Detection and Classification of Obfuscated Malware

Moustafa Saleh
Department of Computer Science, University of Texas at San Antonio

DETECTION AND CLASSIFICATION OF OBFUSCATED MALWARE

APPROVED BY SUPERVISING COMMITTEE:

_____
Shouhuai Xu, Ph.D.

_____
Pang Du, Ph.D.

_____
Tongping Liu, Ph.D.

_____
Hugh Maynard, Ph.D.

_____
Meng Yu, Ph.D.

Accepted: _____
Dean, Graduate School

DEDICATION

This dissertation is dedicated to my family. Thank you for providing me with constant inspiration.

DETECTION AND CLASSIFICATION OF OBFUSCATED MALWARE

by

MOUSTAFA E. SALEH, M.SC.,

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2016

ACKNOWLEDGEMENTS

I would never let such a moment go without showing my gratitude and my deepest thanks to those who helped me throughout my life to reach this moment. First of all, I would like to thank my parents for everything they have given me. Thank you for instilling in me the love of seeking knowledge from my early childhood, and thank you for shaping me as the man I am.

My deep thanks go to my supervisor Prof. Shouhuai Xu for all his help and guidance across all my PhD program. Thank you for believing in your students and pushing them past what they have perceived as a limit. I would like to thank Prof. Pang Du, Prof. Hugh Maynard, Prof. Tongping Liu and Prof. Meng Yu. It is my great pleasure to have you serving on my PhD defense committee.

In my daily work in my beloved lab, I was blessed to have the company of my dear friends, Sajjad Khorsandroo, Mahmoud Abdelsalam, Smriti Bhatt and Marcus Pendleton. It is a great pleasure to be working with you. I can not forget in this moment to thank Peter Ferrie. The one that through his many articles taught me the love of malware hunting and its exciting brain-challenging game. Thank you for your patience to answer my many questions and for giving me always the best answers.

I would like also to thank John Charlton and Paul Ratazzi for their comments and editing. I also would like to thank VirusTotal team for providing me with a great set of samples that this dissertation would not be possible without it.

Last but not least, I would like to thank my wife. No words can express my gratitude towards you. You have been always patient and supportive when I spend many nights and weekends away working on this dissertation. Thank you for being a great wife and a wonderful mother.

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

May 2016

This dissertation was partly supported by NSF under Grant No. 1111925 and ARO under Grant No. W911NF-12-1-0286. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the position of the NSF or the US Army.

DETECTION AND CLASSIFICATION OF OBFUSCATED MALWARE

Moustafa E. Saleh, M.Sc.,
The University of Texas at San Antonio, 2016

Supervising Professor: Shouhuai Xu, Ph.D.

The large number of malicious files that are produced daily outpaces the current capacity of malware analysis and detection. For example, Intel Security Labs report that during each hour in the third quarter of 2015, more than 3.5M infected files were exposed to their customers' networks, and an additional 7.4M potentially unwanted programs attempted to install or launch. The damage of malware attacks is also increasingly devastating, as witnessed by the recent Cryptowall malware that has reportedly generated more than $325M in ransom payments to its perpetrators. In terms of defense, it has been widely accepted that the traditional approach based on byte-string signatures is increasingly ineffective, especially for new malware samples and sophisticated variants of existing ones. New techniques are therefore needed for effective defense against malware. Motivated by this problem, the dissertation investigates three new defense techniques against malware.

The first technique aims at the automatic detection of program obfuscation, which has been abused by malware writers as an attack strategy to make their malware evade the defense. The key idea is to extract and exploit useful information from Control Flow Graphs (CFGs) of malware programs. Experimental results show that the new technique can detect a variety of obfuscation methods (e.g., packing, encryption, and instruction overlapping). This patent-pending technique paves the way for developing the two other techniques presented in the dissertation.

The second technique aims at automatically classifying whether a suspicious file is malicious or not. The suspicious file may have been identified as obfuscated via the first technique mentioned above (or any technique of its kind). Machine learning methods are used to learn detection models, which are shown to be effective against both plain and obfuscated malware samples. A key contribution of this technique is the definition and utilization of over 32,000 features of files, including file structure, runtime behavior, and instructions. To the best of our knowledge, this is

the first effort that defines and uses such a comprehensive feature set.

The third technique leverages the first technique mentioned above for the automatic identification of malware packers that were used to obfuscate malware programs. Signatures of malware packers and obfuscators are extracted from the CFGs of malware samples. Unlike conventional byte signatures that can be evaded by simply modifying one or multiple bytes in malware samples, these signatures are more difficult to evade. For example, CFG-based signatures are shown to be resilient against instruction modifications and shuffling, as a single signature is sufficient for detecting mildly polymorphic versions of the same malware. Last but not least, the process for extracting CFG-based signatures is also made automatic.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

Malicious software, often referred to as malware, is a program designed to do harm to individuals, corporations, or governments. Currently, malware is used in a wide range of attacks, from propagating through the network, launching Denial of Service (DoS) attacks against web servers, causing service disruption and stealing credit card information to more sophisticated attacks against nuclear plants [73]. In addition, financial malware programs are increasing in number and having a stronger impact each year. For example, the Carbanak malware was able to steal $1 billion from over 100 financial institutions worldwide in the biggest bank heist in history [28].

Each year a massive number of malware programs are produced and spread, and anti-virus vendors struggle to keep up by producing more signatures and working to maintain timely updates to their customers. In the last quarter of 2014, McAfee received more than 350 million malware samples, 50 million of which were new malware. That means there are 387 new threats every minute [29].

Despite the variety of anti-malware, ranging from host to network-based solutions, malware detection remains an open problem. The primary complication to solving the issue is that malware attacks continuously change. Malware programs employ several techniques to avoid detection and to evolve rapidly compared to detection solutions. Techniques that evade signature detection exist, such as simple encryption, polymorphism, or advanced metamorphism [44]. Other techniques target behavior analysis via hook detection [31] and virtual machine detection [39], or even use mimicry attacks to deceive the behavioral analysis system by showing benign behavior [72].

Although the malware detection problem has been discussed in numerous papers, it is evident that more advanced solutions are needed and more research is encouraged to mitigate this serious threat.

1.1 Problem Statement

The evident reason why malware detection lags behind the development is due to the rapid evolution of malware. Malicious programs employ several techniques against each defense method. It is relatively easy and quick for the malware author to obfuscate the sample, while it is difficult and slow to develop a defense against that obfuscated threat. In the following subsection, we will give an overview on how every aspect of the malicious program that could be useful for the process of detection, such as static properties or dynamic behavior, can be obfuscated by the author to avert detection.

