backdoors), or actively collects information on the victim's computer and sends it out to the intruders. Keylogger is a special type of trojan horse that records user's key strokes stealthily and sends them back to the intruders. Key strokes may contain valuable information such as password and credit card number.

Macro virus [6] is characterized by the type of code it utilizes. Instead of binary code, Macro virus is coded using Macro language provided by application software such as Microsoft Word. A big install base of these application software make Macro virus be able to run on a thousands of millions computers.

Spyware [6] is installed together with legal software without user's notice. Spyware do not propagate automatically and do not infect other system files. They abuse the user consent on the legal software they want to install. Once installed, Spyware collects user's privacy information such as online behavior and sends back to the designated agents.

Rootkit [12] is characterized by its extreme stealth. Rootkit modifies part of operating system files to conceal its files, processes and registry entries. Conventional techniques cannot detect the presence of the rootkit because of invisibility of the trace of rootkit. The focus of rootkit research is to reveal the presence of a rootkit.

Botnet [13] is characterized by infecting a large number of computers into zombies to form a malicious network. The term botnet refers to the whole malicious network. Each victim computer is called a bot. Botnet is frequently used to send spam emails [14] and launch DDos attack. Usually the number of bots consisting a botnet is around several hundred to several thousand [15]. A smaller number of bots is harder to detect and still powerful to be useful to the botnet owner. The focus of botnet research is to turn down the existing botnet rather than defend against the bot code to be installed on end user computers. The installation process of bots is similar to those of worms.

The classification of malicious software mentioned above is not mutually exclusive. Some malicious software can sit on two or more categories. Bots which forms a botnet usually propagates by exploiting a system vulnerability which makes them also computer worms. Some advanced worms conceal themselves using rootkit techniques, which makes them both worm and rootkit.

2.2 Conventional Malicious Software Detection Techniques

Conventional malicious software detection relies on signature matching [6]. The detection mechanism searches for a signature pattern that identifies a particular virus or a strain of viruses.

Simple signature figure 2.1 is a byte sequence requiring exact match against virus code. Short signature results a high false positive rate. False positive rate is the proportion of benign executables that were erroneously reported as being malicious. A high false positive rate is very annoying to the user. However, long signature can be very restrictive that can only capture one particular malicious code not even its closest relatives.

Wildcard signature figure 2.1 is more flexible than simple signature in that they are able to catch a family of malicious code. Wildcard signature contains special "wildcard" character that capture any character. The mechanism is similar to Unix filename wildcard. With wildcard signature, we are able to capture invariant part of a family of malicious code. However, wildcard signature is much slower than the simple signature.

Signature of malicious software can be derived either manually or automatically. When a new malicious software comes into wide, computer security expert analyzes the malicious code and find out a unique feature of it. With the domain knowledge, security expert can find near optimal signature of the malicious code. However, this approach is costly

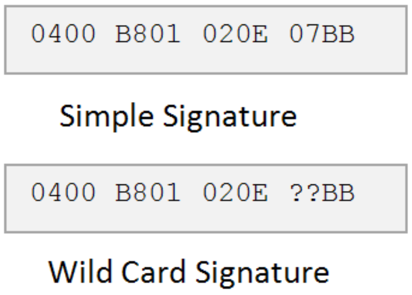


Figure 2.1: Virus Signature

and slow. When the outbreak of the malicious software becomes faster and faster, security experts are becoming overwhelmingly busy. An alternative approach is to extract signature automatically [4]. This approach relies on maintaining a benign signature database. For every malicious software, a random signature is generated and then compared with benign signature database to ensure that no benign program contains that byte sequence.

Signature database for malicious software is a critical asset for an anti-virus company. It is usually a well guarded secret and not available to general public. However, some open source anti-virus software do open its signature database. ClamAV [16] is one of the most popular open source anti-virus software, and its signature database is available to public. At the time this thesis is written, there are 513,978 malicious software signatures in its database and this number is keep increasing. ClamAV signature support wildcard character “?” and “*”, the meaning of both characters are similar to those in Unix wildcard. The length of the signature varies and most signatures are longer than 100 bytes to avoid false positive. All signatures are intended to capture only one malicious code rather than a family. Table 2.1 illustrates some of the signatures inside the ClamAV signature database.

A constant update of the signature database is essential to make it possible to capture latest malicious code. There is certainly a time gap between the outbreak of a malicious code and the availability of its signature. This gap often creates problem because by the

Table 2.1: ClamAV Database Sample

Trojan.Pakes-17	363e0380085a21b6b3e2e5f9f945a6aba3d92323820d488a220622589ceab79fbd9209c564b5b3230082d8c69d02806400468503b78122222621f8bcb0f9dd510000f8eeaa173cec93ee9ce8740980ca1e5d0e44ba269134375c244108497583c4671c586540e0f2d1e441120ab53354e7667f28c0e45075a99ade135960a47b
Trojan.Win32.Rootkit	d3001110010e68026e81887f2e2bf6ff7984881de4b73371205cb518d46000857e6bb5da517a0ea53b38df5f5a800e06852f1b0fe627d85452881fc3034dc2180b961bbfc7231006176462c0139624f9b6a6e21fa188903100e53c9c60e51c6edc474b68261c0c6f0d84e061095809eaeb1907c44861e951a8bb340041e39196
JS.Feebs.Y	6c616e67756167653d{-1}6a617661736372697074{3-6}3d22*3d756e65736361706528{-4}293b{-25}3b6576616c28{-4}293b{-4}3d2228??223b??28{-4}293b3c2f736372
Trojan.Downloader.Small-1170	3005723d25640c0826092ea489c3034d414a914845d544c2275680525920423a4947265ad49c4d433a294c4f950318776a74974a036720c36d6e7602ab5cfb7273aeeb57
Trojan.Downloader.Small-1171	6b68626073686e6d725b4b687273000000003b2b3b464f42434d46453b31000000005a65726174756c2e20427261737361692073
Email.Phishing.Pay-4	496620796f7520646f6e??742061677265652077697468207468697320{-1}6d61696c20616e6420696620796f75206e656564206173736973746616e63652077697468206f7572206163{-28}617374652074686520666616c6c6f77696e6720{3-4}20696e20796f757220(69|49)6e7465
HTML.Phishing.Bank-379	6c6f67206f666620616674657220757369696e6720796f7572206f6e6c696e65206163636f756e743c2f6469763e3c6469763e{-7}636c69636b2074686520666f6c6c6f77696e67206c696e6b2c20746f20766572696679207796f7572206163636f756e74206163746976697479a3c2f64

time a massive outbreak could already happen. For every unknown malicious code, there will be such a gap. Even for a variation of known malicious code, existing signature fails.

Creating a variation of known malicious code is easy. Manually, malicious software writer can change some part of the source code to create a new virus code. As long as the captured signature of old malicious code changes, the new variation will penetrate the protection of existing anti-virus software. Although signature database of most anti-virus software is not open, virus writer can test this new binary code against existing anti-virus software to make sure the new code will not be captured by existing malicious software database. If none of existing anti-virus software captures it, the virus writer can launch a zero-day attack again easily.

Another way to create a variation of an existing malicious software is through polymorphism [2] and metamorphism [3]. Polymorphism is a self-encrypted code with a unique encryption key embed for each malicious software copy. The initial part of the execution decrypt the encrypted code using the embed encryption key. Each time malicious code infect other file, it changes its encryption key to make a complete different copy. Only the initial decryption code is the same for all those copies, and these invariant codes are the key to detect a polymorphic malicious code. Metamorphism on the other hand, do not encrypt the code. It replaces the original codes with equivalent but transformed codes. The techniques for metamorphic code include code swap, redundant code injection, etc. This impose a challenge to the malicious software writer. They need to reorganize the malicious code as they propagate. However, it is a greater challenge for security expert. No part of the malicious code is reliable to detect all variations of the metamorphic code. There are several efforts aim to deal with metamorphism [17,18,19,20]. However, no general solution has been invented yet.

All above mentioned approaches: constant code rewriting, polymorphism and metamorphism impose a great challenge for the signature based detection mechanism. The effort to derive a new signature is much more than the effort to create a new malicious code variation. To cope with this, a different mechanism to detect unknown malicious code is highly desired.

One promising research area is smarter virus signature. In addition to wildcard signatures, Brumley et al [21] experiment Turing machine signatures, Symbolic constraint signatures and regular expression signatures.

In [22, 23], the authors use binary as their features and derive a profile hidden Markov model from a family of computer viruses. They use the profile hidden Markov model as a signature to score an unknown executable to determine the nature of the unknown executable.

Smarter signature relax the constraint of the virus signature. However, the basic problem of signature still exists. If the new variation is very different or the virus comes from another computer virus family, smarter virus signatures fail as well.

2.3 Detecting Unknown malicious software

Although the problem of determining whether an unknown software is malicious or not has been proven to be generally undecidable [24], detecting malicious software with an acceptable detecting rate is still possible. A number of approaches have been proposed to detect unknown malicious software. Here we discuss the approaches to detect Win32 computer viruses only.

2.3.1 Heuristics

Unlike conventional programming, virus code need to inject its code into existing program without the support of a compiler. Some unique problem need to be solved by the virus writer.

(i) Consistency of Win32 executable head. For a conventional program, compiler and linker will generate correct Win32 executable head structure. Every Win32 executable consists of a head followed by several sections of code and data. The head contains meta data for the executable and each individual section. It also contains import table and export table for the executable. Pietrek describe Win32 head structure in detail [25,26]. A virus, however, injects its code into the victim executable, the consistency of the victim executable head is compromised. Virus writer should adjust the Win32 head of the victim executable carefully to correct those inconsistency.

(ii) Placement of code. It is hard to move existing code inside victim executable around. So the easiest and most popular strategy is to add the virus code in the end. That is, append to the last section of the code or create a new section in the end.

(iii) Self relocation. A large amount of instructions take absolute values of virtual memory address as their operands. However, virtual memory address of a program is determined at runtime by the program loader. The relocation table within executable file will point to those operands and the value of these operands will be adjusted when loading. Injected virus code lacks the support of relocation table and virtual addresses inside virus code cannot be adjusted at load time. Virus writers have two choices to deal with this problem. Either do not use absolute addressing at all, or manipulate the relocation table when injecting the virus code. The former approach is much easier and popular. The idea is to use some special instruction sequences to get PC address of the current instruction. For the afterward access to the virtual memory address, use relative address instead of absolute value. Figure 2.2 illustrate a self relocation code snippet.

(iv) Locate Win32 API call. All destructive actions of a virus are performed through Win32 API calls. Win32 API calls are provided by system DLLs. A Win32 program can invoke a Win32 API call dynamically or statically. Static approach utilizes the DLL entries

```
        call next;                push address of “next” on stack
next:
        pop ebx;                  save address of “next” to ebp
        sub ebx, offset next;     calculate the “delta”, and save it
                             ;    to ebx

        lea lea eax,         ;    every time we refer a variable,
            [ebx + szHello]  ;    plus delta value
```

Figure 2.2: Self Relocation Code

information in the import table in PE head. However, there is no guarantee that the Win32 API the virus required is already in the PE head and it is hard to modify the import table. An easier solution is to use dynamic DLL call. However, two DLL invocations are required to perform an arbitrary dynamic dll call, “LoadLibrary” and “GetProcAddress”. These two API calls are inside “kernel32.dll”. The base address for module “kernel32.dll” differs even if the minor versions of the Windows operating system are different. Virus writers usually do a memory search to find “kernel32.dll”, and get the virtual address for these two system APIs.

To detect unknown Win32 viruses, simple heuristics is one frequently used approach by commercial anti-virus software. It is based on invariant parts of a computer virus. [6] describes many heuristics to detect Win32 viruses.

A well structured binary code is usually compact. So it is hard for a virus to insert its code between sections. Most viruses choose to insert their code into the last section or append a new section to the victim binary, both of which can be extended almost unlimitedly. Once they have appended the viral code, they also change the program entry point to the newly added viral code. The result is that the entry point points to the last section of the executable binary. It is totally legal and some compilers will generate executable binary share this character. However, most compilers put all executable codes into “.text” section, and “.text” section rarely located in the last section. The fact that entry point points to the last section by itself is not obvious enough to tell malicious code from benign. However, this is an important clue if combined with other observations.

Another strategy used by virus writers is to inject virus code into the holes between neighboring sections. Each hole is small, but by combining several holes, it could be large enough to fit all code of a virus. Virus code uses control flow transfer instructions to glue each small part together. It is more difficult to write this kind of virus code but it is a more stealthy way in that entry point do not need to be in the last section. A heuristic to detect such phenomena is to get the percentage of occupation for the last blocks of every section. If last blocks of all sections are very full, then this executable binary code is suspicious.

Each section has a section name. Most compiler will use some well defined section names such as ".text" or ".code" for instructions, ".data" for global variables, ".bss" for uninitialized variables, etc. Although users are able to override these default settings, it is not common for most of programs. That makes section name another heuristics to detect unknown viruses.

Size consistency is a heuristics based on the correctness of section size calculation. For a well-formed executable file, the size field in PE head should equal to the aggregate of the size all sections of the executable plus the size of the PE head. However, virus code may skip the size calculation and leaves an inconsistent PE head.

To solve the problem of self-relocation, virus frequently use "call next address" operation (See figure 2.2) in the beginning of virus body. This operation is meaningless in regular program. The hex representation for this operation is "E800000000". Existence of such bytes is another important heuristics in detecting Win32 virus.

Memory search for "PE" and "kernel32.dll" is suspicious because it is a mechanism frequently used by virus to locate "LoadLibrary" and "GetProcAddress" Win32 API. Legitimated programs do not need to do that because they can use these functions directly with the support of a compiler and linker.

However, heuristics is becoming less effective in detecting new generation of computer viruses. Once known to the virus writers, these simple heuristics can be easily avoided as follows.

(i) Consistency of executable is not a hard problem to solve. Virus writer should be more careful to adjust all fields in Win32 head.

(ii) Entry point obscuring (EPO) [27] technology makes code placement heuristics less effective. With EPO, virus code is still placed in the end of the executable, however, entry point is located in other location in the executable other than the virus code. A control flow transfer instruction will be placed near the entry point to divert the execution of the code to the virus code. With such an arrangement, entry point is not necessarily to be placed in the last section.

(iii) Self-relocation seems to be unavoidable in an executable. However, most executables do not need relocation and there is no relocation table inside executable. Relocation process is usually used by dynamic linked library. It is guaranteed that executable file module is the first module to be loaded into the process address space. So executable is always granted the preferred base address and there is no need to relocate. Without relocation, it is much easier for a virus to calculate the virtual address operands of an instruction. And because of this, self-relocation code becomes unnecessary.

(iv) API locating operation is unavoidable. However, there could be a huge variation for API locating code and capture these code is not easy. The conventional way is to check the string “PE” and “kernel32.dll” for which the virus code is searching. However, using simple string encryption hides the string and compromises this heuristics.

2.3.2 Research Domain

There are other approaches in the research domain which go beyond simple heuristics. We can divide them into static approaches and dynamic approaches.

Static approaches check executable binaries or assembly codes without actually executing the unknown program.

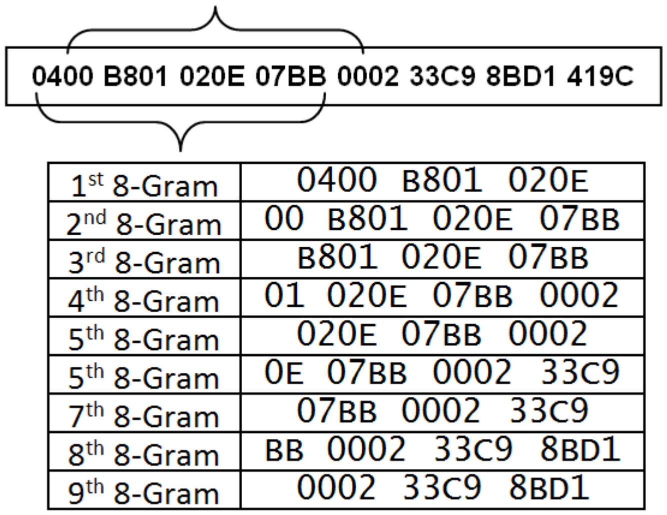
Weber et al [28] develop a Portable Executable Analysis Tool (PEAT) system that is basically a GUI tool to assist virus researchers to identify computer viruses. It first checks the consistency of PE header, and then presents key statistics such as instruction frequencies, call distances, together with byte-type view, ASCII view and other views to the end user. End users need their own experience to judge if the executable is a virus using the provided statistics on the given program.

The work of Arnold et al [29] uses binary trigram as their detecting criteria. They use neural network as their classifier and reported a good result in detecting boot sector viruses for a small sample size.

InSeon et al [30] also use binary sequences as features. However, they construct a self organizing map on top of these features. Self organizing map converts binary sequences into a two dimensional map. They claim that malicious viruses from the same virus family demonstrate the same characteristic in the resulting graph. But they do not give a quantitative way to differentiate a virus from benign code.

Schultz et al [31] use comprehensive features in their classifiers. They use three groups of features. The first group is the list of DLLs and DLL function calls used by the binary. The second group is string information acquired from GNU strings utility. The third group is a simple binary sequence feature. They conducted experimentation using numerous classifiers such as RIPPER, Naive Bayes and Multi-Naive Bayes.

In recent years, researchers start to explore the possibility to use N-Gram in detecting computer viruses [32, 33, 34]. Here N-Gram refers to consecutive binary sequences of fixed



1st 8-Gram	0400 B801 020E
2nd 8-Gram	00 B801 020E 07BB
3rd 8-Gram	B801 020E 07BB
4th 8-Gram	01 020E 07BB 0002
5th 8-Gram	020E 07BB 0002
5th 8-Gram	0E 07BB 0002 33C9
7th 8-Gram	07BB 0002 33C9
8th 8-Gram	BB 0002 33C9 8BD1
9th 8-Gram	0002 33C9 8BD1

Figure 2.3: Illustration of 8-Gram

size inside binary code. In binary executables, it stands for n consecutive binary sequences as shown in figure 2.3.

Kolter et al [34] extract all N-Gram from a training set and then perform a feature selection process based on information gain. Top 500 N-Gram features are selected. Then, they mark the presence of each N-Gram in the training dataset. These binary tabular data are used as the input data for numerous classifiers. They experimented with Instance-based Learner, TFIDF classifier, Naive Bayes, Support Vector Machines, Decision Trees and Boosted Classifiers. Instead of accuracy, they only reported AUC (Areas Under Curves). The best result is achieved by boosted J48 at AUC, 0.996.

Although above approaches show satisfactory results, these detection techniques are limited in that they do not distinguish the instructions from data and are blind to the structure of the program which carries important information to understand its behavior. We have repeated the experiment mentioned in [34] and we have found that the key contributors that lead to the classification are not from bytes which representing virus code, rather, they are from structural binary or string constants.

Table 2.2: Top 8-Gram

Benign	Virus
`0000000000000000`	`0000000000000000`
`0404040404040404`	`9090909090909090`
`2000200020002000`	`5858585858585858`
`0020002000200020`	`cccccccccccccccc`
`4040404040404040`	`2a2a2a2a2a2a2a2a`
`ffffffffffffffff`	`2121212121212121`
`cccccccccc8bff55`	`4142434441424344`
`cccccccc8bff558b`	`4441424344414243`
`cccccc8bff558bec`	`4344414243444142`
`2020202020202020`	`4243444142434441`

Table 2.2 lists top ten 8-Grams from both benign and malicious dataset. In the top benign 8-Grams, the first six are only structural bytes which have nothing to do with instructions. Seventh to ninth 8-Grams are partial instructions and partial structural bytes. Actually all three 8-Grams contain the same instruction sequences. This instruction sequences repeat three times with a slightly different window. The tenth 8-Gram contains only string information. Binary “20” represents space in ASCII.

In the top virus 8-Grams, second and third 8-Gram could be instructions. “90” is “nop”and “58” is “pop ax” in 80x86 instruction set. However, they are just “boring” instructions. The primary reason for virus writers to use these instructions is padding rather than doing real stuff. Fifth to tenth 8-Grams are string information.

We examine many top N-Grams, and we find that most top N-Grams are not instructions. They are either structural bytes or string information. Even if there are a few instructions, most of them are “boring” instructions, such as “nop”, rather than critical instructions. Most instruction N-Grams are actually the same instruction sequences repeated for many times with a slightly different window.

Since structural binary and string constants are not essential components to a virus, N-Gram detection mechanisms which are primarily based on these features can be evaded.

Ye et al [35] uses the imported functions inside import table of an executable as main criteria to detect viruses. They derive association rules by mining the combination of imported functions for both virus and benign software. The major problem of this approach is the ignorance of the virus codes which do not rely on import table entries. Many virus codes load dynamic linked libraries dynamically and do not put Win32 API entries into import table of the executable.

Another set of static approaches focus on higher level features which are constructed on top of assembly code. Christodorescu et al [36] use template matching against assembly code to detect known malicious behavior such as self-modification. In [37], the authors propose to use control graph extracted from assembly code and then use graph comparing algorithm to match it against known virus control graphs. Kinder et al [38] is also based on assembly code. The authors first identify temporal dependencies among different instructions. Then they generalize the dependencies into a formal specification. They present the specification for CopySelf and ExecOpened behavior. Using a model checking algorithm, they can check if a specific executable contains such temporal dependencies. Siddiqui et al [39] uses opcode N-Gram based on disassembly code. This paper put more focus on statistical manipulation.

These approaches seem to be promising. The problem is that disassembling executable code itself is a hard problem. The difficulty comes from two sources.

First, virus writers use some packers [40] to transform their binary code. Some virus even pack its code several times to make it even harder to unpack. IDA-Pro [41] equipped with a Universal PE Unpacker Plug-in [42] deals with packed code. This approach is fast, however, to use this plug-in, user need to know the original entry point of the binary code. This approach is specific to deal with certain packed code. Two other groups [43,44] describe general frameworks to reveal unpacked code. The problem of both approaches are the poor performance. Due to single step debugging mode, the execution is several magnitudes slower. Further more, this method makes unpacking some binary code virtually impossible.