1.1.1 Attack Against Static Analysis

Signature detection is one of the most used techniques for malware detection. It is simple and rarely produces false-positive errors [60]. However, one of the most effective attacks against this method is code obfuscation. Code obfuscation is usually done by using packers [65]. Packers are used extensively in malware, and it is widely accepted that the vast majority of malware files are packed [10]. Packers are software programs that employ compression techniques on the executable file to decrease its size, or to hinder copyright infringement. This compression obfuscates the code, so signatures can no longer be detected when an anti-malware system scans the file. Major anti-virus software can be evaded using simple methods of obfuscation [13].

Other attacks, such as polymorphism and metamorphism, change the code continuously during every infection. In these methods, the obfuscation is done on the instruction level. The malware uses techniques such as instruction reordering, garbage code insertion, register swapping, among others [44]. That is why polymorphism and metamorphism are considered to be some of the most sophisticated obfuscation techniques [60].

Reverse engineering is the process of revealing and studying the instructions that compose the binary file. This process is conducted by a malware analyst to study the structure and the behavior of the malware. The process, in many cases, precedes the signature generation of the malware [60].

```
1          xor eax, eax
2          nop
3          nop
4  L1:
5          push eax
6          cmp eax, eax
7          jne fake
8          add ecx, 333h
9          jmp skip
10 fake:
11         DB 0Fh
12 skip:
13         nop
14         nop
15         mov ecx, ecx
16         mov edx, 444h
17         push offset ProcName
18         push eax
19         call GetProcAddress
```

Listing 1.1: Opaque predicate trick snippet.

Some malware programs use techniques to make the revere engineering process more difficult, or deceive the malware analyst and show incorrect instructions. One of these techniques is opaque predicate. Opaque predicate techniques [16] insert conditional statements (usually control flow instructions) whose outcome is constant and known to the malware author, but not clear in static analysis. Thus, a disassembler will follow both directions of the control flow instruction, one of which leads to the wrong disassembly and affects the resulting control flow graph. As an example, listing 1.1 shows an opaque predicate trick inserted on lines 6 and 7 of the code snippet. Since the "compare instruction" on line 6 will always evaluate to true, the `fake` branch will never be taken at runtime. However, to a disassembler, this fact is not apparent and it will evaluate both paths.

In this example, the disassembler will follow the target of the `jne` instruction on line 7, which leads to a byte of data on line 11. The disassembly will continue starting with this byte, `0F`, resulting in decoding an instruction with opcode **`0F`**`9090 8BC9BA44`. This incorrect instruction is `SETO BYTE PTR DS:[EAX+44BAC98B]` as shown in figures 1.1 and 1.2 for two common disassemblers, IDA Pro and OllyDbg, respectively.

```
; START OF FUNCTION CHUNK FOR start

loc_401028:                ; lpProcName
push    eax
cmp     eax, eax
jnz     short loc_401035
```

```
add     ecx, 333h
jmp     short near ptr loc_401035+1
```

```
loc_401035:
seto    byte ptr [eax+44BAC98Bh]
add     al, 0
add     [eax-78h], ch
xor     [eax+0], al
push    eax                ; hModule
call    GetProcAddress
```

Figure 1.1: Portion of IDA Pro graph for the example in listing 1.1.

```
33C0          XOR EAX,EAX
90            NOP
90            NOP
50            PUSH EAX
3BC0          CMP EAX,EAX
75 08         JNE SHORT 00401035
81C1 3303000( ADD ECX,333
EB 01         JMP SHORT 00401036
0F9090 8BC9B( SETO BYTE PTR DS:[EAX+44BAC98B]
04 00         ADD AL,0
0068 88       ADD BYTE PTR DS:[EAX-78],CH
3040 00       XOR BYTE PTR DS:[EAX],AL
50            PUSH EAX
E8 32000000   CALL <JMP.&kernel32.GetProcAddress>
```

Figure 1.2: Portion of OllyDbg disassembly for the example in listing 1.1.

1.1.2 Attack Against Dynamic Analysis

Behavior analysis systems mainly consist of a virtual machine in which the malicious sample will run, and all its activities will be monitored. Then, a virtual machine could be easily restored to a clean snapshot so another sample could be analyzed. However, the detection of a virtual machine presence has been used by malware to avoid showing malicious behavior. Several techniques proved that a virtualized environment can be discovered by detecting the presence of a hypervisor [39, 42]. The virtual machine environment can be also discovered by checking for a list of known

artifacts that exist for a specific vendor. For example, the hard drive device name in a registry key [1] can have the string `qemu`, `vmware`, `xen`, or `virtual` in the cases of Qemu, VMware, Xen or Hyper-V virtual machines, respectively [4]. Other techniques for behavior monitoring is to monitor hooking to operating system APIs. Whether the hooking is done inside a virtual machine or on a bare metal one, the hooking can be detected if it was implemented in the user space [31]. If hooking or the presence of a virtual machine presence is detected, a malicious sample could deceive the monitoring process by showing benign behavior [72].

Another common method of dynamic analysis is emulation. An emulator is a software program that can emulate the execution of file instructions to determine whether it is malicious or not. Emulators tend to implement common instructions and a limited set of the operating system's APIs. Therefore, an emulator can be attacked by using some uncommon APIs or instructions [60]. For example, the MMX set of instructions [12] can be used by some malware programs to hinder the emulation process, since it is an uncommon set of instructions, most likely not implemented by emulators. Other techniques to attack emulation includes timing attacks, in which the malware can measure how long it takes to execute a single or a set of instructions. If an instruction takes longer than usual to be executed, then most likely the program runs under an emulated environment [41, 60].

Another common technique to attack both automated VM sandboxes or emulators is delayed execution, where the malware sleeps for long enough that the emulator will exit before the malware shows its malicious behavior [41]. Yet another attack on both automated VMs and emulators is detecting mouse movements. For example, `UpClick` malware triggers its malicious behavior when the left button of the mouse is clicked, and `BaneChant` malware activates on the infected host after three mouse clicks [1].

[1] `System\CurrentControlSet\Services\Disk\Enum`

1.2 Current Research

Several research have been proposed to tackle the problem of malware detection. Some improve string signature generation [20]. String signature is one of the most widely used techniques for malware detection. However, this technique is ineffective against novel malware or variants of already existing programs. Other research depends on collecting behavioral data of the program to determine if it is malicious or not [8], but these methods suffer from incomplete coverage of different software execution paths. Another direction is to gather data from the file header, such as the number of sections, size of each section, entry point location, etc [55]. This technique is very fast compared to the former. However, it could produce a high error rate if the file is obfuscated, which usually is. Current research aligned with our goals can be divided into two parts, detection of obfuscation and zero-day malware detection. The next two subsections give a short overview of the current research related to these problems.