Second, even if the program is not packed, disassembling is sometime still hard. The problem is the control flow transfer destination obfustication, which makes it hard to determine the starting point for each new block. This problem is addressed by [45,46,47,48,49]. The authors give some heuristics to relieve the problem. However, no general solution has been yet proposed for this problem.

Besides static analysis, dynamic approaches have also been used in virus research. Most of current approaches are based on system calls collected from runtime.

TTAnalyze [50] is a tool to execute an unknown executable inside a virtual machine. It captures all system calls and parameters of each system call to a log file. An expert can check the log file to find any malicious behavior manually.

Sung A.H.et al [51] propose to collect API call sequences from assembly code and compare these sequences to known malicious API call sequences.

Steven A. Hofmeyr et al [52] propose one of the very first data mining approaches using dynamic system call. They build an N-Gram system call sequences database for benign programs. For every unknown executable, they obtain system call sequences N-Grams and compare it with the database, if they cannot find a similar N-Gram, then the detection system triggers alert.

In [53], the authors propose several data mining approaches based on system call N-Gram as well. They try to build a benign and malicious system call N-Gram database, and obtain rules from this database. For unknown system call trace, they calculate a score based on the count of benign N-Gram and malicious N-Gram. In the same paper, they also propose an approach to use first (n-1) system calls inside N-Gram as features to predict the nth system call. The average violation score determines the nature of the unknown executable.

In [54], the authors also use system call N-Gram. They compare three approaches based on simple N-Gram frequency, data mining and hidden Markov model (HMM) approach, and conclude that though HMM is slow, it usually leads to the most accurate model.

In [55], the authors run virus executables inside a virtual machine, collecting operating system call sequences of that program. The authors intend to cluster viruses into groups. The authors use k-medoid clustering algorithm to group the viruses, and use Levenshtein distance to calculate the distance between operating system call sequences of different runtime traces.

Christodorescu et al [56] examines into dynamic system calls and their arguments. The arguments among different system calls reveal def-use, ordering, value dependent relationship between dependency among system calls. System calls form chains and loops based on these relationship. The authors then compute contrast subgraph by excluding benign behavior from the dependency graph to generate virus template to detect a family of viruses.

Another research area to detect malicious software dynamically is to detect the anomaly behavior of the victim software. Victim software should conform to its design specification unless the software is compromised. Chari et al [57] proposes a policy driven anomaly detection system. The policy used in this paper is per program based and derived manually. Provos [58] also uses a rule based per program policy. The authors look into the parameters used by each system call, and based on the value of parameters, permitting or denying the system call. The policy is derived automatically by a learning process. However, the learning algorithm is very basic. Bailey et al [59] cluster malware uses a higher level abstraction instead of system calls. They use normalized compression distance (NCD) to measure distance between two individual malware instances, then use a hierarchical clustering algorithm to group malware.

Another approach is to derive the policy rule from static analysis [60]. The authors derive system call policy from source code of the executable. They build non-deterministic finite automaton(NDFA), non-deterministic pushdown automaton(NDPDA) and digraph model out of the source code and deploy a runtime monitor system to check the conformance of each system call.

Rabek et al [61] does not require source code. They disassemble the code and extract address and the name of the system call. Comparing to [60], it is simpler and more general, but the false negative rate is higher. In [62], policy is built from dynamic trace. The policy is represented by an automaton. The state of the automaton is the address of the caller and the transition of the automaton is system call. Each program will need one such automaton. Feng et al [63] uses a similar dynamic approach to build policy. They use the whole stack information instead of address to make the policy more specific. More specific policy means a lower false negative rate. In [64], the authors construct a control flow graph (CFG) from the dynamic trace. They claim that the CFG derived dynamically is as good as CFG derived from statical analysis. Chaturvedi et al [65] also derives policy from dynamic trace. However, their focus is to learn the system call arguments rather than control flow information.

Some other approaches are based on dynamic taint analysis [66]. The idea is to keep track of suspicious memory section inside program. Misuse of these memory section triggers the warning. Egele et al [67] keep track of sensitive data by using dynamic taint. If the software manage to send sensitive information out, then it is suspected to be a spyware.

Besides data dependency, [68] also tries control flow dependency. They use Qemu [69]to explore multiple path execution. For every conditional branch based on input data, they create a snapshot, add constraint then do a depth first exploration. After one branch has been fully explored, they restore the state to previous snapshot, change the condition value to explore next branch.

All those researches show the efforts and achievements in unknown program classification and analysis.

CHAPTER 3: DYNAMIC EXECUTION MONITOR

3.1 Overview

Our detection algorithm is based on dynamic instruction sequences. Dynamic instruction sequences collection requires running the benign or malicious code. However, running malicious code is potentially destructive. Running program inside a virtual machine such as VMware [70] protects the computer from being infected directly. For virus researchers, running a program inside a virtual machine is a norm; However, most end user computers do not have virtual machine software installed and it is not realistic to ask end user to run an unknown program inside a virtual machine just to capture dynamic instruction sequences. To this end, we develop a protection mechanism called dynamic execution monitor and run the program within it. The main functionality for our dynamic execution monitor is to capture the instruction sequences of an arbitrary executable without destroying end user's computer. This dynamic execution monitor is run from the user computer and it provides control mechanism after execution of each instruction.

Figure 3.1 shows dynamic execution monitor we have built. The protection is provided by a sandbox to ensure malicious behavior is contained. The sandbox hooks potentially destructive Win32 API functions. Instead of simply ignoring the function and returning failure to the caller, for most functions, we return success and invoke the function in a restrictive way to elicit more malicious behavior. For example, when a program requests a file handler with read/write permission, we return success and give the handler with read permission only. In this way, we can elicit malicious behavior as much as possible. Some viruses would stop execution if they detect failure of a particular system call.

We capture dynamic instruction sequences or system call depending on backend we are using.

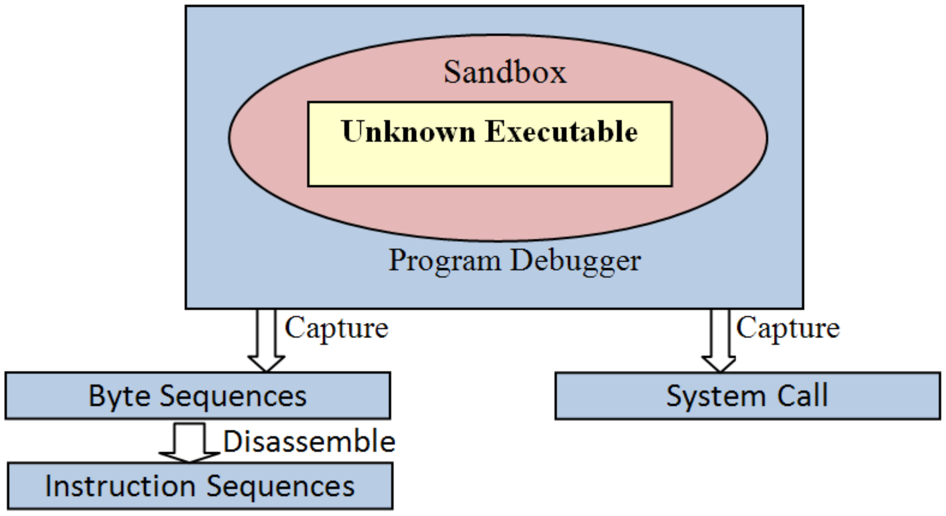


Figure 3.1: Dynamic Instruction Sequence Capturing System

Dynamic instruction sequences are captured by our program debugger. We will put the unknown program running under single instruction execution mode. At single instruction execution mode, it is possible to log the current instruction into instruction sequences log file.

We have two different backend for the monitor.

In dynamic instruction sequences backend, we embed the code to organize the code into logic assembly (See chapter 4), and then simplify it into abstract assembly. Also we have c4.5 and support vector machine code built into our monitor. Once we have abstract assembly, we will invoke c4.5 or support vector machine classifier to make an intelligent guess whether the unknown executable is malicious or not.

In system call backend, we construct structural system call diagram based on captured system calls. We then generate smart signature and use the smart signature to determine the nature of the unknown program.

3.2 Malicious Behavior Sandbox

The first goal of our dynamic execution monitor is to build a sandbox and run the unknown executable inside the sandbox to contain potentially malicious behavior.

Malicious behavior can be either permanent or temporary. Permanent effect persists even if the user restarts his or her computer. Permanent effect of a malicious code is achieved by modifying the content of end user's hard disk. This can be achieved by:

(i) Modify system executables or dynamic linked libraries (DLLs)

(ii) Modify frequently used application executables or dynamic linked libraries

(iii) Leave executable modules in the end user's computer, and modify the registry entry to point to that executable module

(iv) Change system settings, such as system clock, computer names, etc

These compromises can be achieved by the following types of Win32 APIs:

(i) File manipulation API

(ii) Registry manipulation API

(iii) System administration API

If a Win32 API is read only, we consider it safe. If a Win32 API is able to modify, create or delete entries, we need to take action to prevent the permanent effect on end user's computer.

Temporary effect of a malicious code is the compromise of end user's memory. Once the end user restarts his or her computer, the malicious behavior disappears. However, during the up time of the infected computer, some severe consequences may already happen. These include:

(i) Send out packets to predefined site to launch DDos attack

(ii) Send out private information on the end user's computer to malicious users

(iii) Send out infectious packets to compromise other computers

(iv) Send or receive packets to or from other computers on behalf of the malicious user

These kinds of malicious effects can be contained by restraining the packets sent by the host computer.

Some malicious software do not compromise the victim computer directly. Instead, it injects malicious code into an innocent process of the system, and let the victim process do all the malicious work. This makes the malicious software more stealthy in that it diverts the malicious behavior to the process we are not focusing on.

According to [71], there are three ways to inject code into another process. They are:

(i) Install a system wide remote hook by invoking "SetWindowsHookEx"

(ii) Create a remote thread and force the remote thread to load a malicious dynamic linked library, the initialization code of that dynamic linked library will lead to the execution of the injected code

(iii) Inject code into remote process by invoking "WriteProcessMemory", and then create a remote thread to execute the injected code

To prevent this type of attack, we need to restrict the invocation of the following Win32 APIs:

(i) SetWindowsHookEx

(ii) WriteProcessMemory

(iii) CreateRemoteThread

Another consequence of malicious code is disrupting the normal operation of the victim computer. However, compare to the above consequences, this temporary effect is relatively minor and insignificant. Containing all malicious behavior is impossible because some malicious behavior is hard to be distinguished from normal operations without human intelligence. So the focus of our monitor is to contain permanent or severe malicious behavior only, not all malicious behavior. Containing all malicious behavior is costly and impossible.

Currently, our monitor only protects one process and does not provide protection to processes launched by that process. To make sure we do not launch a malicious process, we do not allow process creation API for the executable process under protection for the time being.

3.3 Elicit Malicious Behaviors

By skipping the potentially destructive Win32 APIs, we can ensure most permanent or severe malicious behaviors are contained. However, the ultimate goal for our dynamic execution monitor is to capture the instruction sequences and analyze the malicious patterns. For a good programming practice, programmers need to verify the return value and output parameters of each Win32 API invocation. For most Win32 APIs, the return value is boolean and indicates whether the Win32 API is successful or not. If the return value or output parameters indicate that there is an error during the API invocation, the program will usually divert from the normal execution flow and handle the exception. However, for our purpose, we are more interested in normal execution flow of an executable rather than the exception handling. So simply skipping the potentially destructive Win32 API and throwing an error to the invoking program does not serve our purpose; hence, for most potentially destructive Win32 APIs, we skip the action but return a success to the caller. The invoking program sees the success of the Win32 API and continues the normal execution. Further, for some Win32 APIs which have output parameters, we set it to a typical value if possible.

For example, Win32 API "WriteFile" has an output value for the actual bytes written, we set it to the bytes to write. This gives the calling program the impression that all bytes are written successfully. It is possible that the program will eventually fail because of the skipped action of that Win32 API, but before that, we are able to capture more instruction sequences than otherwise.

The return value of some Win32 APIs are system handles. These Win32 APIs include CreateFileMapping, CreateFile, OpenFileMapping, etc. If these Win32 APIs fail, the return values are NULL. To elicit malicious behavior, we do not return NULL to the invoking program. Instead, we create or open a restrictive system handle. That is, no matter what access the program requests, we only create or open a read only system handle if possible. The invoking program will get a valid handle and continue normal processing. The API for consequent manipulations of the system handle are also contained. So unless the invoking program depends on the actual consequences of that Win32 API, the executable will continue its normal execution flow.

Some program will check whether it is running inside a debugger. If it is, the program takes a different execution path to avoid being analyzed. One of the easiest and most popular approach is to invoke "IsDebuggerPresent" Win32 API. To make sure that in this case, the program is able to take the normal execution path, we always answer "no" to "IsDebuggerPresent" Win32 API.

3.4 API Protection Policy

We provide protection to satisfy above mentioned requirements in our sandbox. However, we have another constraint. Currently, our sandbox only provide protection to the executable module and also dynamic linked libraries with static binding. Dynamic bound DLLs are not protected for the time being. To make sure that the executable is well protected by our sandbox, we also modify the behavior of "LoadLibrary" and "GetProcAddress". When a program invoke “LoadLibrary”, we do not actually load the dynamic linked library, instead, we return a fake module handle. “GetProcAddress” will recognize the fake handle and return a fake function pointer. The invocation of the fake function pointer will fail, but before that, we may be able to capture a lot more instruction sequences. However, the inability to protect Win32 API inside dynamic bound DLLs is a constraint and we plan to improve it in our future work.

Based on the discussion above, our sandbox manipulates the executions of the following categories of Win32 APIs.

(i) File/directory/registry manipulation API

(ii) Remote process/thread/memory manipulation

(iii) LoadLibrary/GetProcAddress

(iv) System administration API (system clock, etc)

(v) Packet sending

Table 3.1 lists all the functions we manipulate. For most of these Win32 APIs, we simply skip the action and return success to the invoking program. Some Win32 APIs need special treatment as we mentioned above. Those functions are underlined in the table.

3.5 Implementation

Windows system calls are invoked through several dynamic linked libraries. The most important libraries include “kernel32.dll”, “user32.dll”, “advapi32.dll” and “winsock.dll”. “kernel32.dll” contains core operating system functions such as process management, memory management and io management. “advapi32.dll” is a supplement to “kernel32.dll” and it contains new functionality such as registry management API. “user32.dll” contains GUI related functionality and “winsock.dll” contains WinSock API.

Table 3.1: Intercepted Functions

CopyFile	CopyFileEx	CopyMetaFile
CreateDirectory	CreateDirectoryEx	CreateFileMapping
CreateFile	CreateProcess	CreateRemoteThread
DeleteFile	GetProcAddress	IsDebuggerPresent
LoadLibrary	LoadLibraryEx	MoveFile
MoveFileEx	OpenFile	OpenFileMapping
RemoveDirectory	SetEndOfFile	SetFileAttributes
SetFilePointer	SetFileTime	SetSystemTime
SetupComm	WSAConnect	WSASend
WSASendTo	WriteFile	WriteFileEx
WritePrivateProfileSection	WritePrivateProfileString	WritePrivateProfileStruct
WriteProcessMemory	WriteProfileSection	WriteProfileString
send	sendto	RegOpenKeyEx
RegCreateKeyEx	RegDeleteKey	RegDeleteKeyEx
RegDeleteValue	RegLoadKey	RegOpenCurrentUser
RegOpenUserClassesRoot	RegReplaceKey	RegRestoreKey
RegSaveKey	RegSaveKeyEx	RegSetValueEx

These dynamic linked libraries are Win32 specific and are well documented. The APIs are consistent as Windows operating system evolves. However, Win32 is only one of the subsystems Windows operating system provides. Internally, these dynamic linked libraries are merely a wrap over the underlying system functions. The shared underlying module for all subsystems is "ntdll.dll". Intercepting "ntdll.dll" will protect all subsystems of Windows operating system. Intercepting "ntdll.dll" is possible, however, "ntdll.dll" is undocumented and its API changes constantly even for a minor upgrade for Windows.

Most operating system functions have to be performed in kernel mode. So in most of time, "ntdll.dll" acts as a gateway to the kernel code. Kernel code is accessed by invoking "Sysenter/Syscall" instruction or "int 2E" interrupt. Kernel code to serve Win32 APIs is provided by the module "ntoskrnl.exe".

Figure 3.2 illustrates architecture of Windows system.

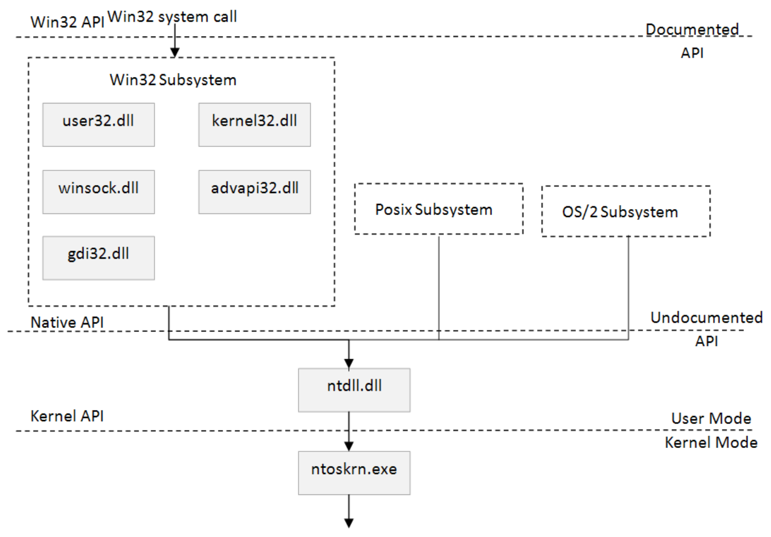


Figure 3.2: Win32 Architecture

There are a couple of ways to hook customized code into the Win32 APIs. The first approach is to intercept higher level Win32 API calls. These APIs are documented and consistent through different versions of Windows.

The second approach is to intercept “ntdll.dll”. The upside is that some malicious software invokes APIs in “ntdll.dll” directly to bypass higher level Win32 API hooks. If that happens, “ntdll.dll” hook is still effective. The downside is that “ntdll.dll” is undocumented and changes fast. It makes it hard to write the hook code and the code requires lots of maintenance as the operating system evolves.

It is also possible to hook kernel code. The key is the service descriptor tables structure defined by Windows operating system. It is a mapping from dispatch ID to the entry point of service code. Changing entry point in the service descriptor divert the regular execution of a service code. User can put their own logic into the existing operating system code. The problem is the same with “ntdll.dll” approach, that is, no documentation and complicated code maintenance.

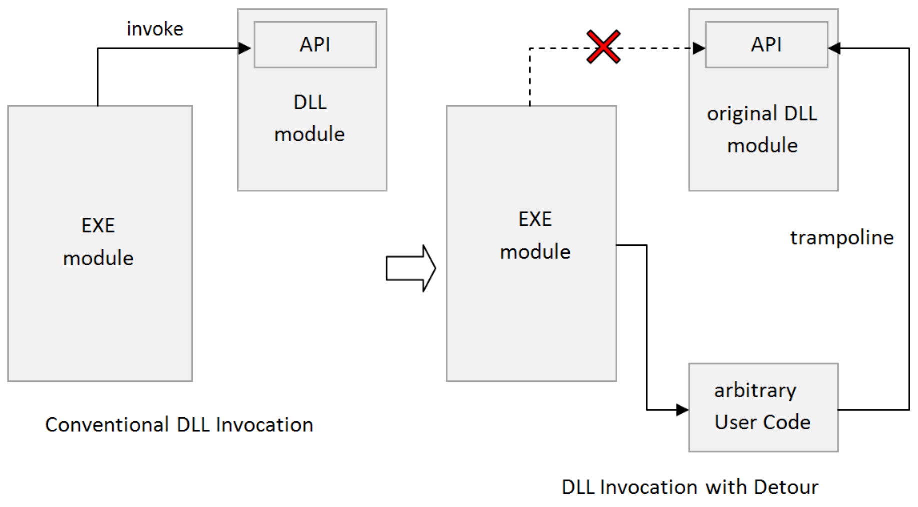


Figure 3.3: Trampoline

We choose to use high level Win32 API interception in our experiment. We use Microsoft detour library [72] to implement our protection mechanism. Detour is a library freely available from Microsoft Research. Detour hook arbitrary user code onto Win32 API call. It provide API call to modify the first several instructions of the target function with a “jmp” instruction. “jmp” instruction diverts the execution flow to the user supplied code. However, the replaced instructions of the target function are not lost. They are saved by Detour to a newly appended section of the executable code. Inside the user supplied code, the user can optionally invoke the original version of the system call through a trampoline function. Figure 3.3 illustrates the structure of a trampoline function.

3.6 Instruction Capturing

To capture the dynamic instruction sequences of an unknown program, we create process of the program and put it in single instruction execution mode. Our monitor system will capture debug events and log the instructions executed by the unknown program.

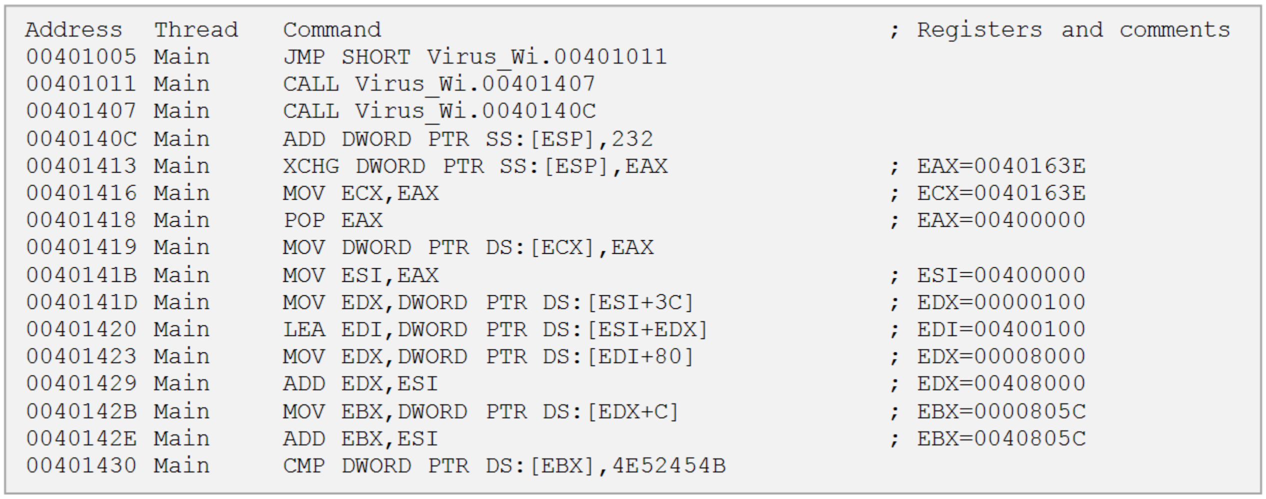
To focus on the behavior of the unknown program module, we only capture the binary code of the module itself and ignore instructions inside dynamic linked library modules.