1.2.1 Obfuscation Detection

Entropy-based detection

Lyda and Hamrock presented the idea of using entropy to find encrypted and packed files [30]. The method became widely used as it is efficient and easy to implement. However, some non-packed files could have high entropy values and thus lead to false-positives. For example, the `ahui.exe` and `dfrgntfs.exe` files have entropy of 6.51 and 6.59 respectively for their `.text` section [68, 69] (These two example files exist in Windows XP 32-bit and are detected by our system as non-packed). In addition to entropy-based evasion techniques mentioned in [64], simple byte-level XOR encryption can bypass the entropy detection as well.

Signature-based detection

A popular tool to find packed files is PEiD, which uses more than 620 packer and crypter signatures [5]. A drawback of this tool is that it can identify only known packers, while sophisticated malware

may use custom packing or crypting routines. Moreover, even if a known packer is used, the malware writer can change a single byte of the packer signature to avoid being detected as packed. In addition, the tool is known for its high error rate [53].

File header anomaly detection

Other research such as [38,50,53,63] use the PE header and structure information to detect packed files. These techniques can get good results only when the packer changes the PE header in a noticeable way.

Besides the listed shortcomings of each technique, notably, none of the aforementioned techniques can statically detect the presence of anti-disassembly tricks or other forms of control flow obfuscation, yet these are now commonly used by a wide range of advanced malware.

1.2.2 Zero-day Malware Detection

There have been many different methods to protect proactively against malicious programs by detecting zero-day malware threats. The following subsections discuss some of these methods.

Static Detection

Almost all anti-virus software use signature based detection. Byte signature detection is very accurate and rarely produces any false positive. However, it takes too long compared to other automatic methods to get a signature of a new malware program. The process of creating the signature entails possession of the malware sample, analyzing it, extracting an accurate signature and making sure there is no collision with other signatures, then deploying the new signature. That is why, although byte signature is usually accurate, it is not an effective solution against unknown malware [60].

Much information can be extracted from the file header, such as imported shared libraries, API names, number of sections, etc. These features could be helpful in having an overview of the file. Some research use PE header features to detect zero-day malware such as [55, 56]. Another tech-

nique combines the header information with opcode and function call frequency [57]. However, these techniques can get good results only when the malware affects the PE header in a distinctive way.

Other techniques use byte n-gram to build a model for classification [26, 35, 48, 49, 54], which rely on the contents of the file. Still, the technique is prone to evasion via obfuscation. Some research utilizes both the control flow graph and the call graph for malware detection, such as [17, 18, 25] where the authors use the features of the extracted graph with machine learning models.

Dynamic Detection

Some papers detect malicious activities by monitoring and logging performance-specific registers of the processor, such as [14, 27, 61], where [61] used the hardware performance information to detect anomalies in execution and hence detect exploitation of legitimate software. Other publications use instruction traces during execution [59]. Others use text strings displayed to the user and compare their semantic to the behavior of the file to determine if the malware is carrying out a stealth activity [23]. Salehi et al use the APIs and their arguments that the sample uses during execution as the feature vector [45]. Other research uses temporal and spatial information of API usage [3], that is, similar to [45], both the APIs and their arguments are collected, but their execution order is considered. In addition, Tian et al uses APIs sequences collected during runtime and then applies pattern recognition algorithms on the extracted pattern [62].

1.3 Thesis Contribution

The main theme of our thesis is automatic proactive detection. In this thesis, we aim to (1) Identify potential malicious files by identifying packed or obfuscated files, so their execution can be terminated or suspended until a human analyst intervenes, or a more rigorous analysis is done. (2) Develop effective and efficient techniques based on machine learning to automatically detect new malware with no need of a previous signature. (3) Automatically construct a compact signature based on the control flow graph of the file to identify packer/obfuscator used in a packed file.

First, we propose a novel technique for detecting obfuscated executable files, which helps to flag potentially suspicious files. Although, benign files can employ some methods of obfuscation, the vast majority of malware programs use a variety of obfuscation methods extensively to evade detection. By detecting obfuscated files, a system administrator on a critical system can stop the execution of the program and apply more rigorous analysis before it does any harm. The proposed technique is able to detect a wide variety of different obfuscation methods such as, packing, encryption, and even some advanced tricks such as instruction overlapping. The method is currently patent pending.

Then we utilize the obfuscation detection technique in the second part of our research to detect new malware samples regardless of the possession of a previous signature. In this method, we combine information from the three different aspects of malware. We define features based on the file structure, the dynamic behavior, and properties extracted from the instructions composing the file. We use machine learning algorithms to combine and learn from these features and output a runtime profile to automatically detect potentially malicious software.

Lastly, we leverage the first method to construct a compact signature based on the control flow graph of the sample that is used to identify the packer or obfuscator applied to the file. We introduce two types of signatures to compare and match two CFGs, an exact signature and an approximate one. The approximate signature with the preprocessing of the graph is able to withstand minor code modifications usually done by attackers. The technique can be used to construct a signature of the control flow graph at the entry point function, and it can be compared to a set of signatures in the database to determine the packer. We believe that the technique has a strong potential to be efficient in detecting malware variants and polymorphic code, which will be implemented in our future work.

1.4 Thesis Organization

The reminder of this thesis is organized as follows. In Chapter 2, we explain the details of our obfuscation detection method, present the experiment, and discuss the result. In Chapter 3, we in-

troduce a method for malware detection using machine learning. We discuss the different contexts or aspects of the malicious sample and how the features are defined and extracted. In Chapter 4, a signature based on the control flow graph of the file is introduced. We discuss graph preprocessing methods, demonstrate the experiment and the result. Finally, the future research directions of this dissertation are summarized in Chapter 5.

Chapter 2: INSTRUCTIONS-BASED DETECTION OF SOPHISTICATED OBFUSCATION AND PACKING

2.1 Introduction

Zero-day malware detection is a persistent problem. Hundreds of thousands of new malicious programs are produced and published on the Internet daily. Although conventional signature-based techniques are still widely relied upon, they are only useful for known malware. Many research efforts have aimed at helping flag and detect unknown suspicious and malicious files. All of these techniques can be categorized into sandbox analysis, heuristic static analysis or code emulation. Among the three, heuristic static analysis is the fastest, yet the weakest against obfuscation techniques. Code obfuscation includes packing, protecting, encrypting or inserting anti-disassembly tricks, and is used to hinder the process of reverse engineering and code analysis. About 80% to 90% of malware use some kind of packing techniques [30] and around 50% of new malware are simply packed versions of older known malware according to a 2006 article [58], and we believe it is far more than that by now. While it is very common for malware to use code obfuscation, benign executable files rarely employ such techniques. Thus, it has become a common practice to flag an obfuscated file as suspicious and then examine it with more costly analysis to determine if it is malicious or not.