Since we do not capture instruction sequences inside dynamic linked library modules, we do not need to put the processor into single instruction execution mode while it is running the code from dynamic linked library modules. Before the control flow returns to the main executable, we do not need to capture any instruction sequences. We shall let the code run in full speed. To achieve this, we monitor the control flow of the executable while it is running under our debugger. Once it invokes a dynamic system call to other module, we put a break point on the next instruction in the executable module, and then put the process out of single step debugging mode. Once the control flow comes back, it stops at the break point and we can put it into single step debugging mode again.

The speed of instruction sequences capturing depends on how much time our monitor is actually capturing instruction sequences. During external dynamic library calls, which include Win32 API call and user supplied external functions, our monitor do not capture instruction sequences. However, the speed to capture instruction sequences is much more consistent than the speed to capture system calls. Usually the monitor system can capture about 6,000 instruction sequences on our computer. We also put a time limit on our monitor. The monitored executable stops execution anyway after 10 seconds even if it cannot capture enough instruction sequences.

This approach is very similar to the popular debugging tool OllyDbg [73]. The "instruction trace" functionality provided by OllyDbg is able to generate runtime instruction trace in the same form as our monitor. The speed of instruction capturing is similar. However, comparing to use OllyDbg directly, our dynamic instruction monitor has two advantages:

(i) Our monitor captures the instructions in a more controlled way. We can have an up bound for the instructions we are planning to capture. We also have a timeout limit to stop the instruction sequences capturing to make sure the instruction capturing process is finished within an acceptable amount of time



```
Address   Thread   Command                                       ; Registers and comments
00401005 Main      JMP SHORT Virus_Wi.00401011
00401011 Main      CALL Virus_Wi.00401407
00401407 Main      CALL Virus_Wi.0040140C
0040140C Main      ADD DWORD PTR SS:[ESP],232
00401413 Main      XCHG DWORD PTR SS:[ESP],EAX                   ; EAX=0040163E
00401416 Main      MOV ECX,EAX                                   ; ECX=0040163E
00401418 Main      POP EAX                                       ; EAX=00400000
00401419 Main      MOV DWORD PTR DS:[ECX],EAX
0040141B Main      MOV ESI,EAX                                   ; ESI=00400000
0040141D Main      MOV EDX,DWORD PTR DS:[ESI+3C]                 ; EDX=00000100
00401420 Main      LEA EDI,DWORD PTR DS:[ESI+EDX]                ; EDI=00400100
00401423 Main      MOV EDX,DWORD PTR DS:[EDI+80]                 ; EDX=00008000
00401429 Main      ADD EDX,ESI                                   ; EDX=00408000
0040142B Main      MOV EBX,DWORD PTR DS:[EDX+C]                  ; EBX=0000805C
0040142E Main      ADD EBX,ESI                                   ; EBX=0040805C
00401430 Main      CMP DWORD PTR DS:[EBX],4E52454B
```

Figure 3.4: Captured Instruction Sequences

(ii) Our monitor runs under the sandbox and potential malicious behavior of the unknown executable is contained

Figure 3.4 illustrates the captured instruction sequences of an executable.

Once we capture the binary codes, we then disassemble the binary code. From the assembly code, we can capture instruction sequences we need.

3.7 Backend

From the captured dynamic instruction sequences, we further process it in our backend. For dynamic instruction sequences backend, further processing include:

(i) Reorganize the dynamic instruction sequences into logic assembly

(ii) Simplify logic assembly into abstract assembly

(iii) Extract top instruction associations from abstract assembly

(iv) Apply classification model on extracted features and make predication. We use either C4.5 decision tree or support vector machine classifiers

Details of these steps will be elaborated in chapter 4.

For structural system call diagram backend, further processing include:

(i) Construct structural system call diagram

(ii) Generate smart signature based on modularized system call blocks

(iii) Use the smart signature to determine the nature of unknown program

Details of structural system call diagram can be found in chapter 5

3.8 Limitation

There are several limitations exist in our system.

(i) Some malicious program will invoke the underlying operating system native API in ntdll.dll or invoke interrupt 2e (or SysEnter/SysCall, on newer versions of Windows) directly, which is not protected by our system.

(ii) Only one execution path will be explored during one invocation of the monitored executable

(iii) Only initial part of the executable is considered

For 1, we are trying to relieve it by hooking the kernel code instead of high level Win32 API. Problem 2 and 3 are common in all dynamic virus detection approaches. Dynamic approaches alone can hardly solve these problems. We shall explore the possibilities to use both dynamic approaches and static approaches to find a solution.

CHAPTER 4: DETECTING WIN32 VIRUSES USING DYNAMIC INSTRUCTION SEQUENCES

In this chapter, we present a novel approach to detect unknown virus using dynamic instruction sequences mining techniques. We collect runtime instruction sequences from unknown executables and organize instruction sequences into basic blocks. We extract instruction sequence patterns based on three types of instruction associations within derived basic blocks. Following a data mining process, we perform feature extraction, feature selection and then build a classification model to learn instruction association patterns from both benign and malicious dataset automatically. By applying this classification model, we can predict the nature of an unknown program. We also build a program monitor which is able to capture runtime instruction sequences of an arbitrary program. The monitor utilizes the derived classification model to make an intelligent guess based on the information extracted from instruction sequences to decide whether the tested program is benign or malicious. Our result shows that our approach is accurate, reliable and efficient.

We start with system overview in 4.1, logic assembly in 4.2, the concept of instruct association in 4.3, data mining process in 4.4, performance analysis in 4.5, our experimental results in 4.7, comparison with other approaches in 4.8, an analysis tool to provide a shell to this approaches in 4.9.

4.1 SYSTEM OVERVIEW

Figure 4.1 shows the overview of our system. It includes both training process and the decision process. We have already discussed dynamic instruction collection process before. So start from dynamic instruction sequences, the training process derives logic assembly and abstract assembly. From abstract assembly, we can derive our feature vector. Through our feature selection and modeling process, we can build our classification model. On the end user

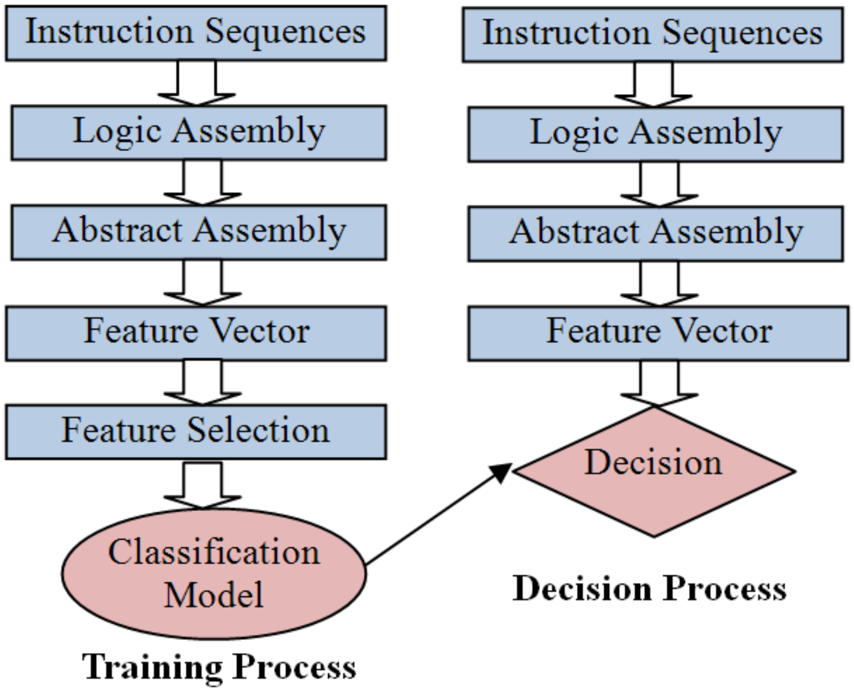
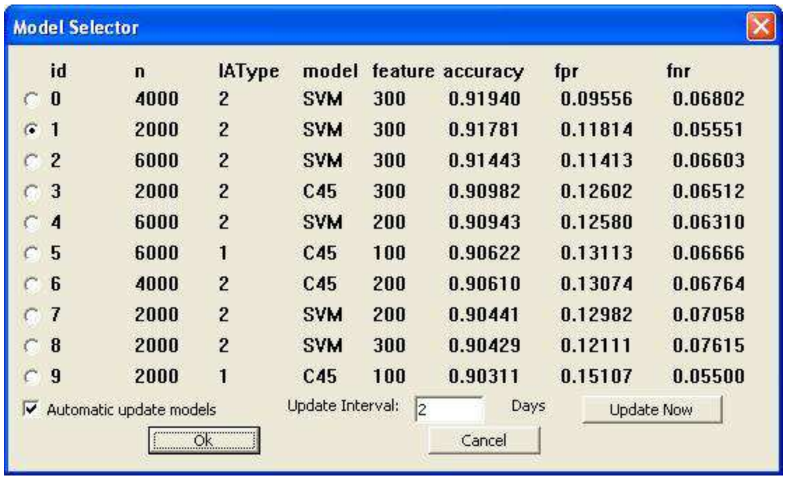


Figure 4.1: System Overview

side, we install a monitoring system which will run unknown program under our protection mechanism. Based on the captured dynamic instruction sequences we run a decision process which derives logic assembly and abstract assembly, collects all the required features of the classification model, and uses these features to make our decision by the classification model.

Every classification model includes model parameters, feature set, and classifier. On the end user's computer, models are constantly updated through an automatic update process. Latest models are built using the up to date virus and benign dataset to maintain the quality of the classification model. End user also has the ability to change preferred model through a GUI, see Figure 4.2. In this dialog, 10 best models are sorted on accuracy (over testing dataset). The user can see how many instructions the model collects (n), what type of instruction association does it use (IAType), what is the classifier (model), how many features does it use (feature), what is false positive rate (fpr) and false negative rate (fnr).


Model Selector

	id	n	IAType	model	feature	accuracy	fpr	fnr
○	0	4000	2	SVM	300	0.91940	0.09556	0.06802
●	1	2000	2	SVM	300	0.91781	0.11814	0.05551
○	2	6000	2	SVM	300	0.91443	0.11413	0.06603
○	3	2000	2	C45	300	0.90982	0.12602	0.06512
○	4	6000	2	SVM	200	0.90943	0.12580	0.06310
○	5	6000	1	C45	100	0.90622	0.13113	0.06666
○	6	4000	2	C45	200	0.90610	0.13074	0.06764
○	7	2000	2	SVM	200	0.90441	0.12982	0.07058
○	8	2000	2	SVM	300	0.90429	0.12111	0.07615
○	9	2000	1	C45	100	0.90311	0.15107	0.05500

[✓] Automatic update models   Update Interval: 2 Days

Figure 4.2: Model Selector

4.2 LOGIC ASSEMBLY

Our dynamic instruction capturing system captures execution log at a rate around 6,000 instructions per second on our computer. For some executables requiring interaction, we use the most straightforward way, such as typing "enter" key in a command line application or press "Ok" button in a GUI application to respond.

In a conventional disassembler, assembly instructions are organized into basic blocks. A basic block is a sequence of instructions without any jump targets in the middle. Usually disassembler will generate a label for each basic block automatically. However, execution log generated by our system is simply a chronological record of all instructions executed. The instructions do not group into basic blocks and there are no labels. We believe that basic blocks capture the structure of instruction sequences and thus we process the instruction traces and organize them into basic blocks. We call the resulting assembly code "logic assembly".

Compared with assembly code, dynamic captured instruction sequences may have incomplete code coverage. This fact implies the following consequences about logic assembly code:

(i) Some basic blocks may be completely missing

(ii) Some basic blocks may contain less instructions

(iii) Some jump targets may be missing, which makes two basic blocks merge together

Despite these differences, logic assembly carries as much structural information of a program as possible.

We design the algorithm to construct logic assembly from runtime instructions trace. The algorithm consists of three steps and we describe below:

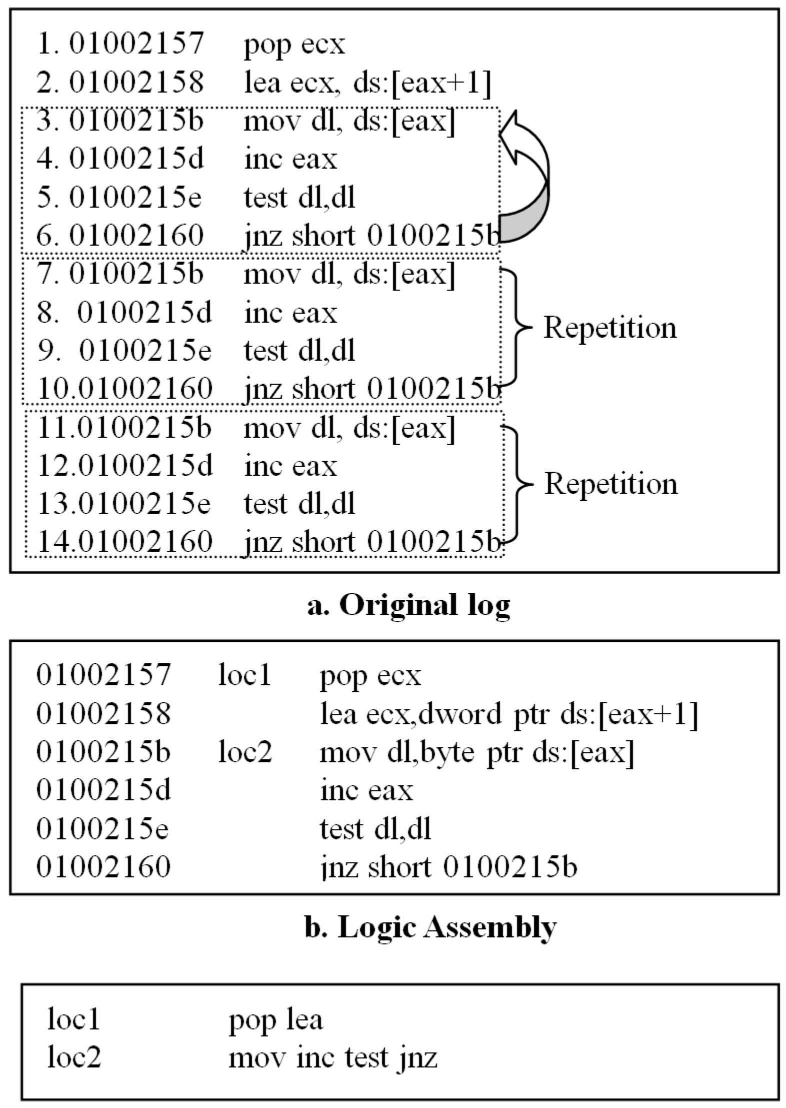
(i) Sort all instructions in the execution log on their virtual addresses. Repeated code fragments will be ignored

(ii) Scan all jump instructions. If the address of an instruction is a destination of control flow transfer instruction (conditional or unconditional), or the previous instruction is a control flow transfer instruction, we mark it as the beginning of a new basic block

(iii) Output sorted instructions and generate labels if applicable

We further prove that instructions inside a basic block are consecutive. Here consecutive means there is no hole inside address space representing the binary code of the basic block. If not, suppose there is a gap between adjacent instruction i_1 and i_2 inside the same basic block, then there must be a control flow transfer instruction j leads to i_2, and j must be right before i_2. So j should be in our chronicle instruction log as well. We will generate a new label for instruction i_2. Then i_1 and i_2 should be in different basic blocks. This contradicts our initial assumption.

Each instruction usually consists of operation code (opcode) and operands. Some instruction sets such as 80x86 also have instruction prefix to represent repetition, condition, etc. We pay attention to the opcode and ignore the operands and prefix since the opcode represents the behavior of the program. The resulting assembly code is called abstract assembly [74].



```
1. 01002157    pop ecx
2. 01002158    lea ecx, ds:[eax+1]
3. 0100215b    mov dl, ds:[eax]
4. 0100215d    inc eax
5. 0100215e    test dl,dl
6. 01002160    jnz short 0100215b
7. 0100215b    mov dl, ds:[eax]
8.  0100215d   inc eax
9.  0100215e   test dl,dl
10.01002160    jnz short 0100215b
11.0100215b    mov dl, ds:[eax]
12.0100215d    inc eax
13.0100215e    test dl,dl
14.01002160    jnz short 0100215b
```

a. Original log

```
01002157    loc1    pop ecx
01002158            lea ecx,dword ptr ds:[eax+1]
0100215b    loc2    mov dl,byte ptr ds:[eax]
0100215d            inc eax
0100215e            test dl,dl
01002160            jnz short 0100215b
```

b. Logic Assembly

```
loc1        pop lea
loc2        mov inc test jnz
```

c. Abstract Assembly

Figure 4.3: Logic Assembly and Abstract Assembly

Figure 4.3 shows an example of logic assembly and abstract assembly. Figure 4.3.a is the original instruction sequences captured by our system. We remove duplicated code from line 7 to line 14, and generate label for jump destination line 3. Figure 4.3.b is the logic assembly we generated. We further omit the operands and keep only opcode, and we finally get abstract assembly as in figure 4.3.c.

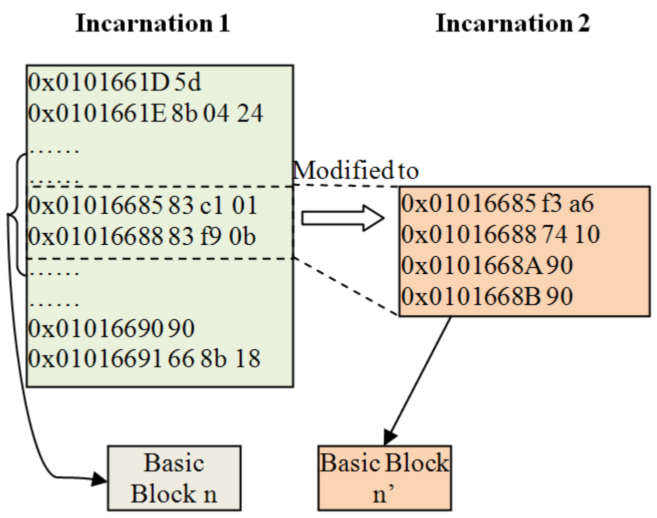


Figure 4.4: Different Incarnations

One merit of using dynamic instruction sequences is that dynamic instruction sequences expose some type of self-modifying behavior. If a program modifies its code at runtime, we observe two different instructions at the same virtual address in runtime trace. A program may even modify its own code more than once. We devise a mechanism to capture this behavior while constructing logic assembly.

We associate an incarnation number for each virtual address we have seen in the dynamic instruction sequences. Initial incarnation number is 1. Each time we see an instruction at the a virtual address, we compare this instruction with the one we have seen before at that virtual address, if there is any. If the instruction changes, we increment the incarnation number. Subsequent jump instruction will mark the beginning of a basic block on the newest incarnation. We treat instructions of different incarnations as different code segments, and generate basic blocks separately. Figure 4.4 illustrates this process. In this way we keep the behavior of any historical invocations even if the code is later overwritten by newly generated code.

4.3 INSTRUCTION ASSOCIATIONS

With abstract assembly we are interested in finding relationship among instructions within each basic block. We believe the way instruction sequences group together within each block carries the information of the behavior of an executable.

The instruction sequences we are interested in are not limited to consecutive and ordered sequences. Virus writers frequently change the order of instructions and insert irrelevant instructions manually to create a new virus variation. Further, metamorphic viruses [3] automatically change the order of sequences and insert garbage among useful instructions to make new virus copies. The resulting virus variation still exhibits the malicious behavior. However, any detection mechanism based on consecutive and ordered sequences such as N-Gram could be fooled.

We have two considerations to obtain the relationship among instructions. First, whether the order of instructions matters or not; Second, whether the instructions should be consecutive or not. Based on these two criteria, we use three methods to collect features.

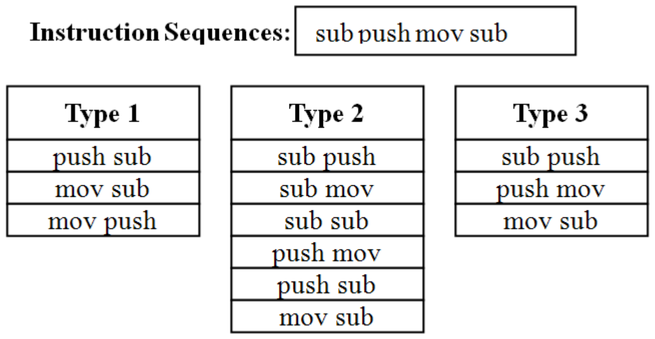
(i) The order of the sequences is not considered and there could be instructions in between

(ii) The order of instructions is considered. However, it is not necessary for instruction episodes to be consecutive

(iii) The instructions are both ordered and consecutive

We call these "Type 1", "Type 2" and "Type 3" instruction associations. "Type 3" instruction association is similar to N-Gram. "Type 2" instruction association can deal with garbage insertion. "Type 1" instruction association can deal with both garbage insertion and code reordering.

Figure 4.5 illustrates different types of instruction associations of length 2 we have obtained on an instruction sequences consisting of 4 instructions.