Most current work of detecting obfuscated files is based on executable file structure characteristics as we will show in Section 2.2. Many public packers, indeed, exhibit identifiable changes in the packed PE file. However, this is not always the case with custom packers and self-encrypting malware. Moreover, packing is not the only obfuscation technique used by malware writers. Malware can use anti-analysis tricks that hinder the disassembly or analysis process. Such tricks can leave absolutely no trace in the header as it is based on obfuscating the instructions sequence and the execution flow of the program. Other methods depend on detecting the signature of known packers in the file. The drawback of this method is obvious as it does not work with unknown and custom

packers and cryptors. It also fails if the signature is slightly modified. Calculating the entropy score of the file is another method of identifying packed and encrypted files. This method could be effective against encryption or packing obfuscation, but is ineffective against anti-disassembly tricks. In addition, the entropy score of a file can be reduced to achieve low entropy similar to those of normal programs.

In this chapter, we present a new method for detecting obfuscated programs. We build a recursive traversal disassembler that extracts the control flow graph of binary files. This allows us to detect the presence of interleaving instructions, which is typically an indication of the opaque predicate anti-disassembly trick. Our detection system uses some novel features based on referenced instructions and the extracted control flow graph that clearly distinguishes between obfuscated and normal files. When these are combined with a few features based on file structure, we achieve a very high detection rate of obfuscated files.

More specifically, our contributions of the chapter are:

- We leverage the fact that some advanced obfuscated malware use opaque predicate techniques to hinder the process of disassembly, and describe a technique that turns this strength into a weakness by detecting its presence and flagging the file as suspicious (Section 2.3).

- We identify distinguishing features between obfuscated and non-obfuscated files by studying their control flow graphs. These features help detect obfuscated files while avoiding drawbacks of the other methods that rely on file structure.

- We achieve a fast scanning speed of 12 ms per file on average, despite the fact that our method encompasses disassembly, control flow graph creation, feature extraction, and file structure examination.

The rest of the chapter is structured as follows: Section 2.2 briefly review the related work. Section 2.3 discusses the opaque predicate technique that can hinder the process of disassembly. Section 2.4 reveals the statistical characteristics we identified for distinguishing obfuscated files

from non-obfuscated ones. Section 4.5 describes our experiments and results. Section 4.6 discusses the results and potential limitations. Section 4.7 concludes the chapter.

2.2 Related work

2.2.1 Entropy-based detection

Lyda and Hamrock presented the idea of using entropy to find encrypted and packed files [30]. The method became widely used as it is efficient and easy to implement. However, some non-packed files could have high entropy values and thus lead to false-positives. For example, the `ahui.exe` and `dfrgntfs.exe` files have an entropy of 6.51 and 6.59 respectively for their `.text` section [68, 69] (These two example files exist in Windows XP 32-bit and are detected by our system as non-packed). In addition to entropy-based evasion techniques mentioned in [64], simple byte-level XOR encryption can bypass the entropy detection as well.

2.2.2 Signature-based detection

A popular tool to find packed files is PEiD, which uses around 620 packer and crypter signatures [5]. A drawback of this tool is that it can identify only known packers, while sophisticated malware use custom packing or crypting routines. Moreover, even if a known packer is used, the malware writer can change a single byte of the packer signature to avoid being detected as packed. In addition, the tool is known for its high error rate [53].

2.2.3 File header anomaly detection

Other research such as [38, 50, 53, 63] use the PE header and structure information to detect packed files. These techniques can get good results only when the packer changes the PE header in a noticeable way.

Besides the shortcomings of every technique, notably, none of the aforementioned techniques can statically detect the presence of anti-disassembly tricks or other forms of control flow obfus-

cation, yet these are now commonly used by a wide range of advanced malware. In addition, our proposed system does not depend on a coarse-grained entropy score of the file or section, signature of packers, or file header features. Thus, it is able to overcome these shortcomings.

2.3 Anti-disassembly tricks used by malware writers

Malware writers use a variety of anti-analysis tricks to protect against all kinds of analyses. One class of these is anti-disassembly tricks. Anti-disassembly tricks hinder the process of disassembly and hence reduce the effectiveness of static analysis-based detection of malware. One of the most common techniques is to use an Opaque Predicate. Although there are legitimate reasons for including opaque predicate tricks, such as watermarking [36] and to hinder reverse engineering, they are commonly used in malware to prevent analysis.

Opaque predicate tricks [16] insert conditional statements (usually control flow instructions) whose outcome is constant and known to the malware author, but not clear in static analysis. Thus, a disassembler will follow both directions of the control flow instruction, one of which leads to the wrong disassembly and affects the resulting control flow graph. As an example, listing 2.1 shows an opaque predicate trick inserted on lines 6 and 7 of the code snippet. Since the "compare instruction" on line 6 will always evaluate to true, the `fake` branch will never be taken at runtime. However, to a disassembler, this fact is not apparent and it will evaluate both paths.

In this example, the disassembler will follow the target of the `jne` instruction on line 7, which leads to a byte of data on line 11. The disassembly will continue starting with this byte, `0F`, resulting in decoding an instruction with opcode **`0F`**`9090 8BC9BA44`. This incorrect instruction is `SETO BYTE PTR DS:[EAX+44BAC98B]` as shown in figures 2.1 and 2.2 for two common disassemblers, IDA Pro and OllyDbg, respectively.