Instruction Sequences: sub push mov sub

Type 1
push sub
mov sub
mov push

Type 2
sub push
sub mov
sub sub
push mov
push sub
mov sub

Type 3
sub push
push mov
mov sub

Figure 4.5: Example for instruction associations of length 2

4.4 DATA MINING

4.4.1 Process Overview

The overall data mining process can be divided into 7 steps. They are:

(i) Run executable inside a virtual machine, obtain instruction sequences from our system

(ii) Construct logic assembly

(iii) Generate abstract assembly

(iv) Feature selection

(v) Generating training dataset and testing dataset

(vi) Build classification models

(vii) Evaluation

This process is illustrated in Figure 4.6.

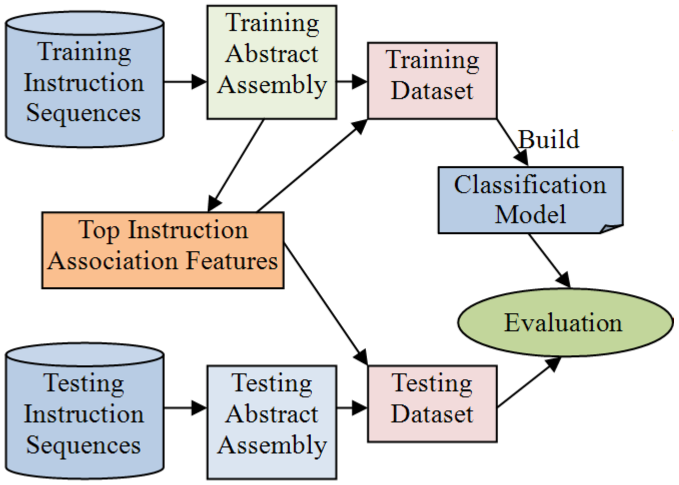

Training Instruction Sequences
Training Abstract Assembly
Training Dataset
Build
Classification Model
Top Instruction Association Features
Evaluation
Testing Instruction Sequences
Testing Abstract Assembly
Testing Dataset

Figure 4.6: Data Mining Process

4.4.2 Feature Selection

Here we describe step iv which is feature selection in detail. The features for our classifier are instruction associations. To select appropriate instruction associations, we use the following two criteria:

(i) The instruction associations should be frequent in the training dataset consisting of both benign and malicious executables. If it occurs very rarely, we would rather consider this instruction association is a noise and not use it as our feature

(ii) The instruction associations should be an indicator of benign or malicious code; In other words, it should be abundant in benign code and rare in malicious code, or vice versa

To satisfy the first criteria, we only extract frequent instruction associations from training dataset. Only frequent instruction associations will be considered as our feature.

We use a variation of Apriori algorithm [75] to generate all three types of frequent instruction associations from abstract assembly.

The basic idea of the Apriori algorithm in our experiment is to generate candidate instruction combinations of particular size and then scan the whole file to count these to see

```
k = 0
$C_1$ = I  // Initial candidates to 1-instruction
repeat
      k = k + 1
      scan whole file, count $C_k$
      $C_{k+1}$ = generate_candidate($C_k$)
until $C_{k+1}$ == $\Phi$
```

```
generate_candidate($C_k$)
      $C_{k+1}$ = all possible k+1 combinations
      for each S in $C_{k+1}$
            if any k-subset of S is not in $C_k$
                  remove S
      return $C_{k+1}$
```

Figure 4.7: Apriori Algorithm

if they are frequent enough [7]. One observation is that if one instruction combination is frequent, then all its subsets must be frequent. First, all 1-instruction will be counted. If the percentage of lines containing such instruction exceeds the minimum support, we will mark it frequent. All 2-instructions candidates will be generated based on frequent 1-instruction. Only the 2-instructions whose 1-instruction subset is frequent will be considered as a candidate. Another round of file scan will be needed and all 2-instructions candidates will be counted. This process continues until no longer candidates are generated as shown in figure 4.7.

Although there exist algorithms to optimize Apriori algorithm [76], the optimization is only applicable to type 1 instruction association. Besides, this step only occurs at training time. We believe optimizing decision process is more critical because it will run on each end user computer under protection. Training, however, only need to be done on some specific server computers.

One parameter of Apriori algorithm is "minimum support". It is the minimal frequency of frequent associations among all data. More specifically, it is the minimum percentage of basic blocks that contains the instruction sequences in our case. We have done experiments on different support level as described in section 4.7.

To satisfy the second criteria, we define the term contrast:

$$Contrast(Fi) = \begin{cases} \frac{Bcount(Fi)+\varepsilon}{Mcount(Fi)+\varepsilon} & Bcount(Fi) \geq Mcount(Fi) \\ \frac{countM(Fi)+\varepsilon}{Bcount(Fi)+\varepsilon} & Bcount(Fi) < Mcount(Fi) \end{cases} .$$

BCount (Fi)	normalized count of Fi in benign instruction file
MCount (Fi)	normalized count of Fi in malicious instruction file
ε	a small constant to avoid error when the denominator is 0

In this formula, normalized count is the frequency of that instruction sequences divided by the total number of basic blocks in abstract assembly. We use a slightly larger benign code dataset than malicious code dataset. The use of normalization will factor out the effect of unequal dataset size.

We select top L features as our feature set. For one executable in training dataset, we count the number of basic blocks containing the feature, normalized by the number of basic blocks of that executable. We process every executable in our training dataset, and eventually we generate the input for our classifier.

4.4.3 Classification

We use two classifiers in our experiments: decision tree and Support Vector Machine.

4.4.3.1 Decision Tree

Decision tree is a classification algorithm that is constructed by recursively splitting the dataset into parts. Each such split is determined by the result of the statistical test of all possible splits among all attributes inside the tree node. This statistical test can be Gini, entropy gain or Chi-square test. The decision tree keeps growing as more splits are performed until a specific stop rule is satisfied. Typical stop rules include minimum record number inside a tree node, maximum depth of the tree, and purity of the records in a node. After growing a full tree, an additional process called postpruning can be used to simplify the decision tree. During postpruning, some splits are removed while others are retained. Postpruning can reduce overfitting problem. Each leaf node is then tagged with majority class of its records and complete the construction of the decision tree.

Once a decision tree is built, it is used for classification of a new record. When a record of an unknown class comes in, it is classified through a sequence of nodes from the tree root down to the leaf node. Then, it is labeled by the class the leaf node represents. Depending on the choice of split criteria, stop rules, postpruning strategies, and some other properties, there are several variations of decision trees such as ID3 [77], C4.5 [78], CART [79], and CHAID [80].

C4.5 is a popular decision tree package based on ID3. It uses entropy gain as its splitting criteria. The splitting will stop only when all records within a node is of the same class or a node is too small. After C4.5 tree has fully grown, there is a pruning process. C4.5 takes both discreet and continuous features.

We use Quinlan's c4.5 decision tree [78] as our decision tree implementation.

4.4.3.2 Support Vector Machine

Support vector machine (SVM) [81, 82] receives attention since its advent. It is considered to be one of the most stable and accurate classifiers. SVM is essentially a mathematical optimization problem which has originated from the linear discriminant problem, see figure 4.8.a. There are two approaches for finding the optimal linear discriminant in a linear separable case. The first approach is to find the best plane bisects closest points in the convex hull, see figure 4.8.b and the second approach is to find the best plane maximizes the margin, see figure 4.8.c.

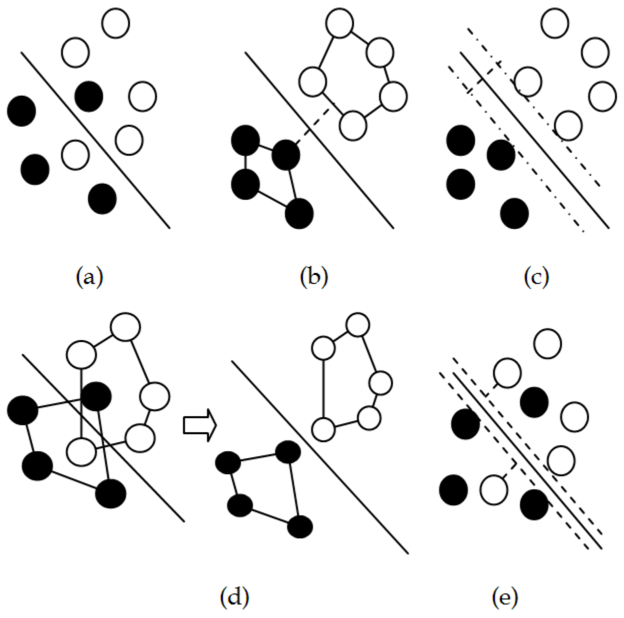


Figure 4.8: Support Vector Machine

The optimization method works even in a linear inseparable case. In convex hull approach, the convex hull will shrink to make two classes linear separable, see figure 4.8.d. In maximum margin approach, we allow some errors for the margin (soft margin), that means, some data may not obey the discriminant. We will calculate the total error term of these data, the smaller the error term, the better the discriminant, see figure 4.8.e.

SVM extends this simple linear case. The idea is, if two classes are inseparable in two dimensions such as figure 4.8.a, SVM can use a mapping, which is called kernel function, to map two dimension data into a higher dimension. The two classes may be separable in higher dimension. Even if it is still inseparable, the error term may be much smaller.

We use libSVM [83] package in our experiments. It is based on SMO (Sequential Minimal Optimization) [84], which is regarded as one of the best implementations for SVM. libSVM takes both categorical and continuous features. It is a C implementation of SVM on Unix.

4.4.3.3 Other Classifier

We also tested some other classifiers such as random forests [85]. We do not detect any classifier has clear advantage over others in the measure of accuracy. One reason leads us to use C4.5 and SVM in our experiment is that both classifiers are efficient to make decision. The performance of decision making process is the key to our system performance (See Section 4.5). Recently, we did some preliminary experiments on TreeNet [86]. we see improvements by using this new classifier. We need to more extensive experiment using TreeNet in our future work.

4.4.4 N-folds Cross Validation

Cross validation [87] is a method to generate relatively fair performance measurements from the dataset. It first divides the whole dataset randomly into n equal size folds. Every time we use n-1 folds as training dataset and use the remaining 1 fold as testing dataset. This process is repeated n times. Every fold is selected as testing dataset once. The overall performance is calculated by averaging all n results. The result of N-folds cross validation is more accurate than simple training dataset / testing dataset approach. This approach is especially useful for a small dataset. It keeps most data as training dataset, resulting in a more accurate classification model. By averaging n results, this approach does not suffer the randomness caused by a small testing dataset.

4.4.5 Ensemble Model

One approach to improve the classification accuracy is using an ensemble model which is a combination of several different models. Usually a majority vote will be performed and the winner decides the final classification of the unknown case. Compare to a single classifier, ensemble model is more robust and resistant to overfitting. However, it takes more time to build and apply an ensemble model since it will build several classifiers rather than one. Bagging [88] and boosting [89] are two most popular ensemble models.

4.4.6 Performance Measurement

In this thesis, performance of the classification models are measured by three criteria. They are accuracy, false positive rate and false negative rate. In a two-category classification problem, the classification result will be one of the following cases:

(i) true and classified as true (True Positive)

(ii) false but classified as true (False Positive)

(iii) false and classified as false (True Negative)

(iv) true but classified as false (False Negative)

And (All cases) = (True Positive) + (False Positive) + (True Negative) + (False Negative).

Here are the definition of accuracy, false positive rate (FPR) and false negative rate (FNR) respectively:

$$Accuracy = \frac{(TruePositive)+(TrueNegative)}{All}$$

$$FPR = \frac{FalsePositive}{(TruePositive)+(FalsePositive)}$$

$$FNR = \frac{FalseNegative}{(TrueNegative)+(FalseNegative)}$$

False positive rate is the proportion of benign executables that were erroneously reported as being malicious. On contrary, false negative rate is the proportion of malicious executables that were erroneously identified as benign.

In short, accuracy is the overall accuracy in testing dataset, false positive rate is the accuracy among all positive case, false negative rate is the accuracy among all negative case.

4.5 PERFORMANCE ANALYSIS

In this section, we focus on average performance in measure of speed when applying the classification model on the end user computer. The performance to process one unknown executable is determined by the following factors:

(i) Capturing instruction sequences

(ii) Generating logic assembly

(iii) Counting the occurrence of instruction associations in feature set to generate feature vector

(iv) Applying classification model

Unlike system calls, instruction sequences are generated fast and at a relative stable rate. On our test computer, we generate an average of 6,000 instruction sequences in 1 second inside our system. That is enough for the input for our classifier. This is one advantage over system call approach, which takes more time to collect enough traces.

However, instruction sequences collection rate is not really stable. We could spend much more time than average to capture enough instruction sequences. For example, some system calls will block the process so we cannot capture any instructions for this program module during system call. Some program will wait for user's response to continue executing. To solve this problem, we also specify a time limit on instruction capturing. We will stop capturing instruction sequences and continue to the next step when a timeout occurs.

Generating logic assembly consists of three phases. In the first phase, we need to sort the instruction sequences according to their virtual address. This could take up to O(nlogn) to finish, in which n is the number of instructions captured. In the second phase, we mark jump destination using one linear scan of all instructions, which takes O(n). Maintaining different incarnations requires a memory map to remember the instruction and incarnation of each virtual address. Every instruction takes linear time to check this memory map, so this additional task will not increase the order of the overall processing time. Finally, we traverse the sorted instruction list to output basic blocks, which takes O(n). So the overall time complexity in logic assembly generation is O(nlogn).

Generating features vector requires counting the frequency of L features. Suppose one basic block contains average k instructions, thus we have average n/k basic blocks. For every basic block, we will do a search for each one of L features.

Different types of instruction association require different approaches to search inside a basic block. For type 1 instruction association, we use an occurrence bit for every instruction in the association, if all bit is on, then the basic block contains that instruction association. For type 2, we construct a finite state machine (FSM), and scan the basic block from the beginning. If we encounter an instruction matching the state in FSM, we change the state of FSM, and begin matching the next instruction. For type 3, it is similar to a substring search. All these three types of search requires one linear scan of the basic block, makes the average complexity O(k).

We can calculate the processing time of feature vector generation as the product of the above factors. So this step takes (search time per feature per basic block)* (feature number) * (basic block number) = O(n/k*k*L) = O(nL).

The time complexity to apply a classifier is a property of that specific classifier.

For C4.5 decision tree, the applying time complexity is proportional to the depth of the tree [78], which is a constant at the decision time. SVM takes O(L) time to apply the model on a specific feature vector [90].

Based on the discussion above, we conclude that the average time complexity to process an unknown executable is bounded by max (O(nlogn), O(nL)), in which L is the number of features.

In our experiment, to process instructions captured in 1 second, for which n$\leq$6000, the calculation time is usually less than 3 seconds. This suggests that this approach can be used in practice.

4.6 Dataset

Due to the prevailing dominance of Win32 viruses today, we use Win32 viruses as our virus dataset. We collect 267 Win32 viruses from VX heaven [91].

We also choose 368 benign executables which consist of Windows system executables, commercial executables and open source executables. These executables have similar average size and variation as the malicious dataset.

Some viruses fail to execute fully and we can only collect less than 10 dynamic instructions. We exclude these viruses from our dataset due to lack of enough information for further processing. By manually analyzing these viruses, we found that these viruses quit early due to failure to identify or execute some system functions. This is usually due to the virus author making false assumption about the operating system which is not present in the destination system. For example, Win32.Alcaul.b is trying to write to a read only page at 5th instruction. While it is possible in earlier version of Windows, current version of Windows correct the problem and this operation is no longer possible. The virus code quit quickly due to access violation. All such viruses we have encountered are not actually presenting malicious behavior before they quit and thus would not destroy end user computer. So we believe ignoring these viruses in our system would not cause a big problem.

In one experiment, we try to use IDA Pro [41] to get a assembly code from each binary executable. Our study shows that a non-negligible amount of binary executables cannot be fully disassembled. Only around 50% of malicious executables can be fully disassembled and more than 10% of them cannot be disassembled at all; that means the disassembler recognizes no identifiable instructions in the binary executable at all. For benign executables,

Table 4.1: Dataset Statistics

	Benign	**Virus**
Number	368	267
Average Binary Size	126k	38k
STDEV(size)	319k	75k
Average Assembly Lines	20495	5717
STDEV(lines)	30566	18677

around 90% executables can be fully disassembled. In those cases, further disassembly can be achieved manually. However, manual disassembly requires advanced skills and is time consuming. This result justifies our choice of using dynamic approach in favor of static approach.

The dataset is publicly available at `http://www.cs.ucf.edu/~daijy/instructionsequence_dataset.zip`

Table 4.1 lists key statistics for our dataset.

4.7 EXPERIMENTAL RESULTS

4.7.1 Methodology

We use 5-fold cross-validation [7] to evaluate the performance in our experiments. We randomly divide the dataset into 5 subsamples of the same size. We then repeat the cross-validation process 5 times. Every time we use one subsample as testing dataset and all other subsamples as training dataset. Our result is the average of all 5 runs.

In out experiment, we use accuracy on testing dataset as our main criteria in evaluation of the performance of classification models. However, we also calculate false positive rate and false negative rate. We believe in a virus detection mechanism, low false negative rate is more vital than low false positive rate. It is wise to be more cautious against those suspicious unknown executables. High false positive rate certainly make things inconvenient for the user, but high false negative rate will destroy a user's computer, which is more harmful.

4.7.2 Parameter Selection

There are five primary parameters in our classifier, they are:

(i) Instruction association type IA (type 1, 2 or 3)

(ii) Support level of frequent instruction association (S), we experiment with 0.003, 0.005, 0.01

(iii) Number of features (L), we try 10, 20, 50, 100, 200, 300. At some support level, some instruction association types generate relatively fewer number of available features. For example, at support lever 0.01, only 23 type 1 instruction associations are frequent. In that case, we use up to the maximum available features

(iv) Type of classifier (C), we compare C4.5 decision tree and SVM (Support Vector Machine)

(v) Number of instruction captured (N). We try 1000, 2000, 4000, 6000, 8000

Table 4.2 list top 10 configurations we get along with training and testing accuracy. Table 4.3 is their according false positive rate and false negative rate.

The result shows that support level 0.01 is clearly superior to others. That shows that frequent patterns are more important than infrequent patterns.

Instruction association type 1 and 2 outperform type 3. That is an interesting result which could serve to justify our approach over traditional N-Gram based approach in that N-Gram checks consecutive and ordered sequence only.

The performance of SVM is a little bit better than C4.5 in our top 10 list. The average testing accuracy of SVM in top 10 models is 0.911 compared with 0.906 of C4.5.

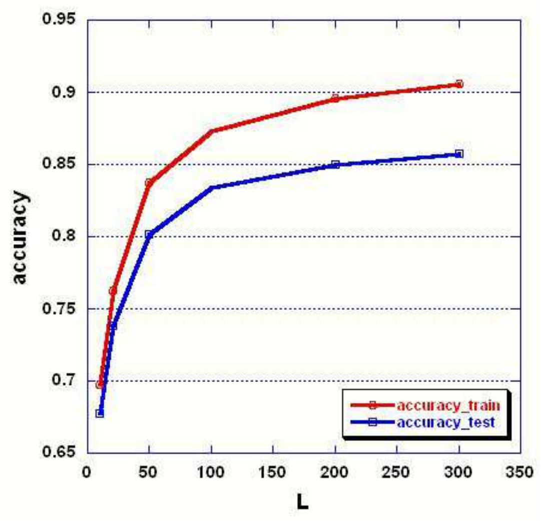


Figure 4.9: Effect of L

Figure 4.9 shows the effect of number of feature L vs average accuracy among all models. Generally, more features give more accurate result. However, the margin is small after L>100. Larger feature number implies longer processing time for monitor system. Usually 200-300 features are optimal.

The effect of number of instructions captured N is not quite clear yet. We further calculate average accuracy at different N in Figure 4.10. We see that generally accuracy increases when we use a large N. However, the difference becomes very small when N>2000, especially for testing dataset. That justifies that when we use the initial part of instructions, we can capture the behavior of the unknown executable. One interesting phenomenon is when N=2000, we get some really good result. Half of our top 10 models use 2000 instruction sequences. That means in some settings, first 2000 instructions already capture the character of the executable, further instructions might only give noise.

4.7.3 Outlier Analysis

We are interested to see if there are some common properties shared by those viruses that fail to be correctly classified by our model. We do expect that those misclassified samples

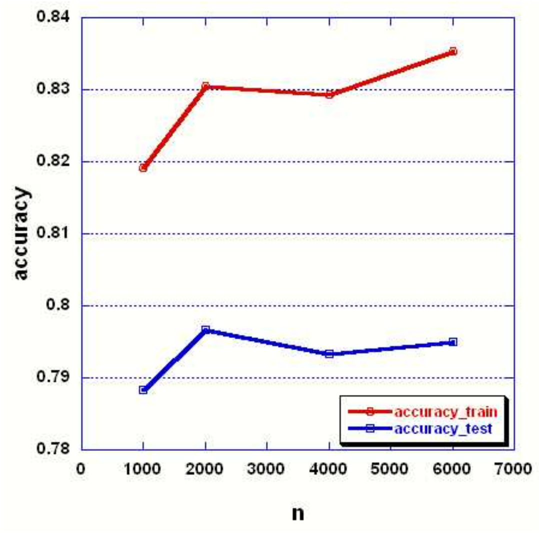


Figure 4.10: Effect of N

Table 4.2: Top 10 Configurations

Model#	**IA**	**S**	**L**	**C**	**N**	**Accuracy**
1	2	0.01	300	SVM	4000	0.954/0.919
2	2	0.01	300	SVM	2000	0.954/0.918
3	2	0.01	300	SVM	6000	0.955/0.914
4	2	0.005	300	C45	2000	0.937/0.910
5	2	0.01	200	SVM	6000	0.948/0.909
6	1	0.01	100	C45	6000	0.938/0.906
7	2	0.01	200	C45	4000	0.934/0.906
8	2	0.01	200	SVM	2000	0.949/0.904
9	2	0.01	300	SVM	2000	0.947/0.904
10	1	0.01	100	C45	2000	0.915/0.903

IA:	**Instruction Association type**
S:	**Support**
L:	**Number of features**
C:	**Classifier**
N:	**Number of dynamic instructions**
Accuracy:	**Accuracy in training set / testing set**

Table 4.3: FPR/FNR for top 10 models

Model#	FPR	FNR
1	0.074/0.096	0.025/0.068
2	0.074/0.118	0.027/0.056
3	0.072/0.114	0.027/0.066
4	0.094/0.126	0.041/0.065
5	0.085/0.126	0.028/0.063
6	0.120/0.131	0.015/0.067
7	0.102/0.131	0.038/0.068
8	0.084/0.130	0.028/0.071
9	0.089/0.121	0.028/0.076
10	0.145/0.151	0.038/0.055

FPR:	**False Positive Rate for training set / testing set**
FNR:	**False Negative Rate for training set / testing set**

are "hard" viruses, such as packed, polymorphic or metamorphic viruses. However, to our surprise, we cannot identify any shared patterns for those failed samples. Table 4.4 lists misclassified virus samples by our top models. For a concrete example, in Win32.HLLW virus family, we have 23 virus samples, 1 is misclassified. In Win32.HLLP virus family, we have 27 virus samples, 3 are misclassified. None of these virus samples are packed, polymorphic or metamorphic.

4.7.4 Decision Process

Table 4.5 shows typical processing time to process one unknown program in seconds on our test computer. The test computer has a typical configuration of recent personal computer with 1 AMD Turion 1.58G CPU and 1G memory. "Log" is the time to capture necessary instruction sequences. "Assembly" is the time to construct logic assembly and then get abstract assembly. "Feature" is the time to extract feature vector. "Model" is the time needed to apply the classifier. Table 4.6 is normalized execution time divided into different components.

Table 4.4: Misclassified Sample

Virus.Win32.Ruff.4859
Virus.Win32.Evol.c
Virus.Win32.Elkern.c
Virus.Win32.Cabanas.Release
Virus.Win32.Henky.504
Virus.Win32.HLLW.Myset.b
Virus.Win32.Gaybar
Virus.Win32.HLLO.Rozak.a
Virus.Win32.Resur.b
Virus.Win32.ZMist
Virus.Win32.Chiton.p
Virus.Win32.Evyl.d
Virus.Win32.Delikon
Virus.Win32.HLLP.Xinfect.i
Virus.Win32.HLLP.Vedex.b
Virus.Win32.InvictusDLL.b
Virus.Win32.HLLP.Gezad
Virus.Win32.Kaze.3228
Virus.Win32.Seppuku.1606
Virus.Win32.Mystery.2560
Virus.Win32.Redemption.a
Virus.Win32.Ditto.1539
Virus.Win32.Refer.2939
Virus.Win32.Lash.b

Some programs could take much longer time to collect enough instruction sequences because of some long system calls or some function requires user interaction. We have a hard limit on “log” time which is 3s.

Other time components are relatively stable. SVM model takes significantly more time than C4.5.

The overall processing time is usually under 3 seconds.

We also tested the capacity of our malicious behavior protection system. We run all 267 malicious programs from our dataset inside our monitor system. None of them present

Table 4.5: Typical Runtime for Monitor System

Model#	Total	Log	Assembly	Feature	Model
1	1.79	0.25	0.06	0.37	1.12
2	1.66	0.22	0.03	0.13	1.28
3	2.86	0.34	0.08	1.24	1.20
4	0.34	0.15	0.03	0.14	0.02
5	2.29	0.36	0.09	0.85	1.02
6	0.45	0.35	0.08	0.02	0.01
7	0.56	0.27	0.05	0.23	0.01
8	1.29	0.18	0.03	0.10	0.99
9	1.44	0.27	0.03	0.13	1.01
10	0.23	0.19	0.07	0.01	0.01

Table 4.6: Normalized Runtime for Monitor System

Model#	NLog	NAssembly	NFeature	NModel
1	0.06	0.015	0.031	0.37
2	0.11	0.015	0.021	0.42
3	0.06	0.013	0.069	0.40
4	0.08	0.015	0.023	/
5	0.06	0.015	0.071	0.51
6	0.06	0.013	0.003	/
7	0.07	0.013	0.029	/
8	0.09	0.015	0.025	0.50
9	0.14	0.015	0.022	0.50

NLog	=	**Log Time / 1000 Instructions**
NAssembly	=	**Assembly Time / 1000 Instructions**
NFeature	=	**Feature Selection Time / (100 Features * 1000 Instructions)**
NModel	=	**SVM Classification Time / 100 Features**

obvious malicious behavior. Although our protection scheme is not complete as discussed in chapter 3, most viruses are well guarded by our system.

4.7.5 Ensemble Model

Ensemble model is a meta-model which applies several models simultaneously, and takes the majority decision as the decision of the ensemble model. We use top 3, 5, 10 models to build our ensemble model. Table 4.7 shows our result.

Table 4.7: Performance of Ensemble Models

Model Numbers	**Accuracy**	**FPR**	**FNR**
3	0.943/0.908	0.093/0.151	0.031/0.045
5	0.943/0.908	0.089/0.137	0.035/0.058
10	0.947/0.917	0.088/0.161	0.028/0.016

We do not see improvement for the result of ensemble model. Further, ensemble model takes much more time to make decision on the monitor system. So we decide not to use ensemble models in our system.

4.8 Comparison Study

To evaluate the performance of dynamic instruction sequences approach, we experiment on a serial of approaches to find out the effectiveness of our dynamic instruction sequences approach.

4.8.1 Simple Heuristics Approach

Simple heuristic features refer to the feature set extracted from Win32 PE head or strings inside executable body.

[6] illustrates many heuristics to detect structural anomaly of Win32 viruses. In our experiment, we use the following features:

(i) Entry point locates in the last section

(ii) Suspicious section name

(iii) Suspicious section characteristics

(iv) Inconsistent size calculation

(v) Suspicious imports by ordinal

(vi) Critical function imports

(vii) Corrupt relocation table

(viii) Occupation rate of last block in sections (if over 10%)

In our experiment, we uses the presence of each of such feature as our boolean feature. Since we only have 8 features, we do not perform feature selection. Like dynamic instruction approach, we use 5-folds cross validation approach to get our result and report accuracy, false positive and false negative on both training and testing dataset.

4.8.2 NGram Approach

N-Gram refers to consecutive binary sequences of fixed size inside binary code.

In our experiment, we repeat the experiment in [34] on our dataset, and compare the result of N-Gram with our approach.

We select top k features as usual. For each feature, we will find out the presence of that feature in both training and testing dataset. We also use C4.5 decision tree and support vector machine as our classifier. We report accuracy, false positive rate and false negative rate as well.

4.8.3 Static Assembly Approach

Static approaches are based on the features derived from static analysis. Some approaches are based on assembly code derived by disassembling the executable binary [39]. Other approaches are based on higher level structure based on assembly code such as control flow graph (CFG) [37]. However, disassembling binary code itself is a hard problem and no general solution has been discovered yet [45].

The reason for the failures in disassembling can be classified into two folds.

(i) Encrypted and polymorphic virus can not be disassembled by conventional disassembler. Each of these viruses encrypts virus body in binary file and the encrypted code is not directly runnable. After Windows loader loads the virus into memory, execution starts with a special part of code called decryptor. Decryptor is usually small and not encrypted. The decryptor code will modify the memory image and restore virus body to runnable code. Usually encryption key changes every time so that virus body does not remain the same and thus makes signature based detection ineffective. The difference between encrypted virus and polymorphic virus is that encrypted virus uses a constant decryptor and polymorphic virus mutates the descriptor. Disassembler analyzes the binary code statically, so it can not disassemble the virus body except for the loader

(ii) Current disassembler uses linear sweep or recursive traversal techniques which are inherently limited

Linear sweep disassembles binary codes into instructions byte by byte. An instruction may consist up to 13 bytes. Linear sweep assumes the code is consecutive, there is no hole in the between even if the previous instruction is a control flow transfer instruction. In reality, especially in malicious code, there could be a hole after some control flow transfer instructions, thus defeat linear sweep.

Recursive traversal try to find the destination of each transfer control instruction at first, then perform linear sweep for each block. Even if there is a hole after the control flow transfer instruction, the assembler still works because it does not assume that the next instruction is the starting point of the next block. However, not all start point can be easily extracted. Some control flow transfer instructions do not use an address constant as parameter, instead, they use value inside a variable. The value inside a variable cannot be derived statically. Current disassembler use some heuristics to guess the destination address of the control flow transfer instruction. However, in many cases, this approach fails. Figure 4.11 shows an

```
00401001                   call [loc_401121+1]
loc_401121:
00 8B 34 24 8B EE          add [ebx-1174DBCCh], cl
81 ED 06 10 40 00          sub ebp, offset loc_401006
8D BD DF 17 40 00          lea edi, dword_4017DF[ebp]
```

Figure 4.11: Wrong Assembly Code

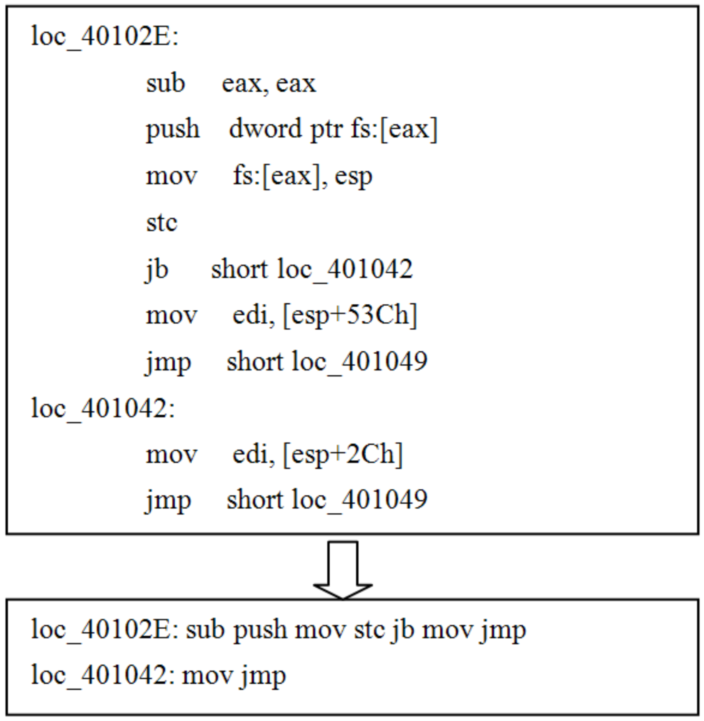
```
00401001                   call loc_401121+1
loc_401122:
8B 34 24                   mov esi, dw ptr SS:[ESP]
8B EE                      mov ebp, esi
81 ED 06 10 40 00          sub ebp, 00401006
```

Figure 4.12: Correct Assembly Code

example of such case IDAPro fail to disassemble. The correct form should be figure 4.12. The first instruction is a "call" instruction and its destination is "loc_401121+1", which refers to "00401122". Ideally, disassembler starts disassembling a new function from "00401122". However, IDAPro is not clever enough to do the calculation and only makes a simple guess that "00401121" is the destination address. The content in "00401121" is "00", which is the opcode of "add" instruction. IDAPro interpret it as an "add" instruction and treat the bytes following it as its operands. Actually, the following byte "8B" should be interpreted as opcode. Now everything is messed up. In these cases, further disassembling can be achieved manually in IDAPro. In the example, we can manually tell IDAPro that "00401122" is the function start address. However, manual disassembling requires advanced skills and is time consuming.

Some other approaches [45, 47] have also been proposed to alleviate this problem. However, none of current approaches solve the problem completely.

Although our feature is based on assembly code, the disassembility is not the key to our approach. If an executable cannot be fully disassembled, we extract fewer instruction



```
loc_40102E:
        sub    eax, eax
        push   dword ptr fs:[eax]
        mov    fs:[eax], esp
        stc
        jb     short loc_401042
        mov    edi, [esp+53Ch]
        jmp    short loc_401049
loc_401042:
        mov    edi, [esp+2Ch]
        jmp    short loc_401049
```

```
loc_40102E: sub push mov stc jb mov jmp
loc_401042: mov jmp
```

Figure 4.13: From Assembly to Abstract Assembly

sequences or no instruction sequences at all. Our process will still work even in these cases. The disassembility becomes a heuristics in our approach.

We first disassemble the binary code into assembly code. The assembly code breaks into natural blocks. The start point of each block is associated with a disassembler-generated label. Usually it is a destination of a control flow instruction elsewhere in the program. For each block, we generate a line with a sequence of instruction operation codes inside the block as shown in Figure 4.13. The result is similar to the abstract assembly we get in dynamic instruction sequence approach.

If an executable cannot be fully disassembled, we extract fewer sequences or no sequences at all. Our process will still work even in these cases.

Once we get the abstract assembly, we use the same approach to extract features, select top features and then collect the accuracy, false positive rate and false negative rate for both training dataset and testing dataset.

Table 4.8: Performance of Simple Heuristic Features

	Accuracy	**FPR**	**FNR**
C4.5	0.900/0.897	0.026/0.234	0.028/0.248
SVM	0.906/0.853	0.022/0.225	0.017/0.233

Table 4.9: Performance of NGram (C4.5)

N	**Accuracy**	**FPR**	**FNR**
8	0.986/0.882	0.009/0.111	0.021/0.127
10	0.981/0.896	0.007/0.082	0.035/0.135
12	0.978/0.909	0.005/0.065	0.046/0.127
14	0.981/0.917	0.009/0.054	0.003/0.123
16	0.973/0.909	0.007/0.054	0.055/0.142
18	0.976/0.906	0.009/0.084	0.044/0.108

4.8.4 Comparison Result

Table 4.8 lists the experimental result for simple heuristic features. C4.5 decision tree performs better than support vector machine in this experiment. The accuracy over testing dataset is 89.7%.

Table 4.9 and table 4.10 shows the experimental result of NGram features using C4.5 and SVM classifier respectively. The best performance of NGram features is achieved when N equals to 14, using c4.5 classifier. This is in par with dynamic instruction sequences approach. That shows NGram approach is still effective to detect current virus. However, as we discuss before, NGram approach does not directly use instruction information of a program. So it may not be able to capture the most important characteristic of a malicious software.

Figure 4.14 and figure 4.15 shows the training accuracy and testing accuracy for both C4.5 decision tree and support vector machine classifiers. From the diagram, we can see testing accuracy of c4.5 decision tree improves when N increases from 8 to 14. After that,

Table 4.10: Performance of NGram (SVM)

N	**Accuracy**	**FPR**	**FNR**
8	0.958/0.896	0.019/0.065	0.073/0.158
10	0.957/0.904	0.018/0.060	0.078/0.146
12	0.961/0.901	0.012/0.062	0.076/0.150
14	0.961/0.904	0.013/0.051	0.075/0.157
16	0.954/0.899	0.018/0.051	0.085/0.169
18	0.957/0.904	0.017/0.052	0.077/0.157

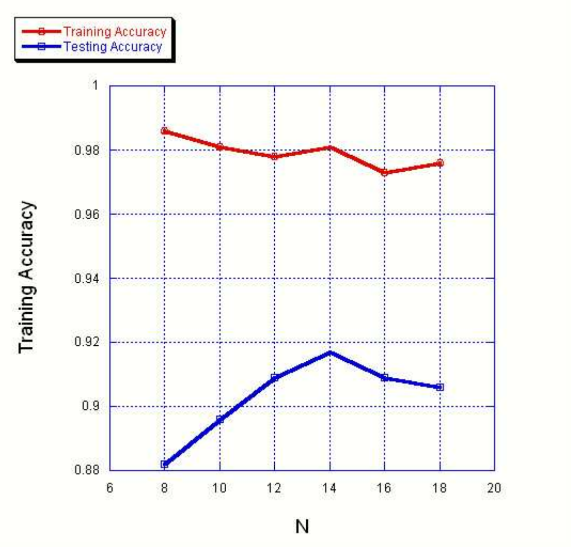


Figure 4.14: Experimental Result for NGram (C4.5)

the testing accuracy begins to drop. For support vector machine, testing accuracy increases first and then fluctuates when N ¿ 10. A large N does not imply a better performance.

Table 4.11 shows top 10 models using static assembly features. The best models is achieved when support level is 0.005, feature size is 300 and instruction association type is 2. The accuracy over testing dataset for this model is 87.5%. This is much worse than dynamic instruction sequences features. Table 4.12 shows the false positive rate and false negative rate for top 10 models based on static assembly features. Most of them have a higher false negative rate than false positive rate. As we have already discussed in 4.7, low false negative

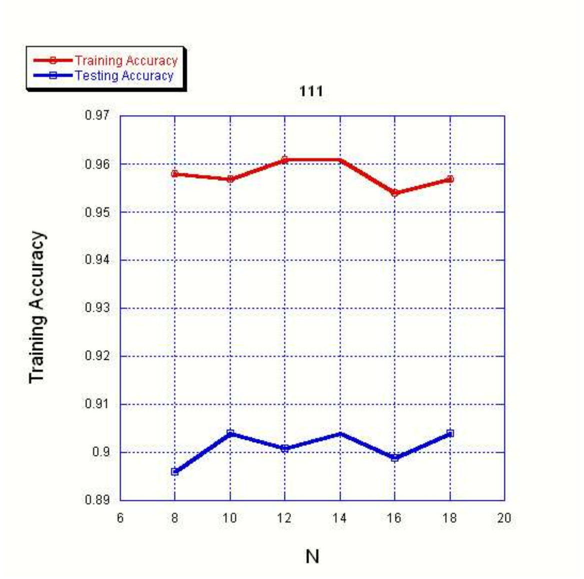


Figure 4.15: Experimental Result for NGram (SVM)

Table 4.11: Top 10 static models

Model#	IA	S	L	C	Accuracy
1	2	0.005	300	SVM	0.951/0.875
2	1	0.01	300	SVM	0.961/0.869
3	3	0.003	76	C45	0.895/0.863
4	2	0.01	300	SVM	0.961/0.861
5	2	0.01	200	SVM	0.959/0.861
6	1	0.01	200	SVM	0.942/0.853
7	2	0.003	300	SVM	0.936/0.851
8	2	0.005	100	SVM	0.942/0.849
9	1	0.003	200	C45	0.951/0.849
10	2	0.005	200	SVM	0.950/0.847

IA:	**Instruction Association type**
S:	**Support**
L:	**Number of features**
C:	**Classifier**
N:	**Number of dynamic instructions**
Accuracy:	**Accuracy in training set / testing set**

rate is more important than low false positive rate in real practice. It is not the case in static assembly features approach.

Table 4.12: FPR/FNR for top 10 static models

Model#	FPR	FNR
1	0.022/0.078	0.091/0.209
2	0.014/0.075	0.077/0.229
3	0.170/0.206	0.036/0.066
4	0.020/0.096	0.067/0.207
5	0.019/0.104	0.076/0.194
6	0.040/0.123	0.083/0.184
7	0.045/0.107	0.094/0.216
8	0.041/0.127	0.083/0.187
9	0.058/0.139	0.040/0.175
10	0.025/0.118	0.088/0.210

FPR:	**False Positive Rate for training set / testing set**
FNR:	**False Negative Rate for training set / testing set**

For all the feature sets we are comparing, simple heuristics approach and static assembly approaches performs much worse than dynamic instruction sequences features. NGram and dynamic instruction sequences feature achieve similar detection rate.

False negative rate for simple heuristics features, NGram features and static assembly features are higher than false positive rate, which is not desirable for a malicious software detection system. Dynamic instruction sequences approach instead, has a lower false negative rate.

Dynamic instruction sequences utilize the information of instructions of a program. Instructions capture the most important characteristic of a program. Simple heuristic feature checks program header only. NGram feature mostly relies on program header information and string information, which is easier to be evaded by advanced malicious software.

Based on the above analysis, we conclude that dynamic instruction sequences features is more desirable than other three approaches we are comparing.

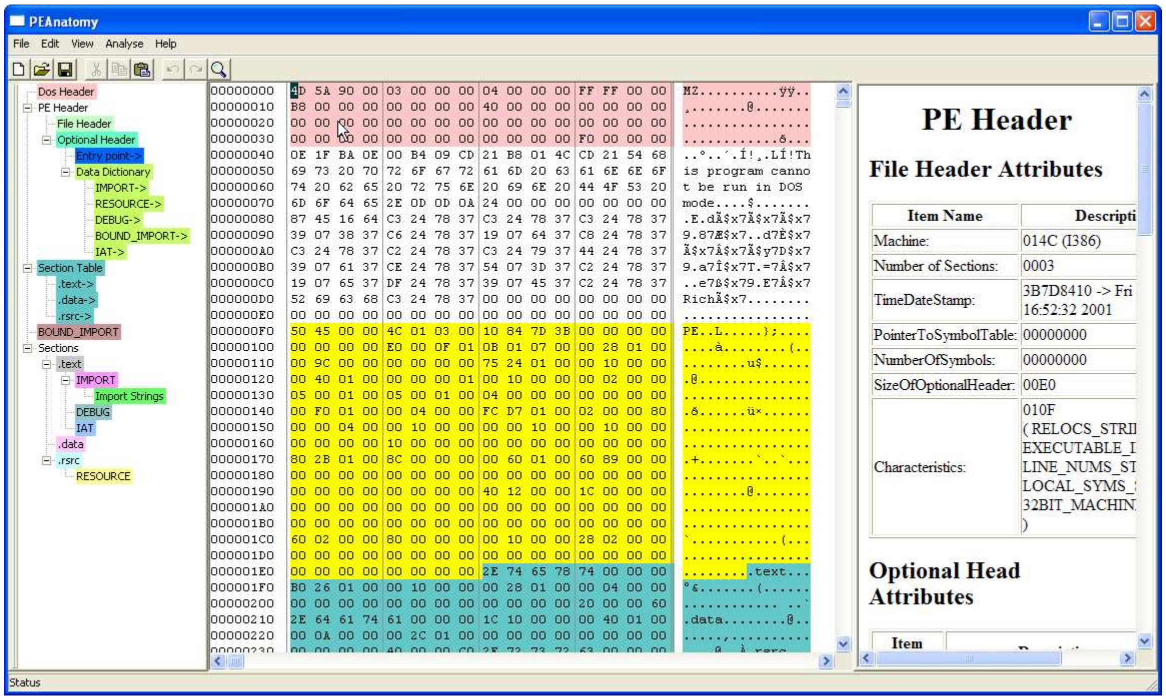
4.9 Malicious Software Analysis Toolkit

our malici analysis is to analyze the executable code without actually executing the code. Popular static analyzing approaches include PE head analysis, assembly analysis and higher level analysis. Existing static analyzing tool include PE head viewer and disassembler.