We developed a recursive traversal disassembler that is able to detect interleaving code and flag the corresponding basic block as problematic, so an analyst could easily know where to find these tricks. Figure 2.3 shows a portion of the control flow graph output from our disassembler for this example. Two blocks are shown in red to indicate that they are interleaving and only one of them

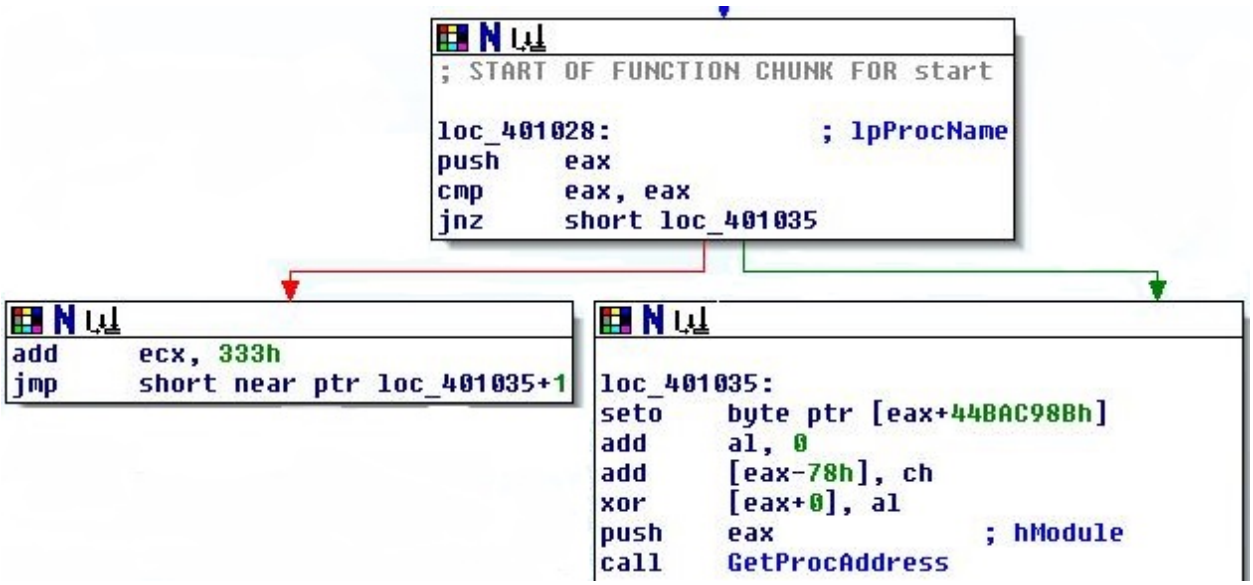

Figure 2.1: Portion of IDA Pro graph for the example in listing 2.1.

is correct.

```
        xor eax, eax
        nop
        nop
L1:
        push eax
        cmp eax, eax
        jne fake
        add ecx, 333h
        jmp skip
fake:
        DB 0Fh
skip:
        nop
        nop
        mov ecx, ecx
        mov edx, 444h
        push offset ProcName
        push eax
        call GetProcAddress
```

Listing 2.1: Opaque predicate trick snippet.

```
33C0          XOR EAX,EAX
90            NOP
90            NOP
50            PUSH EAX
3BC0          CMP EAX,EAX
75 08         JNE SHORT 00401035
81C1 3303000( ADD ECX,333
EB 01         JMP SHORT 00401036
0F9090 8BC9B( SETO BYTE PTR DS:[EAX+44BAC98B]
04 00         ADD AL,0
0068 88       ADD BYTE PTR DS:[EAX-78],CH
3040 00       XOR BYTE PTR DS:[EAX],AL
50            PUSH EAX
E8 32000000   CALL <JMP.&kernel32.GetProcAddress>
```

Figure 2.2: Portion of OllyDbg disassembly for the example in listing 2.1.

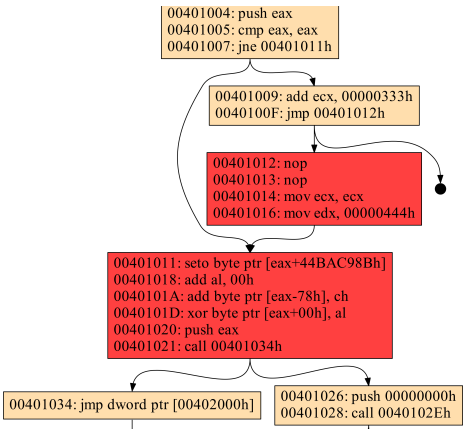

Figure 2.3: Part of our disassembler's output for the example in listing 2.1.

2.4 Instructions-based detection: Turning attackers' strength into weakness

Due to obfuscation techniques such as opaque predicate, the control flow graph (CFG) and the sequence of instructions extracted from obfuscated programs are usually convoluted, resulting in different sizes of basic blocks compared to a normal program, a greater percentage of sink vertices of all basic blocks, and other telltale features. In the following subsections, we introduce interesting features that can effectively identify an abnormal control flow graph and sequence of instructions. We show how each of these features differs in case of obfuscated and clean files. The illustrative statistical distributions presented in this section are from representative file sets that are also used in the experiments of Section 4.5.

2.4.1 Percentage of sink vertices in CFG

The CFG of a given program is a digraph where each vertex represents a basic block. Sink vertices in this context refer to those vertices with zero out-degree. Sink vertices are usually the exit point

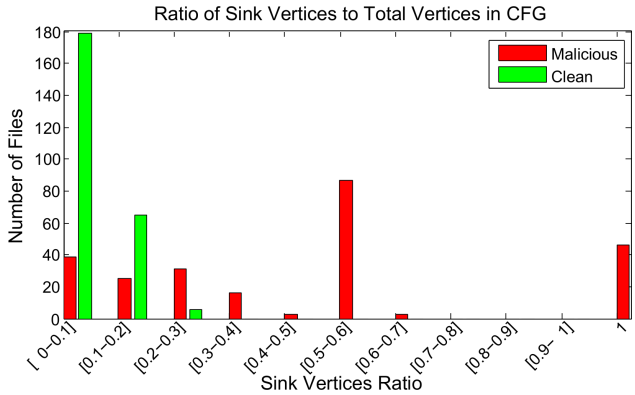

Figure 2.4: Distribution of sink vertices to all vertices ratio for malicious and clean file sets.

of the program, and since a typical program has few exit points in the code, the number of sink vertices is very small compared to other vertices. Obfuscated malware that employ anti-analysis techniques lead to inaccurate static disassembly of the file. Thus, the ratio of sink vertices to the total vertices becomes different from a normal file. Figure 2.4 compares the ratio of sink vertices in both clean non-obfuscated and malicious obfuscated files, respectively.

2.4.2 Percentage of the size of referenced instructions to the entire size of the section

Due to code obfuscation, encryption or packing, the size of referenced instructions compared the size of the section is relatively smaller than that of clean files. The decryption or unpacking routine that exists in the same section of the encrypted or packed code occupies a much smaller size than the actual payload of the file. This fact represents a distinguishable feature between packed and non-packed files. Figure 2.5 shows these values in different files in clean and malicious dataset.

2.4.3 Average number of instructions in basic blocks

After constructing the control flow graph of the program, each basic block will represent a set of instructions with a single entry and a single exit instruction. The exit instruction, in most cases, is a control transfer that affects the flow of the execution. If the disassembly was wrongly redirected into disassembling packed or encrypted data due to anti-disassembly tricks, false instructions will be decoded, which will result in different characteristics of a typical control flow graph of a normal

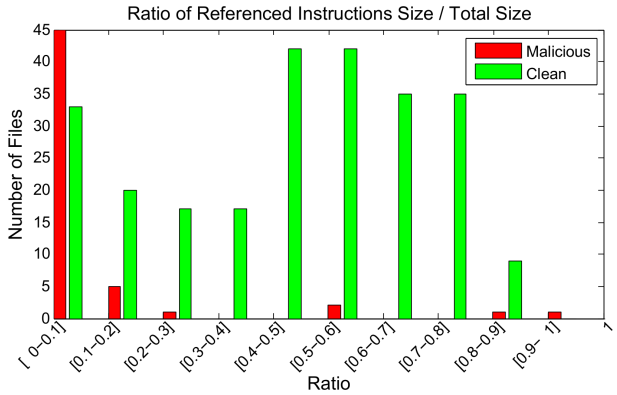

Figure 2.5: Distribution of referenced instruction size to section size ratio for malicious and clean file sets.

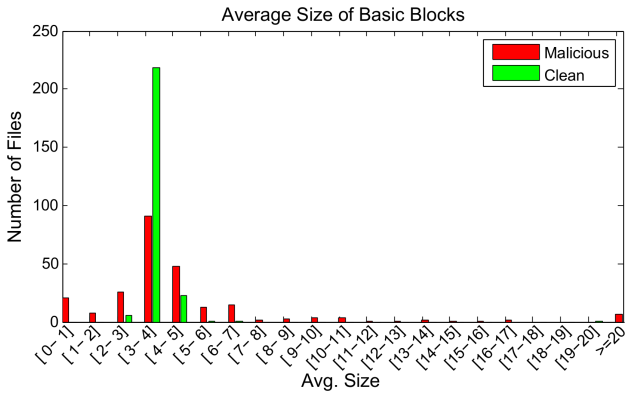

Figure 2.6: Distribution of basic block average size for malicious and clean file sets.

application. One of these characteristics is the average size of instructions in basic block. Figure 2.6 shows the average number of instructions in a basic block in malicious and benign dataset respectively.

2.4.4 Entropy of referenced instructions opcodes

As discussed in Section 2.2, entropy is a measure of randomness which can sometimes be used to detect packed files. Almost all techniques that use entropy to detect packed files calculate the entropy of the entire file, a section, or the file header. However, as explained earlier, an encrypted or packed data can still exhibit low entropy if an entropy reduction method is used. In addition, a normal program could contain data of high entropy within the code. In this case, the entropy of this data will be incorporated in the total entropy. This is a major source of false positives.

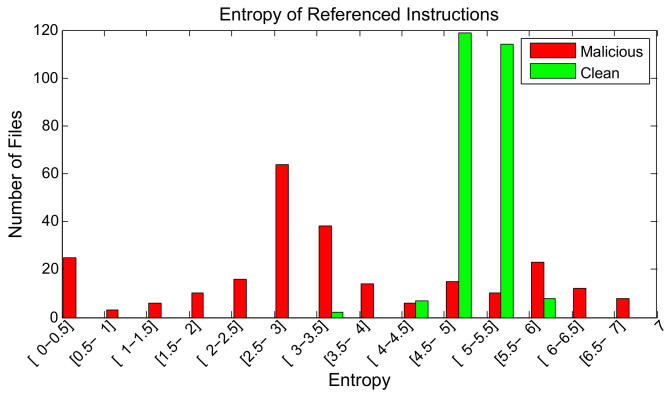

Figure 2.7: Distribution of file entropy for referenced instructions only.

On the other hand, if the program employs an anti-disassembly technique that is able to deceive the disassembler into decoding false instructions, the resulting opcodes of the false instructions will have different statistical distribution than those of real ones. If the entropy of only referenced instructions is computed, we would have a more specific and accurate use of the entropy metric. Thus, even if a normal program contains data of high entropy within the code, the entropy of this data will not be incorporated in the total entropy calculation, because the flow of execution of a normal program ensures jumping over this data during execution. Figure 2.7 shows the distribution of file entropy when only referenced instructions are considered, for both non-obfuscated clean and obfuscated malicious files, respectively.

2.4.5 Existence of interleaving instructions

In our system we flag any file with interleaving instructions as obfuscated, since unobfuscated applications do not intentionally employ opaque predicate. Existence of such interleaving instructions is a clear flag of obfuscation, unless it is a bug or an artifact in an unobfuscated program. In Section 4.5, we show how our system found an artifact in non-obfuscated Windows files when interleaving instructions were detected in them.

2.4.6 Existence of unknown opcodes

If an unknown opcode is encountered while disassembling the file, it means that the disassembly process is diverted from the normal execution path, and the file is flagged as obfuscated. It is worth mentioning that our disassembler does not make assumptions about indirect addressing. Assumption in this case would be uncertain and following uncertain paths would definitely lead to high false-positive rate. Therefore, since the disassembler covers almost all opcodes of the x86 and x86-64 architecture, even some of undocumented instructions, finding an unknown opcode in the extracted instructions would be a strong evidence of obfuscation.

2.5 Experiment and evaluation of our detection method

We ran our proposed system on two sets of files. The first set consists of 250 clean files taken from a clean Windows XP 32-bit machine, all of them non-packed. The second set consists of 250 malicious packed files, packed by both commercial and custom packers. The ranges of values for the features introduced in Section 2.4 are shown in Tables 2.1 and 2.2 for both clean and malicious sets, respectively. Based on the result in Table 2.1, we established six conditions to test if the file is packed or not:

1. The sink vertices ratio lies within the range of non-packed files.
2. The referenced instructions ratio lies within the range of non-packed files.
3. Average number of instructions in basic block lies within the range of non-packed files.
4. Entropy of referenced instructions lies within the range of non-packed files.
5. The code has one or more anti-disassembly tricks.
6. The code references an unknown opcode.

Based on these conditions, we achieved 100% correct detection of the clean files as non-packed and 98.8% of the malicious files as packed or obfuscated. This result is shown in Table 2.3. In

Table 2.1: Value ranges of statistical features in Windows XP clean file set.

Property	Min	Max
Sink vertices ratio	0.0260047	0.254545
Referenced instructions ratio	0.000544839	0.884181
Average number of instructions in basic block	2.3125	19.6357
Entropy of referenced instructions	3.38	5.61
Files with a referenced unknown opcode	0	
Files with anti-disassembly trick	0	

Table 2.2: Value ranges of statistical features in malicious file set.