A set of tools [92, 93, 94] can extract PE head information [25, 26] and present it to the user. These tools print out all the information of the PE head in a list form on screen which include PE head, data dictionary, section table, import address table (IAT), import table and export table. However, these tools only list all the information and do not try to show how these information locates inside an executable binary. In virus detection, the location of these information is very important. For example, Entry point is a very important heuristics to detect virus. Many virus codes will modify the original entry point of the victim. And the fact that entry point points to the last section is suspicious. We need a way to visualize the entry point within the executable and sections.

Several dissemblers [41, 92] are available for static analysis. All these dissemblers try to disassemble the binary code globally. The techniques they are using are based either on linear sweep or recursive traversal. However, neither solution is perfect. In many cases, we cannot get complete assembly code automatically. IDA Pro provide manual control over the disassembly code. You can manually adjust the assembly code if observing anything wrong or incomplete. However, manual disassembling requires experience and quite time consuming.

We try to build a different type of static analysis tool and focus on the visualization of the static analysis process. We call this tool PEAnatomy. The center piece of our analyzer is the binary representation of the executable to be analyzing. To visualize the executable, we mark different parts of the executable with different colors. The head part of an executable include Dos head, PE head (include data dictionary) and section table. After that, we have



```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  87 45 16 64 C3 24 78 37 C3 24 78 37 C3 24 78 37  .E.dÃ$x7Ã$x7Ã$x7
00000090  39 07 38 37 C6 24 78 37 19 07 64 37 C8 24 78 37  9.87Æ$x7..d7È$x7
000000A0  C3 24 78 37 C2 24 78 37 C3 24 79 37 44 24 78 37  Ã$x7Â$x7Ã$y7D$x7
000000B0  39 07 61 37 CE 24 78 37 54 07 3D 37 C2 24 78 37  9.a7Î$x7T.=7Â$x7
000000C0  19 07 65 37 DF 24 78 37 39 07 45 37 C2 24 78 37  ..e7ß$x79.E7Â$x7
000000D0  52 69 63 68 C3 24 78 37 00 00 00 00 00 00 00 00  RichÃ$x7........
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  50 45 00 00 4C 01 03 00 10 84 7D 3B 00 00 00 00  PE..L.....};....
00000100  00 00 00 00 E0 00 0F 01 0B 01 07 00 00 28 01 00  ....à........(..
00000110  00 9C 00 00 00 00 00 00 75 24 01 00 00 10 00 00  ........u$......
00000120  00 40 01 00 00 00 00 01 00 10 00 00 00 02 00 00  .@..............
00000130  05 00 01 00 05 00 01 00 04 00 00 00 00 00 00 00  ................
00000140  00 F0 01 00 00 04 00 00 FC D7 01 00 02 00 00 80  .ð......ü×......
00000150  00 00 04 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
00000160  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  80 2B 01 00 8C 00 00 00 00 60 01 00 60 89 00 00  .+.......`..`...
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 40 12 00 00 1C 00 00 00  ........@.......
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001C0  60 02 00 00 80 00 00 00 00 10 00 00 28 02 00 00  `...........(...
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001E0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001F0  B0 26 01 00 00 10 00 00 00 28 01 00 00 04 00 00  °&.......(......
00000200  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
00000210  2E 64 61 74 61 00 00 00 1C 10 00 00 00 40 01 00  .data........@..
00000220  00 0A 00 00 00 2C 01 00 00 00 00 00 00 00 00 00  .....,..........
```

PE Header

File Header Attributes

Item Name	Descripti
Machine:	014C (I386)
Number of Sections:	0003
TimeDateStamp:	3B7D8410 -> Fri 16:52:32 2001
PointerToSymbolTable:	00000000
NumberOfSymbols:	00000000
SizeOfOptionalHeader:	00E0
Characteristics:	010F (RELOCS_STRI EXECUTABLE_I LINE_NUMS_ST LOCAL_SYMS_ 32BIT_MACHIN)

Optional Head Attributes

Item	

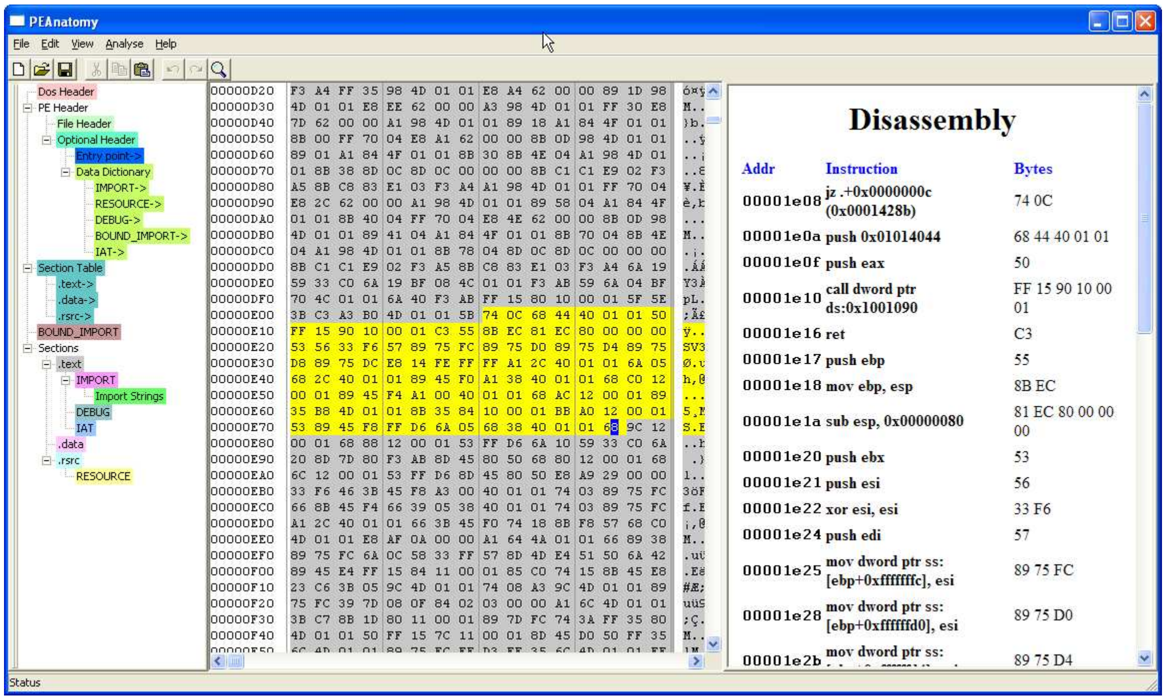
Figure 4.16: PE Anatomy Snapshot 1

a couple of sections. The entry point, import table, export table and IAT are located in regular sections and are clearly marked as well.

For disassembling, we provide a visualized linear sweep disassembler. The difference between our disassembler and other disassembler is that our disassembler is not focus on the entire executable. Instead, using our visualized GUI, users can find the binary section they are interested in, and invoke the disassembler easily with a click and drag.

Figure 4.16 and figure 4.17 show two snapshots of PEAnatomy. Figure 4.16 shows a binary code and the right pane shows the information list of selected component. Here the selected component is PE head. PEAnatomy is able to show information of different structures within the binary code, include section table, import table, export table, IAT, etc. Figure 4.17 shows the disassembling feature of PEAnatomy. By selecting a section of binary code, we can view the according assembly code on the fly in the right pane.

Some other functions current under development are:



```
00000D20  F3 A4 FF 35 98 4D 01 01 E8 A4 62 00 00 89 1D 98
00000D30  4D 01 01 E8 EE 62 00 00 A3 98 4D 01 01 FF 30 E8
00000D40  7D 62 00 00 A1 98 4D 01 01 89 18 A1 84 4F 01 01
00000D50  8B 00 FF 70 04 E8 A1 62 00 00 8B 0D 98 4D 01 01
00000D60  89 01 A1 84 4F 01 01 8B 30 8B 4E 04 A1 98 4D 01
00000D70  01 8B 38 8D 0C 8D 0C 00 00 00 8B C1 C1 E9 02 F3
00000D80  A5 8B C8 83 E1 03 F3 A4 A1 98 4D 01 01 FF 70 04
00000D90  E8 2C 62 00 00 A1 98 4D 01 01 89 58 04 A1 84 4F
00000DA0  01 01 8B 40 04 FF 70 04 E8 4E 62 00 00 8B 0D 98
00000DB0  4D 01 01 89 41 04 A1 84 4F 01 01 8B 70 04 8B 4E
00000DC0  04 A1 98 4D 01 01 8B 78 04 8D 0C 8D 0C 00 00 00
00000DD0  8B C1 C1 E9 02 F3 A5 8B C8 83 E1 03 F3 A4 6A 19
00000DE0  59 33 C0 6A 19 BF 08 4C 01 01 F3 AB 59 6A 04 BF
00000DF0  70 4C 01 01 6A 40 F3 AB FF 15 80 10 00 01 5F 5E
00000E00  3B C3 A3 B0 4D 01 01 5B 74 0C 68 44 40 01 01 50
00000E10  FF 15 90 10 00 01 C3 55 8B EC 81 EC 80 00 00 00
00000E20  53 56 33 F6 57 89 75 FC 89 75 D0 89 75 D4 89 75
00000E30  D8 89 75 DC E8 14 FE FF FF A1 2C 40 01 01 6A 05
00000E40  68 2C 40 01 01 89 45 F0 A1 38 40 01 01 68 C0 12
00000E50  00 01 89 45 F4 A1 00 40 01 01 68 AC 12 00 01 89
00000E60  35 B8 4D 01 01 8B 35 84 10 00 01 BB A0 12 00 01
00000E70  53 89 45 F8 FF D6 6A 05 68 38 40 01 01 68 9C 12
00000E80  00 01 68 88 12 00 01 53 FF D6 6A 10 59 33 C0 6A
00000E90  20 8D 7D 80 F3 AB 8D 45 80 50 68 80 12 00 01 68
00000EA0  6C 12 00 01 53 FF D6 8D 45 80 50 E8 A9 29 00 00
00000EB0  33 F6 46 3B 45 F8 A3 00 40 01 01 74 03 89 75 FC
00000EC0  66 8B 45 F4 66 39 05 38 40 01 01 74 03 89 75 FC
00000ED0  A1 2C 40 01 01 66 3B 45 F0 74 18 8B F8 57 68 C0
00000EE0  4D 01 01 E8 AF 0A 00 00 A1 64 4A 01 01 66 89 38
00000EF0  89 75 FC 6A 0C 58 33 FF 57 8D 4D E4 51 50 6A 42
00000F00  89 45 E4 FF 15 84 11 00 01 85 C0 74 15 8B 45 E8
00000F10  23 C6 3B 05 9C 4D 01 01 74 08 A3 9C 4D 01 01 89
00000F20  75 FC 39 7D 08 0F 84 02 03 00 00 A1 6C 4D 01 01
00000F30  3B C7 8B 1D 80 11 00 01 89 7D FC 74 3A FF 35 80
00000F40  4D 01 01 50 FF 15 7C 11 00 01 8D 45 D0 50 FF 35
```

Disassembly

Addr	Instruction	Bytes
00001e08	jz .+0x0000000c (0x0001428b)	74 0C
00001e0a	push 0x01014044	68 44 40 01 01
00001e0f	push eax	50
00001e10	call dword ptr ds:0x1001090	FF 15 90 10 00 01
00001e16	ret	C3
00001e17	push ebp	55
00001e18	mov ebp, esp	8B EC
00001e1a	sub esp, 0x00000080	81 EC 80 00 00 00
00001e20	push ebx	53
00001e21	push esi	56
00001e22	xor esi, esi	33 F6
00001e24	push edi	57
00001e25	mov dword ptr ss: [ebp+0xfffffffc], esi	89 75 FC
00001e28	mov dword ptr ss: [ebp+0xffffffd0], esi	89 75 D0
00001e2b	mov dword ptr ss:	89 75 D4

Figure 4.17: PE Anatomy Snapshot 2

(i) Read binary code from memory in addition to binary file. Some worms do not have a file trace and we need to analyze them from the bytes inside memory

(ii) Mark address constant. Address constants are double word inside the binary and the value of the binary is within the range of address space of the executable. Address constant is likely to be a destination of control flow transfer instruction. [45] uses address constants to construct its disassembler. Marking address constant inside a binary file helps us to understand the code

(iii) Consistency check. Check the consistency of the PE head listed in [6]. Mark any inconsistency

(iv) Byte profile

(v) Instruction sequence check. We integrate our dynamic instruction sequence approach into PEAnatomy. We can use the result of dynamic instruction sequence check as a criteria

CHAPTER 5: DETECTING MALICIOUS SOFTWARE FAMILY USING STRUCTURAL SYSTEM CALL DIAGRAM

5.1 Introduction

In this section, we present a utilization of dynamic system call trace on a family of malicious software. Captured system call contains information which determines some function of the malicious software. Using our developed visualization tool, we are able to visualize the system call trace of an executable in a way very similar to control flow graph and hence observe the similarities and differences of a family of malicious software. As a concrete application, we construct smart signature to detect a family of malicious software using system call signature. By using our visualization tool, we can manually derive the signature to detect malicious code family. Also, we design a way to automatically generate system call signature of the malicious software family. We do not require an exact match for a system call signature. We use a profile hidden Markov model (PHMM) to describe a system call signature and score it against unknown software to determine the nature of the unknown executable.

There are two categories of malicious software analysis approaches, static approaches and dynamic approaches.

Static approaches check executable binaries or assembly codes without actually executing the unknown program. Based on assembly code, some advanced techniques have been tried to construct higher level structure such as control flow graph (CFG) [37] and template [36] to help security expert to analyze the malicious software.

Dynamic approaches require actual execution of the software being analyzed. Current techniques include debugger [73] and system call capturer [50]. However, no analysis has been proposed on system call trace only.

Analyzing system call trace directly is not easy. System call trace contains repetitions and does not break into blocks. System call trace contains important information about the structure of the executable being analyzed. We use this information to construct a graph similar to the control flow graph.

There are several components we capture with system call trace.

(i) Return address and stack trace

(ii) Parameters and return value

(iii) Timestamp information

To reveal the structure of a program through system call traces, stack information of system calls is the most important. First, return address, which is located on top of stack, reveals the point of invocation of a specific system call. Second, more return addresses on stack reveal more functions and relations between these functions. Repetitions in the system call trace are also very important. They reveal sequential, loop and branch structure among different system calls. The order of system calls inside a block tells which system call comes first and which comes next. It contains important information as well.

We build a visualization tool to show the control flow graph like structure of the system calls. We call this a structural system call diagram.

As an application, we view and compare system call blocks between executables come from the same malicious software family. We find that breaking down system call trace into a structure has advantages to understand the similarities and differences between different executables.

By utilizing well defined boundaries among system calls, we use a whole block of system calls as a signature to detect other malicious software of the same family.

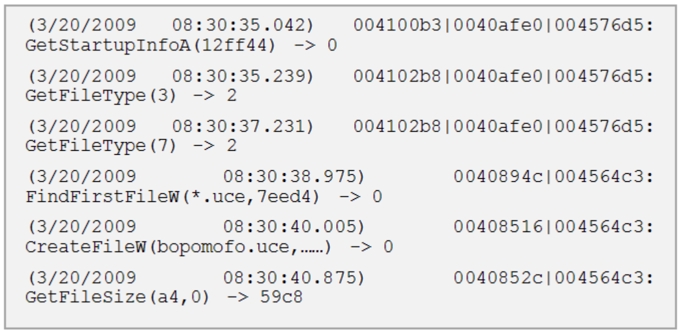
We use two approaches to derive system call signature: (1) manual extraction with the help of our visualizer and (2) automatic extraction by using our automatic system call signature extraction tool without user intervention.

When malicious software evolves, it is not necessary that all system call sequences inside a block remain the same. An exact match is too strict. In this paper, we propose a concept smart signature which allows insertion, deletion and mutation of one or some positions inside system call signature.

We get a system call block as a signature of the malicious software family through a manual or automatic process. We build a profile hidden Markov model (PHMM) [95] for the signature we derive and use it as a smart signature. For an unknown program, we feed each block into PHMM and get a score. The higher the score, the more similar this block is to the system call signature. If the unknown program contains a block very similar to system call signature, we determine it belong to the malicious software family.

5.2 Capturing Dynamic System Call

We choose to use Microsoft Detours library [72] to implement our dynamic system call capturing system. Detours is a library freely available from Microsoft Research. Detours hooks arbitrary user code onto Win32 API call. It provides API call to modify the first several instructions of the target function with a "jmp" instruction. "jmp" diverts the execution flow of Win32 API to the user supplied code. However, the replaced instructions of the target function are not lost. It is saved by Detours to a newly appended section of the executable code. Inside the user supplied codes, the user can optionally invoke the original version of the system call through a trampoline function. In current Detours implementation, every application using Detours should include "detoured.dll", which is easy to be detected by application. However, "detoured.dll" file is a marker that guides Microsoft technical support



```
(3/20/2009   08:30:35.042)   004100b3|0040afe0|004576d5:
GetStartupInfoA(12ff44)  -> 0
(3/20/2009   08:30:35.239)   004102b8|0040afe0|004576d5:
GetFileType(3)  -> 2
(3/20/2009   08:30:37.231)   004102b8|0040afe0|004576d5:
GetFileType(7)  -> 2
(3/20/2009        08:30:38.975)        0040894c|004564c3:
FindFirstFileW(*.uce,7eed4)  -> 0
(3/20/2009        08:30:40.005)        00408516|004564c3:
CreateFileW(bopomofo.uce,……)  -> 0
(3/20/2009        08:30:40.875)        0040852c|004564c3:
GetFileSize(a4,0)  -> 59c8
```

Figure 5.1: System Call Trace

personnel and tools by helping them quickly determine that a process has been altered by the Detours package [72]. It is not a limitation inherent to Detours.

In our implementation, we capture the following information along with the name of system call:

(i) Return address of the system call and the stack trace information

(ii) Inputs and outputs of system call

(iii) Timestamp of the system call

We capture the system call information before and after the real system call invocation. Before the invocation, we capture the program stack at the moment of system call invocation, input parameter and time stamp. After the invocation, we capture return value and output parameters. Figure 5.1 shows one system call trace we have captured. Every line in figure 5.1 represents a captured system call. It starts with a time stamp, followed by call stacks, and then the name of system and its parameters.

Stack information refers to the active stack frames of the running program. In figure 5.2.b, we show stack layout for a typical program. The program itself is listed in figure 5.2.a. Each time we invoke a function, we first push the return address, which is the address of next instruction in the calling function, onto the stack. Next we save EBP, which is the frame point of previous function, in the stack. EBP will be used as frame point of the new function.

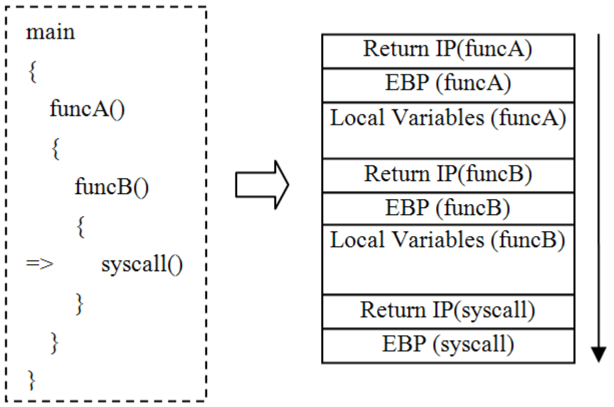


Figure 5.2: Stack Structure

EBP remains constant during the execution of function body. From the current EBP, we can infer position of local variable, saved return IP and saved EBP on stack. Especially, from the current EBP, we can get position of EBP of the calling function. Thus, EBP of all calling functions form a chain. In figure 5.2, we show the state of stack right after we invoke "syscall" in the program. From the current EBP, we can get EBP of all invoking functions, which are funB, funA and main. Because return IP address of the function is right above saved EBP, we can get return IP address of the function easily. We use the return IP address as the identification of a function.

However, not all programs are well-behaved especially for some malicious software. Sometimes they do not put function base pointer onto the stack. If this happens, our program cannot extract correct stack trace information. In this case, we only capture the stack information up to the point where we can.

5.3 Structural System Call Diagram

The goal to construct a structural system call diagram is to visualize the system call trace in a natural way. System call trace contains the information about dynamic system calls of the program. However, system call trace contains repetitions. In our statistics, more than 80% of the contents of the system call trace are simple repetitions. Also, system call traces are simple chronicle logs and do not break into blocks. This sequential form of system call is not intuitive for human to read.

Our target diagram is a control flow graph like visual structure. To build a structural system call diagram, we proceed as follows:

(i) Break sequential system call trace into natural blocks

(ii) Remove repetition

(iii) Connect different blocks with edges which represent the control flow

(iv) For each block, list all system call information

(v) For each edge, calculate the number of times that execution follow this path

5.3.1 Generating Blocks

First, we describe the algorithm to break sequential system call trace into natural blocks. We divide this process into two steps. First, we divide the system calls into functions. Then we further break each function into several blocks.

5.3.1.1 Identifying Functions

In section 5.2, we have discussed that calling function of a system call can be identified by stack trace information. We can get the calling chain from that system call all the way up to the main function. Our goal here is to identify system calls belonging to the same function from system call trace.