Property	Min	Max
Sink vertices ratio	0.0	1.0
Referenced instructions ratio	1.89169 x 10^{-6}	0.92139
Average number of instructions in basic block	1	2142
Entropy of referenced instructions	0	6.81
Number of referenced unknown opcode	0	5
Files with anti-disassembly trick	63	

addition, we found that by adding an extra criterion by measuring the entropy of the entry point section and marking files with entropy greater than 6.5 as packed, we could achieve 100% detection of malicious files as packed, i.e., 0% false negatives. However, this introduced a 14.8% false positive rate as some of the clean non-obfuscated files were flagged as obfuscated.

Although structural features of the files were not our main concern in this reseach, we added a few checks on the file structure which further improved the result. The following list of structural features were used to help identify obfuscated files:

1. The entry point is in file header before any section.

2. There is no `.text` or `CODE` section in the file.

3. The entry point is in the last section while it is neither `.text` nor `CODE` section.

Table 2.3: File set analysis results.

	FN	FP	Correctly detected	Percentage %
Instructions-based features only				
Clean files, WinXP	0%	0%	250 / 250	100%
Clean files, Win7	0%	0.2%	416 / 417	99.8%
Malicious files	1.2%	0%	247 / 250	98.8%
Instructions-based features with checking entropy of entry point section				
Clean files, WinXP	0%	14.8%	213 / 250	85.2%
Clean files, Win7	0%	13.2%	362 / 417	86.8%
Malicious files	0%	0%	250 / 250	100%
Structural features only				
Clean files, WinXP	0%	0%	250 / 250	100%
Clean files, Win7	0%	0%	417 / 417	100%
Malicious files	36.8%	0%	158 / 250	63.2%
Instructions-based with structural features				
Clean files, WinXP	0%	0%	250 / 250	**100%**
Clean files, Win7	0%	0.2%	416 / 417	**99.8%**
Malicious files	0%	0%	250 / 250	**100%**

4. `SizeOfRawData` $= 0$ and `VirtualSize` > 0 for some sections.

5. Sum of `SizeOfRawData` field of all sections is greater than the file size.

6. Two or more sections overlap.

7. The file has no imports at all or the import table is corrupted.

The scanning result when using each detection features is shown in Table 2.3 where FN and FP refer to false negative and false positive rates, respectively.

We collected 423 clean non-obfuscated files from a clean Windows 7 32-bit Home Basic Edition and scanned them using only our instructions-based condition. We used the ranges mentioned

in Table 2.1 as detection conditions. There was 7 out of the 423 files (1.64%) detected as obfuscated. Since we know that those files are not obfuscated, we considered, at the beginning, the result as a false-positive. However, after manually reverse engineering the files, it turned out there is an artifact of some incomplete code generation [19] in six of them. The files have overlapped instructions that, if executed, would likely crash the programs under certain conditions. Although these conditions were not clear to us, based on the instructions' location in the file, we feel that is unlikely that that execution of these faulty instructions would ever take place [19]. The seventh file is `sppsvc.exe` that has `Referenced Instruction Ratio` $= 0.000273778$, which is less than the minimum boundary set in Table 2.1. Therefore, since the first six files contain an artifact, we could safely exclude them from the set and consider that there was no false-positive in our results except that corresponding to `sppsvc.exe`. On the other hand, when we combined both instructions-based and structural-based conditions, we had five more false-positives, flagged because they do not have any imports. Finally, we note that when we used entropy for detection, it led to the worst result as some non-obfuscated files show high entropy in the code section. The full results of this analysis of the file sets are shown in Table 2.3.

We ran another test on a larger set of 10,171 malicious files. The set is a collection of live malware given to us by a security firm. Unfortunately, we have not been given details about the set in terms of packing/obfuscation. Although we admit that scanning result of this set is not a concrete measure of the effectiveness of the system since we cannot give a confirmed value of false-positive or false-negative, we opted to show the result for the sake of illustration. Table 2.4 shows the value ranges of each condition, while Table 2.5 shows the result of scanning the large malware set.

For all of our experiments, we observed that the system was able to process files at an average rate of 12 ms each.

Table 2.4: Value ranges of statistical features in large set of malicious files.

Property	Min	Max
Sink vertices ratio	0.0	1.0
Referenced instructions ratio	0.62 x 10^{-6}	1.0
Average number of instructions in basic block	1	69600
Entropy of referenced instructions	0	7.23
Number of referenced unknown opcode	0	163
Files with anti-disassembly trick	1835	

Table 2.5: Analysis results from the large set of malware.

	Detected as packed	Percentage %
Instructions-based features only	9982 / 10171	98.14%
Instructions-based with structural features	10161 / 10171	**99.9%**

2.6 Discussion and limitations

We can summarize the features used in our system into two categories. Instructions-based statistical features, and structural features. The file structure features have the same advantages and limitations of the previous research discussed in Section 2.2. The major contribution of the chapter is the instructions-based method of detection.

All the features mentioned in Section 2.4 except the one in Subsection 2.4.2 are useful metrics when the file under consideration has features to intentionally deceive the disassembler into decoding wrong execution paths. This is due to existence of anti-disassembly tricks or other control flow obscuring techniques.

On the other hand, if the file is packed or encrypted with no control flow obfuscation, the feature discussed in Subsection 2.4.2 (`Referenced Instruction Ratio`) comes into play. It can detect that just a small portion of the section is executed, which is typically the case when a small routine is responsible for unpacking or decrypting the relatively large remainder of the file. However, the limitation in this case is when this small routine exists in a separate section from the

code to be unpacked. In this case, the section containing the unpacking routine would contain just the referenced instructions of the unpacking routine, and thus its ratio will be high. Hence, our detection would likely be evaded if a file is packed such that the unpacked routine and the packed code exist in two different sections, the file does not affect the header in a distinguishable way, and does not have anti-disassembly tricks.

Finally, the proposed system cannot yet analyze .NET and Java files because these are represented by an intermediate language which needs other methods of disassembly. The system was developed in C++ and it uses the BeaEngine library [9]. The experiment was conducted under Windows 7 64-bit on a notebook with Intel Core i5 processor and 8GB of RAM. The average execution time was observed to be around 12 ms per file.

2.7 Conclusion

Due to the high number of malware being produced every day, the need for a fast and efficient system detection persists. If there is an efficient, fast way to detect the presence of obfuscation in a sample and then move it to a more rigorous test, this would reduce some of the burden on the more costly methods and help keep up with the big number of samples.