In figure 5.3.a, we demonstrate two simple functions, funcA and FuncB. FuncB invokes FuncA and FunA invoke three system calls respectively. If we run FunB, we get a stack trace of three system calls in figure 5.3.b. Although the immediate return IP of the system

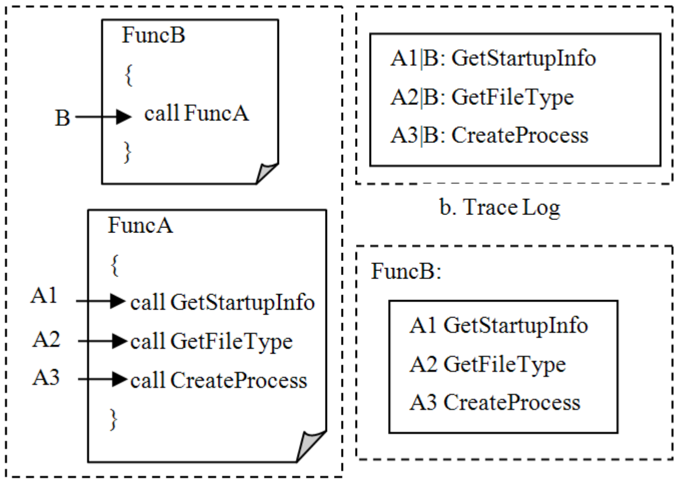


Figure 5.3: Identifying Functions

calls are different (A1, A2, A3 in figure 5.3.a), the second level return IP are the same (B in figure 5.3.a). From this example, we can infer that if the second level return IPs of two system calls are the same, then they belong to the same function (figure 5.3.c). We use return IP to identify that function. However, this is not absolutely true. We will discuss exceptions to this rule later.

On the other hand, return IP is a dynamic concept which is not perfect to describe a function. Even if the return IPs are different, they could be different invocations of the same function. Figure 4 shows an example. In this example, we identify two different functions, B1 and B2. However, FuncA and FuncB are merely two different invocations of the same function, FuncA. We use a heuristics to address this. If the address spaces of two different function identified by return ID overlap, then the two functions are the same. This heuristics cannot address all situations though. We only observed a partial address space of the function inside system call trace. Even if observed address spaces of two functions do not overlap, they could be the same function as well.

In sum, here are the criteria we are using to organize system calls into functions:

(i) If two system calls have the same second level return IP, we consider these two system calls are inside the same function

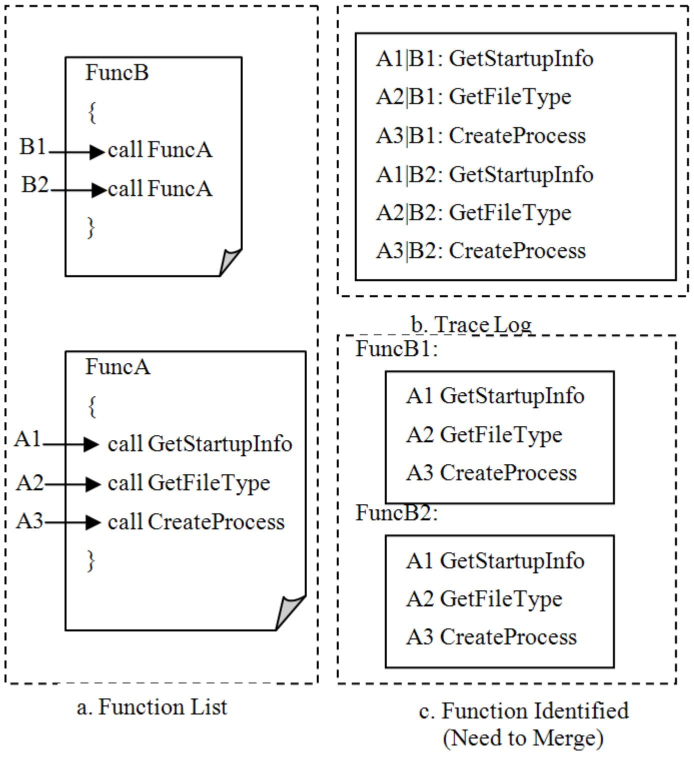


Figure 5.4: Merge Functions

(ii) For any two different function blocks derived by 1, if the address spaces overlap, then we merge these two function blocks

Now, let us discuss exceptions to rule 1. Some programs use function point variables in their program. Figure 5.5 shows an example. FuncA1 and FuncA2 are two different functions. However, by using a function point variable, we invoke FuncA1 and FuncA2 at the same location. In this case, even second level return IP are the same, we are invoking different functions.

There is no simple solution to solve these problems. System call trace only contains partial information of the program structure. We cannot get everything from it.

In addition, system call trace is not perfect. As we have already discussed before, sometimes we cannot get correct program stack. This adds extra complexity and incompleteness to the problem.

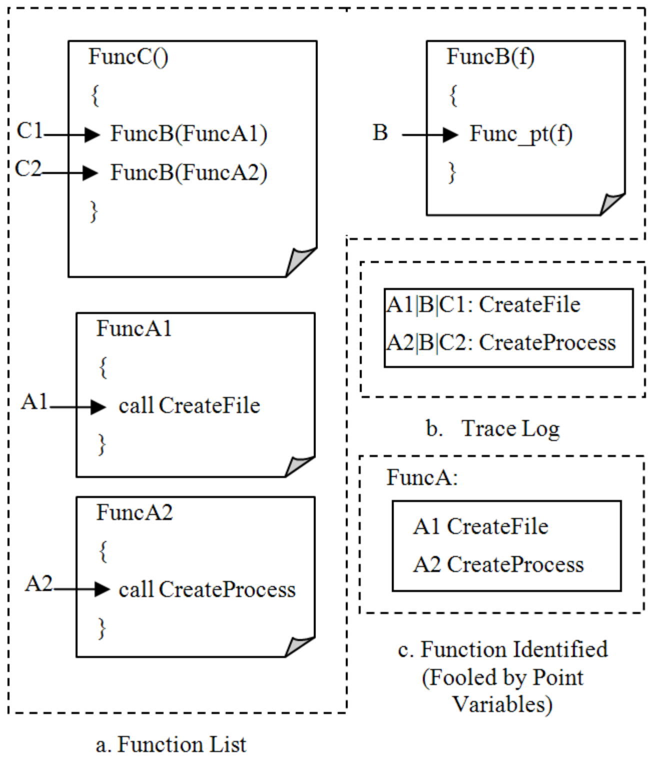


Figure 5.5: Functions Point Variables

5.3.1.2 Breaking Functions

To visualize the system call trace and make the structural system call trace graph useful, the size of the block is very important. If the size of the block is too large, it is hard for human to read. If it is too small, then we will end up with too many blocks. In the function level, the number of system calls for each function is irregular. Some function contains many system calls while other functions do not contain any system call. For a large function, we need to further split it; for smaller functions, we need to merge them.

The splitting process is based on the repetitions and branches of system calls inside system call traces. Ideally, one block should consist of sequential instructions and there is no control flow transfer instructions or control flow transfer destination from outside in the between. However, there is no explicit control flow transfer instruction in the system call trace. We will use the repetition of the system call to infer the presence of the control flow transfer instructions.

Here are several observations we shall take advantage of when we traverse through the system call log one system call by one system call.

(i) If the current system call starts a repetition, then first occurrence of this system call should start a new block, since it is a destination of a control flow transfer operation (loop). Because it is a repeated system call, surely we need to remove subsequent occurrence

(ii) If the repeated segment in the system call trace branches into a block other than the current block, then we can infer that there is a control flow transfer instruction in the between. We shall split the current block and the splitting point is right before this branch

Figure 5.6 illustrates the splitting process. Suppose we have system calls in our trace shows in figure 5.6.a. From the system call trace, we can infer that there exists a control flow transfer instruction from GetProcAddr back to LoadLibrary. So we divide this block into two.

From the system call trace, we can infer that there exists a control flow transfer instruction from GetProcAddr back to LoadLibrary. So we divide this block into two.

On the contrary, some blocks are too small. We need a reverse process to merge several blocks into one. The heuristics we use to merge different blocks include:

(i) If a parent function invokes a serial of functions, and all these functions contain small number of system calls, we can leave parent function and remove all child function and move all system calls to the parent function.

(ii) Inside a function, at the block level, if a block contains very few system calls, and if there is only one incoming edge (except for self-loop), and all the set of its downstream

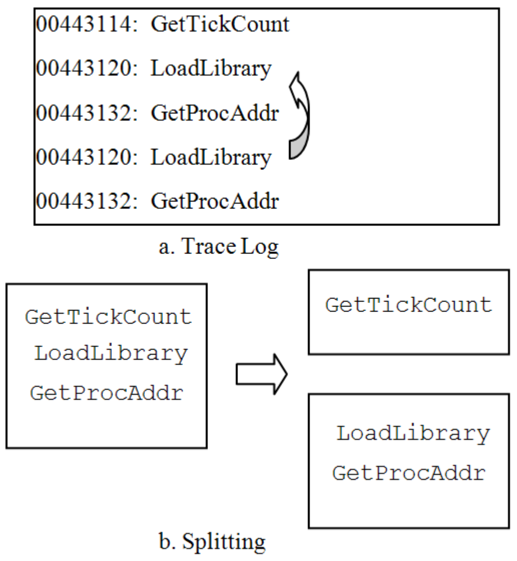


Figure 5.6: Splitting into blocks

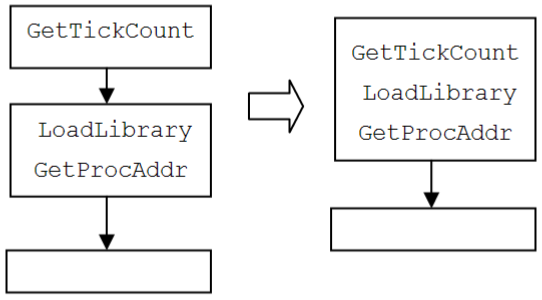


Figure 5.7: Merge blocks

blocks are the subset of the parent nodes' downstream blocks, then we can merge the block into its upstream block and maintain the logic correctness. Figure 4 is a case which we can merge the block to its upstream block

5.3.2 Constructing Graph

Now we finish dividing the system call trace into blocks, and removing repetition. Following the system call trace, we can add edges to the block according to the execution flow revealed by the system call trace. We also accumulate the weights of every edge. Every time control flow follows this edge, we add weights by 1.

5.3.3 Generating Block Description

The remaining work is to put the contents inside each block. The questions we need to answer here are:

(i) What is the order of system calls inside a block

(ii) How to summarize the parameters of each system call

For question 1, we have two ways to order system call:

(i) Sequential order as appeared in system call trace

(ii) Sort the return address of the system call

Since the goal of our structural system call diagram is to visualize the dynamic system call trace, the sequential order of system calls inside the system call log is more important and understandable, so we pick the first criteria. The system call appear first in system call trace will be the first in the block.

Each system call can be invocated several times with different parameters. In our visualizer implementation, we focus on string parameter only. The most important parameters include filename, directory name, registry key name, function name, etc. If there are different invocations for a system call, we list all string parameters. Also, some string parameters are very long, such as absolute path to a system file. In these cases, we use partial string instead of full string.

5.4 Structural System Call Visualizer

Based on the approach we discuss in the last section, we have built a structural system call diagram visualizer based on system call trace. The visualize will first execute the target program, extract system call trace, then build structural system call diagram and present it to the user.

To avoid excessive waiting time, we only capture system call up to 10 seconds even if the target program is not finished. Because of this, we cannot capture the complete system call trace in many cases. However, most malicious software will reveal its malicious behavior in the beginning part of its execution. We believe 10 seconds is a good compromise between effectiveness and efficiency.

5.5 Smart System Call Signature

In this section, we present an application of structural system call diagram. We build a smart signature based on system call trace and use this signature to detect a family of malicious software.

The prevailing technique in the anti-virus industry is based on signature matching. The detection mechanism searches for a binary signature pattern that identifies a particular virus or strain of viruses. However, through constant code rewriting, polymorphism [2] and metamorphism [3], malicious software writer can create a new variance of the same malicious software family quickly. The old signature cannot capture the new variance of malicious software.

Malicious software is able to change its appearance easily. However, the behavior of the malicious software of the same family is quite similar. Behavior of software is revealed by a serial of system calls. This is why we use system call as signature to detect malicious software family.

Structural system call diagram breaks system call traces into natural blocks reflecting the structure of the program. We use system calls inside a block as signature.

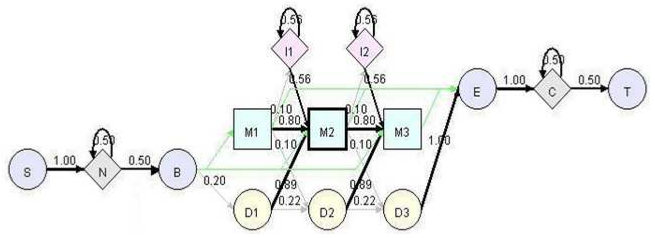


Figure 5.8: Profile Hidden Markov Model

In this section, we discuss the profile hidden Markov model (PHMM) [95] and how to derive the system call signature manually. In next section, we will describe an automatic process to derive the system call signature.

Using our visualizer, security experts are able to compare the similarities and differences between different variances of the same malicious software family through structural system call diagram. They will derive the most prominent block and use this block as a signature.

However, malicious software changes its code for every variation and there may be slight difference between variations. In this sense, we do not use exact match, instead, we build a smart signature using PHMM. Our smart signature allows insertion, deletion or mutation for one or more positions inside system call signature.

PHMM is a specific form of hidden Markov model (HMM) and is widely used in bioinformatics for sequence analysis [95]. In simple words, profile hidden Markov model is a probabilistic model to represent a serial of sequences. For an unknown sequence, we can use existing PHMM to score it. The result shows the probability that the sequence being generated by the PHMM.

Figure 5.8 shows a diagram of PHMM. The PHMM starts from start (S) state and ends at terminal (T) state. The core of PHMM between beginning (B) and ending (E) states consists of the matching (M) states, insertion (I) states, and deletion (D) states. A matching state (e.g., M1-M3 in Figure 5.8) represents a fairly stable system call. Each matching state has a deletion state (e.g., D1-D3 in figure 5.8) associated with it, allowing the deletion of the matching state (or position). Each matching state except for the last one also has an insertion state (e.g., I1-I2 in figure 5.8) associated with it, allowing the insertion of additional positions after it. Transitions between I and D states are not allowed. N and C are two special states to accommodate additional insertions before and after the conserved regions of a family of sequences, which allows local alignment between a sequence and the PHMM (i.e. matching a part of a sequence against the core of PHMM between state B and state E). Another interesting feature of the PHMM is that there is a transition from B to each M state, and a transition from each M state directly to E state. These transitions make it possible to match only a part of the model against a sequence, allowing local alignment with respect to the PHMM. Each M, I, N, C states has an emission probability vector derived from input sequences. In our case, every state emits a system call of different category with a specific distribution.

When we score a structural system call block against the PHMM, we will get a log-odds score. It is the logarithm of the ratio between the probability that the block is generated by the PHMM and the probability that it is generated randomly.

In our first experiment, we build PHMM from one manually picked system call block. For every system call inside the block, we assign a matching state. We use a fixed value for all transition probabilities and emission probabilities for simplicity. The PHMM we built is able to identify the structural system call block we pick, and the system call block similar to it, even if there are some insertions, deletions or mutations.

For every unknown program, we first run the program, capture system call traces, and then generate structural system call diagram for that unknown program. For each block, we score it using derived PHMM. If we find any block similar to our signature, we consider it a variance of the malicious software family.

5.6 Automatic Smart Signature Generation

Using our visualizer, security experts are able to visualize and analyze system call trace easier. However, derive a structural system call block signature is still time consuming. To this end, we designed a mechanism to derive system call signature automatically from a collection of malicious software from the same family.

Malicious software signature should have the following characters:

(i) It is very rare in benign code

(ii) It exists in many variances in the malicious software family

We design an algorithm to generate system call signature automatically. The input is a family of malicious software code and a pool of benign software. The algorithm consists of two separate steps:

(i) We first generate a benign system call block database which consists of all system call blocks exist inside our benign software pool. To do that, we need to construct structural system call diagram for every benign code in the pool. After that, we generate structural system call diagram for every malicious code. For each block inside structural system call diagram, we build a PHMM and score it against each system call block in our benign system call block database. If the number of matches exceeds a threshold, we mark this block as benign and do not use it as malicious software signature

(ii) The remaining system call blocks of malicious software family are candidates for signature. Our goal is to find the system call block which exists in most malicious software. We first construct structural system call diagram for each malicious code, build PHMM for each system call block. We try to find similar block in other malicious code. So for each PHMM, we score it against blocks of all other malicious code. If we find a similar block in one malicious code, it means this signature is able to capture that malicious code. We maintain a counter for each signature candidate. The signature candidate captures the most number of malicious code is the signature we choose

5.7 Experimental Results

5.7.1 Visualizer

We first use our visualizer to generate structural system call diagram for both benign and malicious code.

Figure 5.9 shows the structural system call diagram for a benign program "charmap.exe". From the diagram, we can see that the executable start with reading profile information, and then open several Unicode extension files. Instead of reading hundreds of lines of system call traces, structural system call diagram is much succinct and easier to read.

Figure 5.10 shows a similar structural system call diagram for a malicious program "SD-Bot.afz".

5.7.2 Dataset

To experiment on system call signature, we choose two bots families: SDBot and RBot. Botnet [13] is malicious software characterized by infecting a large number of zombies to form a malicious network. Each victim computer is called a bot. We choose to use bot code in our experiment because we can find a large number of variances for each bot family. We find 935 different variances of SDBot and 1249 different variances of RBot from vx heaven [13]. Some of those variances are not runnable on our computer. We use the runnable variances only as our dataset which include 595 SDBot and 1055 Rbot.

We use commercial antivirus software to scan our malware dataset. Table 5.1 shows the result virus scanner by three commercial antivirus software. It is no surprise that antivirus

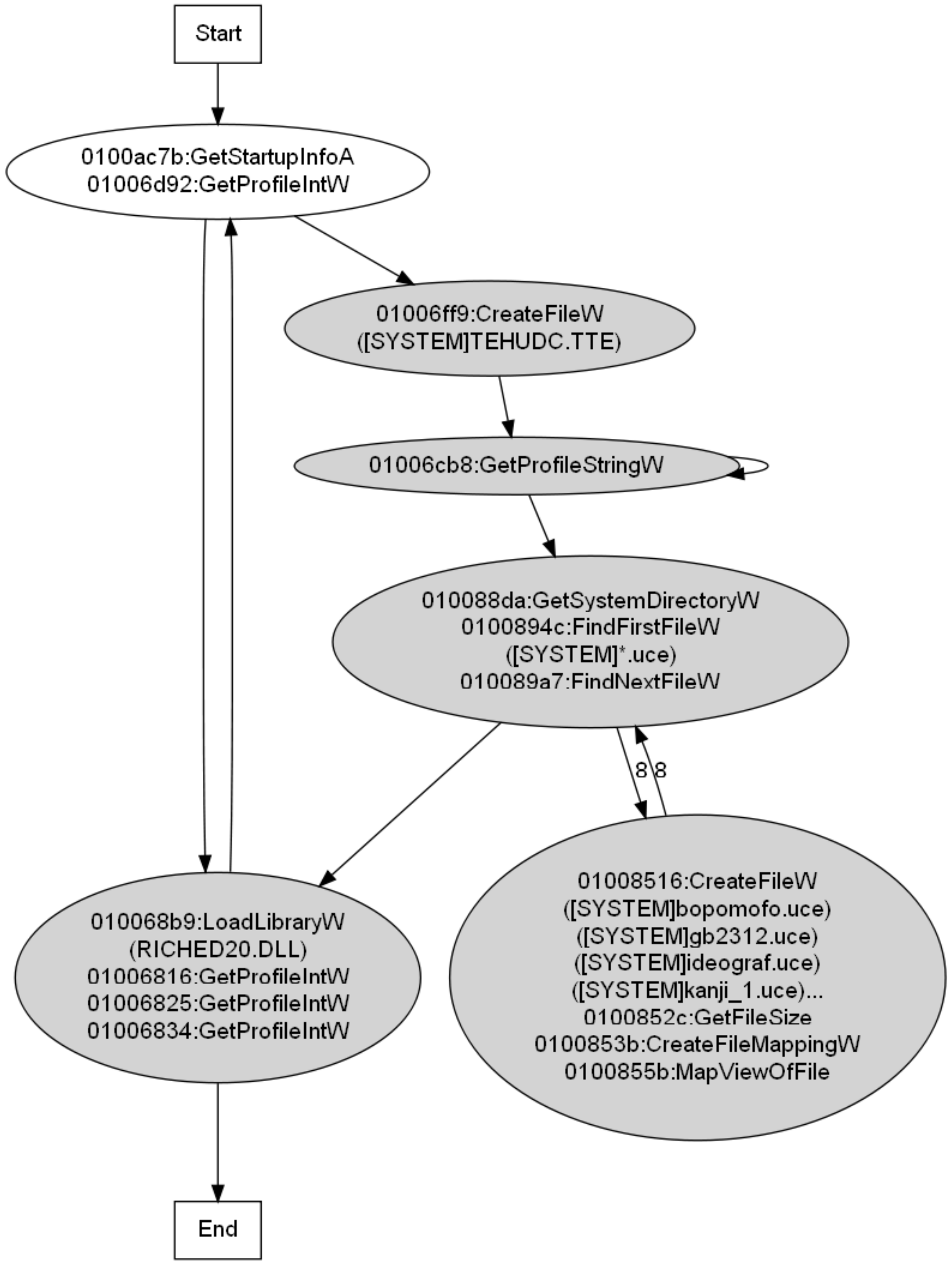


Figure 5.9: Structural system call diagram for “charmap.exe”

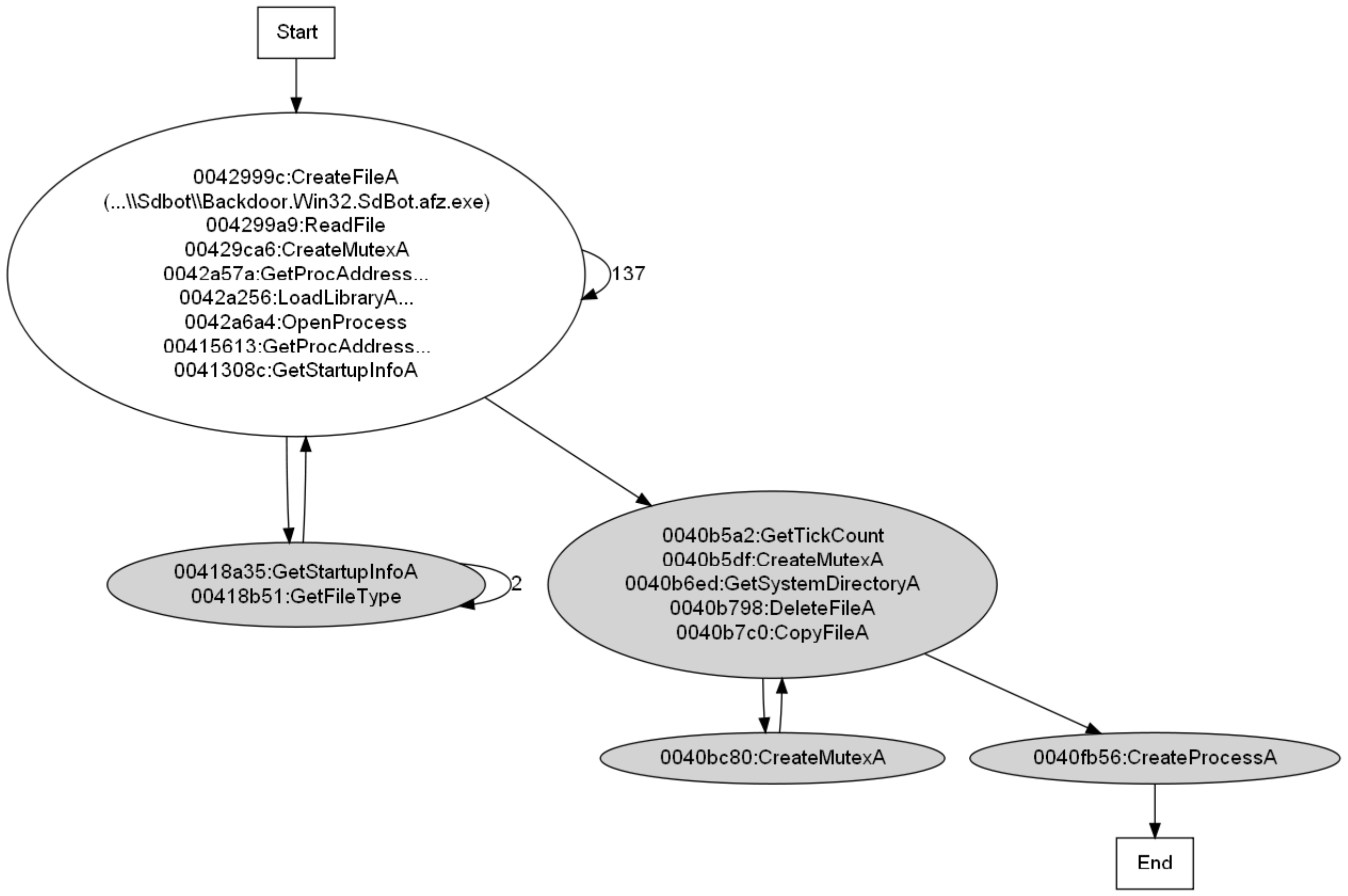


Figure 5.10: Structural system call diagram for “SdBot.afz”

software capture most of the malware samples since they are known malware and be around for times. None of our samples is missed by all three virus scanners we have tested.

We also randomly pick 878 benign executables which consist of Windows system executables, commercial executables and open source executables as our benign dataset.

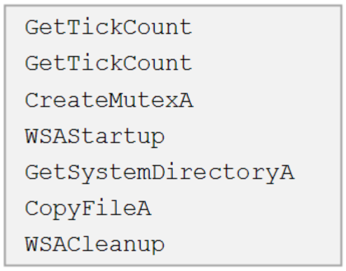
We report detection rate on the malicious software dataset and the false positive rate on the benign dataset as the criteria to evaluate the performance of our smart signature.

5.7.3 Manual Signature Generation

We first manually analyze structural system call diagram using our visualizer for SDBot. By carefully comparing different variances of SDBot instances, we conclude that there is a

Table 5.1: Missed Malware by Commercial Antivirus Software

	Norton Antivirus 2009	**McAfee VirusScan Plus**	**Kaspersky AntiVirus 2009**
SDBot	SdBot.aig SdBot.dh	SdBot.aig SdBot.dh	SdBot.jb
RBot	Rbot.abk Rbot.ael Rbot.gt Rbot.iw Rbot.jb Rbot.yv Rbot.zf	Rbot.am Rbot.ia	Rbot.aek



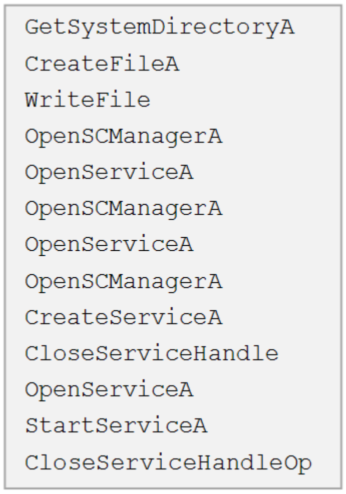
```
GetTickCount
GetTickCount
CreateMutexA
WSAStartup
GetSystemDirectoryA
CopyFileA
WSACleanup
```

Figure 5.11: System call signature for SDBot

specific system call pattern for many of variances of SDBot family. We use that block as our signature. Figure 5.11 shows the signature we get.

Using this system call signature, we are able to capture 222 out of 595 SDBot, or 37.3% detection rate. The false positive rate for the signature in our benign dataset is 0.9%.

Another interesting finding is that inside the SDBot variances which cannot be captured by our system call signature, we can still find some other system call patterns. So we exclude SDBot we have already captured from the dataset, and run our automatic signature generation tool again. We get a secondary system call signature (Figure 5.12) which is able to capture another 58 SDBot variance. By using two system call signatures, our detection rate is 47.1%, and false positive rate is 1.6%.



```
GetSystemDirectoryA
CreateFileA
WriteFile
OpenSCManagerA
OpenServiceA
OpenSCManagerA
OpenServiceA
OpenSCManagerA
CreateServiceA
CloseServiceHandle
OpenServiceA
StartServiceA
CloseServiceHandleOp
```

Figure 5.12: Secondary system call signature for SDBot

5.7.4 Automatic Signature Generation

Next, we experiment our automatic system call signature generation tool.

We use 5-fold cross-validation [7] to evaluate the performance in this experiment. We randomly divide the dataset into 5 subsamples of the same size. We then repeat the cross-validation process 5 times. Every time we use one subsample as testing dataset and all other subsamples as training dataset. Our result is the average of all 5 runs. We will generate smart signature based on the information given by the training dataset and use it to detect the nature of testing dataset.

The detection rate for SDBot is 37.0% and the false positive rate is 0.9%, very close to that of manually generated signature.

We also experiment on another bot family, RBot. Use the automatic generated system call signature, we capture 40.1% variances of RBot, and the false positive rate is 1.0%.

However, not all malicious software families can achieve similar result. We also tested system call signature on AgoBot, the detection rate is 21.4%. The effectiveness of system call signature depends on the character of the specific malicious software family. By using our visualizer, we can get an impression whether or not the specific malicious software family shows a pattern inside system call blocks. If it does, we can extract the signature manually or use our signature generation tool to extract the signature automatically.

5.8 PERFORMANCE ANALYSIS

In this section, we use the following symbols.

M	average size of system call signature
N	number of training malwares
B	average number of blocks in a structral system call diagram
T	average length of system call trace

We discuss two different aspects of system time complexity.

Decision time complexity is the time complexity to determine whether an unknown executable is a malware or not based on an existing model. The dominant part for the decision process is the scoring algorithm of profile hidden Markov model. It is bounded to $O(M^2T)$.

To generate system call signature automatically, we first need to generate system call blocks. By one linear scan, we can organize system calls into different functions, which takes O(T). We then need to compare each pair of function, if their address spaces overlap, we merge them. Suppose the average number of functions is F, this process takes $O(F^2)$. Merge and Splitting is performed recursively, however, we put a limit on the number of iterations. To find a candidate block to split, we need to scan all the blocks for the size exceeding the threshold, which takes O(B). For each splitting, we need to traverse all the system calls inside the block which is O(T/B). So the time complexity for splitting is O(c*B*T/B) = O(T), where c is the limit we put on number of iterations. For merging, it also takes O(B) to search merge candidate pairs. The actual merge include merging system calls inside both block, which is O(T/B). So the time complexity for merging is O(c*B*T/B) = O(T) as well. Adding all above together, we have $O(T) + O(F^2) + O(T) \leq O(T^2)$ to generate a signature.

It takes O(M) to build profile hidden Markov model for that signature using our approach. Altogether, we have O(B*N) blocks. We will score each block against all other O(B*N) blocks.

The overall time complexity to generate a system call signature is bounded by $O(T^2)$* O(M)*O(B*N) *O(B*N) = $O(MT^2B^2N^2)$.

5.9 Conclusion

In this section, we illustrate a structural dynamic system call graph and use it to detect a family of malicious software automatically. We focus only on the dynamic system call captured from runtime, and then create a graph structure similar to control flow graph. Use this structural system call diagram we are able to visualize and analyze an executable efficiently.

We then use system call block inside structural system call diagram as a natural boundary and generate system call signature. We use both manual approach and automatic approach to extract system call signature from a malicious software family. Our experimental result shows that this system call signature is effective in detecting some variances in a malicious software family.

Detection rate for system call approach is low, combining system call and dynamic instruction sequence may provide a better solution. We will investigate it in our future work.

CHAPTER 6: CONCLUSION

In this research, we have built a dynamic execution environment. We can run arbitrary program inside the dynamic execution environment with control. Based on this dynamic execution environment, we have proposed two novel malicious code detection approaches:

(i) Mining dynamic instruction sequences to detect Win32 viruses.

(ii) Utilizing structural system call diagram to detect a family of malicious software.

There are limitations in our approaches. The experiments utilize a finite number of instructions or system calls, hence will not detect any malicious code hooked in the remaining part of the executable code. In addition, since we follow one possible executable path, it is possible to miss malicious section of code in our experiment.

REFERENCES

[1] "Symantec Global Internet Security Threat Report," http://www4.symantec.com/Vrt/wl?tu_id=gCGG123913789453640802, July 2009.

[2] Carey Nachenberg, "Computer Virus-Antivirus Coevolution," *Communications of the ACM*, vol. 40, no. 1, pp. 46–51, Jan. 1997.

[3] Peter Szor and Peter Ferrie, "Hunting for Metamorphic," in *11th International Virus Bulletin Conference*, 2001.

[4] Jeffrey Kephart, William Arnold, "Automatic Extraction of Computer Virus Signatures," in *Virus Bulletin International Conference*, 1994.

[5] http://zert.isotf.org, July 2009.

[6] Peter Szor, *The Art of Computer Virus Research and Defense.* Addison Wesley, 2005.

[7] Jiawei Han, Micheline Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, Mar. 2006.

[8] http://www.cert.org/advisories/CA-2001-19.html, July 2009.

[9] http://www.f-secure.com/v-descs/klez.shtml, July 2009.

[10] http://www.microsoft.com/technet/security/Bulletin/MS01-020.mspx, July 2009.

[11] http://vil.nai.com/vil/content/v_145625.htm, July 2009.

[12] Greg Hoglund, James Butler, *Rootkits: Subverting the Windows Kernel.* Addison Wesley, 2005.

[13] "Bots and Botnet: An Overview," http://www.sans.org/reading_room/whitepapers/malicious/bots_and_botnet_an_overview_1299.

[14] Anirudh Ramachandran, Nick Feamster, “Understanding the Network-Level Behavior of Spammers,” in *ACM SIGCOMM Computer Communication Review*, 2006.

[15] Ryan Vogt, John Aycock, “Attack of the 50 Foot Botnet,” *Technical report, Department of Computer Science, University of Calgary*, 2006.

[16] http://www.clamav.net, July 2009.

[17] Arun Lakhotia, Aditya Kapoor, Eric Uday, “Are Metamorphic Viruses Really Invincible?” in *Virus Bulletin*, 2004.

[18] Mohamed Chouchane, Arun Lakhotia, “Using Engine Signature to Detect Metamorphic Malware,” in *Proceedings of the 4th ACM workshop on Recurring malcode*, 2006.

[19] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, Arun Lakhotia, “Normalizing Metamorphic Malware Using Term Rewriting,” in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 2006.

[20] Abhishek Karnik, Suchandra Goswami, Ratan Guha, “Detecting Obfuscated Viruses Using Cosine Similarity Analysis,” in *First Asia International Conference on Modelling and Simulation, 2007*, 2007.

[21] David Brumley, James Newsome, Dawn Song, Hao Wang, Somesh Jha, “Towards Automatic Generation of Vulnerability-Based Signatures,” in *2006 IEEE Symposium on Security and Privacy*, 2006.

[22] Wing Wong, Mark Stamp, “Hunting for Metamorphic Engines,” *Journal in Computer Virology*, vol. 2, no. 3, 2006.

[23] Mark Stamp, Srilatha Attaluri, Scott McGhee, “Profile Hidden Markov Models and Metamorphic Virus Detection,” *Journal in Computer Virology*, 2008.

[24] Fred Cohen, “Computational Aspects of Computer Viruses,” *Computers and Security*, vol. 8, pp. 325–344, 1997.

[25] Matt Pietrek, “An In-Depth Look into the Win32 Portable Executable File Format by Matt Pietrek (Part I),” http://msdn.microsoft.com/en-us/magazine/cc301805.aspx.

[26] ——, “An In-Depth Look into the Win32 Portable Executable File Format by Matt Pietrek (Part II),” http://msdn.microsoft.com/en-us/magazine/cc301808.aspx.

[27] http://www.securityfocus.com/infocus/1841, July 2009.

[28] Michael Weber, Matthew Schmid, David Geyer, Michael Shatz, “A Toolkit for Detecting and Analyzing Malicious Software,” in *18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

[29] William Arnold, Gerald Tesauro, “Automatically Generated Win32 Heuristic Virus Detection,” in *Virus Bulletin Conference*, Sept. 2000.

[30] InSeon Yoo, “Visualizing Windows Executable Viruses Using Self-Organizing Maps,” in *Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, Oct. 2004.

[31] Matthew Schultz, Eleazar Eskin, Erez Zadok, Salvatore Stolfo, “Data Mining Methods for Detection of New Malicious Executables,” in *Proceedings of IEEE Symposium on Security and Privacy*, May 2001.

[32] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, Ray Sweidan, “Detection of New Malicious Code Using N-grams Signatures,” in *Proceedings of the Second Annual Conference on Privacy, Security and Trust (PST’04)*, Oct. 2004, pp. 193–196.

[33] ——, “N-Gram-based Detection of New Malicious Code,” in *Proceeding of the 28th Annual International Computer Software and Applications Conference (COMPSAC’04)*, 2004.

[34] Jeremy Kolter, and Marcus Maloof, "Learning to Detect Malicious Executables in the Wild," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 470–478.

[35] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, Qingshan Jiang , "An Intelligent PE-malware Detection System Based on Association Mining," *Journal in Computer Virology*, vol. 4, no. 4, 2008.

[36] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, Randal Bryant, "Semantics-Aware Malware Detection," in *IEEE Symposium on Security and Privacy*, 2005.

[37] Mihai Christodorescu, Somesh Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *12th Usenix Security Symposium*, 2003.

[38] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, Helmut Veith, "Detecting Malicious Code by Model Checking," *Intrusion and Malware Detection and Vulnerability Assessment*, vol. 3548/2005, pp. 174–187, 2005.

[39] Muazzam Siddiqui, *Data Mining Methods for Malware Detection, PhD Dissertation.* University of Central Florida, 2008.

[40] http://upx.sourceforge.net, July 2009.

[41] http://www.datarescue.com.

[42] "Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables," http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf.

[43] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *22nd Annual Computer Security Applications Conference (ACSAC06)*, 2006.

[44] Min Gyung Kang, Pongsin Poosankam, Heng Yin, "Renovo: A Hidden Code Extractor for Packed Executables," in *The 5th ACM Workshop on Recurring Malcode (WORM)*, 2007.

[45] Christopher Kruegel, William Robertson, Fredrik Valeur, Giovanni Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[46] Cullen Linn, Saumya Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, Oct. 2003.

[47] Benjamin Schwarz, Saumya Debray, Gregory Andrews, "Disassembly of Executable Code Revisited," in *Ninth Working Conference on Reverse Engineering (WCRE 2002)*, 2003, p. 0045.

[48] Nanda, S., Wei Li, Lap-Chung Lam, Tzi-cker Chiueh, "BIRD: Binary Interpretation Using Runtime Disassembly," in *International Symposium on Code Generation and Optimization (CGO2006)*, 2006.

[49] Aditya Kapoor, "An Approach Towards Disassembly of Malicious Binary Executables," *Master Thesis*, 2004.

[50] Ulrich Bayer, Christopher Kruegel, Engin Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *EICAR. 15th Annual Conference of the European Institute for Computer Antivirus Research*, 2006.

[51] Andrew Sung, Jianyun Xu, P. Chavez, Srinivas Mukkamala, "Static Analyzer of Vicious Executables (SAVE)," in *20th Annual Computer Security Applications Conference*, 2004.

[52] Steven Hofmeyr, Stephanie Forrest, Anil Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security*, vol. 6, no. 3, 1998.

[53] Wenke Lee and Salvatore Stolfo, "Data Mining Approaches for Intrusion Detection," in *7th USENIX Security Symposium*, 1998.

[54] Warrender, Stephanie Forrest, Barak Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[55] Tony Lee, Jigar Mody, "Behavior Classification," Microsoft Antivirus team, 2006.

[56] Mihai Christodorescu, Somesh Jha, Christopher Kruegel, "Mining Specifications of Malicious Behavior," in *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, 2007.

[57] Suresh Chari, Pau-Chen Cheng, "BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, 2003.

[58] Niels Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th Usenix Security Symposium*, 2003.

[59] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, Jose Nazario, "Automated classification and analysis of internet malware," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, 2007.

[60] David Wagner, Drew Dean, "Intrusion Detection via Static Analysis," in *2001 IEEE Symposium on Security and Privacy*, 2001.

[61] Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski, Robert K. Cunningham, “Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code,” in *Proceedings of the 2003 ACM workshop on Rapid malcode*, 2003.

[62] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” in *2001 IEEE Symposium on Security and Privacy*, 2001.

[63] Henry Hanping Feng, Oleg Kolesnikov, Prahlad Fogla, Wenke Lee, Weibo Gong, “Anomaly Detection Using Call Stack Information,” in *2003 IEEE Symposium on Security and Privacy*, 2003.

[64] Debin Gao, Michael Reiter, Dawn Song, “Gray-box Extraction of Execution Graphs for Anomaly Detection,” in *11th ACM conference on Computer and communications security*, 2004.

[65] Abhishek Chaturvedi, Eep Bhatkar, R. Sekar, “Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments,” in *IEEE Symposium on Security and Privacy*, 2006.

[66] James Newsome, Dawn Song, “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,” in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[67] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, Dawn Song, “Dynamic Spyware Analysis,” in *USENIX Annual Technical Conference (USENIX 07)*, 2007.

[68] Andreas Moser, Christopher Kruegel, Engin Kirda, “Exploring Multiple Execution Paths for Malware Analysis,” in *2007 IEEE Symposium on Security and Privacy*, 2007.

[69] http://bellard.org/qemu, July 2009.

[70] http://www.vmware.com, July 2009.

[71] http://www.codeproject.com/KB/threads/winspy.aspx, July 2009.

[72] Galen Hunt, Doug Brubacher, “Detours: Binary Interception of Win32 Functions,” in *3rd USENIX Windows NT Symposium*, 1999.

[73] http://www.ollydbg.de, July 2009.

[74] Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, “Malware Phylogeny Generation using Permutations of Code,” *Journal in Computer Virology*, vol. 1, no. 1-2, Nov. 2005.

[75] Rakesh Agrawal, Ramakrishnan Srikant, “Fast Algorithms for Mining Association Rules,” in *Proc. 20th Int. Conf. Very Large Data Bases(VLDB)*, 1994.

[76] Jiawei Han , Jian Pei , Yiwen Yin, “Mining Frequent Patterns Without Candidate Generation,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[77] John Quinlan, “Induction of Decision Trees,” *Machine Learning*, 1986.

[78] ——, *C4.5:Programs for Machine Learning.* Morgan Kaufmann, 1993.

[79] Leo Breiman, Jerome Friedman, Charles Stone, R.A. Olshen , *Classification and Regression Trees*, 1984.

[80] G.V.Kass, “An Exploratory Technique for Investigating Large Quantities of Categorical Data,” *Applied Statistics*, 1980.

[81] John Shawe-Taylor, Nello Cristianini, *Support Vector Machines.* Cambridge University Press, 2000.

[82] Kristin Bennett, Colin Campbell, "Support Vector Machines: Hype or Hallelujah?" *SIGKDD Explorations*, no. 2, 2000.

[83] http://www.csie.ntu.edu.tw/~cjlin/libsvm, July 2009.

[84] John Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," *Microsoft Research Technical Report*, no. MSR-TR-98-14, 1998.

[85] Leo Breiman, "Random Forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.

[86] http://www.salford-systems.com/treenet.php, July 2009.

[87] Ron Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection." Morgan Kaufmann, 1995, pp. 1137–1143.

[88] Leo Breiman, "Bagging Prediction," *Machine Learning*, pp. 123–140, 1996.

[89] Yoav Freund, Robert E.Schapire , "A Decision-Theoretic Generalization of Online Learning and an Application to Boosting," *Journal of Computer and System Sciences*, no. 55, 1997.

[90] Vladimir Vapnik, *Statistical Learning Theory.* Wiley, 1998.

[91] http://vx.netlux.org, July 2009.

[92] http://www.heaventools.com, July 2009.

[93] http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html, July 2009.

[94] http://support.microsoft.com/kb/177429, July 2009.

[95] Anders Krogh, Michael Brown, Saira Mian, Kimmen Sjolander, David Haussler, "Hidden Markov models in Computational Biology: Applications to Protein Modeling," *Journal of Molecular Biology*, vol. 55, pp. 1501–1531, 1994.