This chapter presents a generic heuristic method to detect obfuscation based on both the structural and instructions-based features of the file. We built a complete recursive traversal disassembler for x86 and IA-64 binary files. We were able to detect the instructions overlapping trick and presence of unknown opcodes, which are mainly symptoms of opaque predicate or a bug in the code. In addition, a number of statistical features based on the control flow graph and the instructions that help distinguish malicious and benign files have been presented. When measuring those features combined with structural features of a sample, we achieve very high detection result of obfuscated files with a very fast scanning time of 12 ms on average per file.

A key advantage of our method is that it is not limited to a certain type of packers or a specific obfuscation technique. In our future work, we plan to add more instructions-based features and incorporate machine learning techniques to classify different packers. We believe that if these

future goals are accomplished and the limitations mentioned in 4.6 are overcome, they would lead to more accurate results with less margin of error.

Chapter 3: MULTI-CONTEXTS CHARACTERIZATION OF SOFTWARE FOR MALICIOUS PROGRAMS DETECTION

3.1 Introduction

Malicious software, often referred to as malware, is a program designed to do harm to individuals, corporations, or governments. Currently, malware is used in a wide range of attacks, from propagating through the network, launching Denial of Service (DoS) attacks against web servers, causing service disruption and stealing credit cards information to more sophisticated attacks against nuclear plants [73]. In addition, financial malware is increasing in number and having a stronger impact each year. For example, the Carbanak malware was able to steal $1 billion from over 100 financial institution worldwide in the biggest bank heist in history [28].

Each year a massive number of malware is produced and spread, and anti-virus vendors struggle to keep up with this high number by producing more signatures and work to maintain timely updates to their customers. In the last quarter of 2014, McAfee received more than 350 million malware samples in total, 50 million of them were new malware. That means there are 387 new threats every minute [29].

Despite the variety of anti-malware solutions ranging from host-based to network-based, malware detection remains an open problem. The primary contestant to solving the issue is that malware attacks continuously change. Malware programs employ several techniques to avoid detection and to evolve rapidly compared to detection solutions. Techniques that evade signature detection exist, such as simple encryption, to polymorphism, or advanced metamorphism [44]. Other techniques target behavior analysis via hook detection [31] and virtual machine detection [39], or even using mimicry attack to deceive the behavioral analysis system by showing benign behavior [72].

A common technique used for malware detection is String signature. Nevertheless, the technique is not efficient against new malware samples and their variants. Other techniques exist which depend on collecting the behavior data of a program to determine if it is malicious or not, but these

methods suffer from incompleteness of their coverage to the software execution paths. Another direction is to gather data from the file header, such as the number of sections, size of each section, entry point location, etc. This technique is very fast compared to the former two, however, it could produce a high rate of error if the file is obfuscated. Although the problem of malware detection has been discussed repeatedly in both academia and business settings, it is evident that more advanced solutions are needed and more research is encouraged to find mitigation to this serious threat.

Contributions of the research presented can be summarized as: Defined and introduced a large composite set of features that combines both static and dynamic aspects of the sample. More than 32,000 features compose the feature vector for sample classification in our experiment. This makes it very difficult for an attacker to obfuscate code in such a way as to evade this number of features. Utilized Random Forest and Naive Bayes machine learning techniques to achieve a high detection rate on a large corpus of recent and diverse malicious files first seen by VirusTotal [67] during the period from December 2014 to April 2015. 1,000,000 samples were used for the experiment, divided equally between malicious and clean files. To the best of our knowledge, this is the largest number of files used in an experiment in academic literature for malware detection. With this comprehensive set of features and the large number of files, a training model that led to a high detection rate was able to be produced. The model achieves accuracy up to 96.7% with a false positive rate of 2.1% on a very large and diverse set of files, while working practically well in detecting malware in future months. Evolution of both the features and malware families across five months were investigated. The evolution of malware features across months is explained, and a description of how that is correlated to malware campaigns is provided.

The rest of the chapter is organized as follows: In Section 2 we summarize the related literature in this area. Section 3 describes our features set in details while Section 4 illustrates our methodology of constructing the feature vector. In Section 5 we outline the system architecture, describe its components and the dataset to be used in the experiments. Section 6 describes the different experiments and compares the results. Section 7 discusses the results shown in the preceding section

and describes the evolution of features and malware families distribution during the course of five months; and finally conclusions are given in Section 8.

3.2 Related Work

There have been many different methods used to protect proactively against malicious programs by detecting zero-day malware. The following subsections discuss some of these methods.

3.2.1 Detection based on Dynamic Features

Some papers introduced techniques that detect malicious activities by monitoring and logging performance-specific registers of the processor, such as [14, 27, 61], where [61] used the hardware performance information to detect anomalies in execution and hence can detect exploitation of legitimate software. Other publications use instruction traces during execution such as [59] and [24]. Others used text strings displayed to the user and compared their semantic to the behavior of the file to determine if the malware is carrying on a stealth activity [23]. Salehi et al. [45] used the APIs and their arguments that the sample uses during execution as the feature vector. Other research studies used temporal and spatial information of the API usage [3], that is, similar to [45], both the APIs and their arguments are collected, however, their execution order is considered as well. A similar research done by Miao et al. [33] also utilized API names extracted during sample execution and constructed two layers of information, one that considers API names, while the other layer is more high level and interprets a sequence of API to infer high level behavior. In addition, Tian et al used API sequences collected during runtime and then applied pattern recognition algorithms on the extracted pattern [62].

3.2.2 Detection based on Static Features

Several types of information can be extracted from the file header, such as imported shared libraries, API names, number of sections, etc. These features could be helpful in providing an overview of the file. Many research studies have been done to detect malware based on this in-

formation, such as [38, 50, 53, 63] which used the PE header and structure information to detect packed files. Other PE header features are also used to detect zero-day malware [55,56]. A seminal work of Shultz *et al* [52] used data mining techniques to detect malicious software. The research used the list of DLLs used by the binary, the list of DLL function calls made by the binary, and the number of different functions within each call, all extracted from the PE header. In addition, strings extracted from the file were used as features as well as the byte sequence. Other techniques used byte n-gram to build a classification model [26], which relied on the file contents. But still, the techniques are prone to evasion via obfuscation. Ahmadi et al. [2] classified 21,741 malicious files from 9 different families with a diverse set of static features such as Entropy, imported APIs, strings, sections, instructions, n-gram of hex values of the file contents, etc. The research claimed that it is capable of classifying obfuscated files. However, the experiment is limited in terms of the number of clusters. Ding et al. [15] on the other hand, extracted the instructions from the statically generated control flow graph. After extracting the instructions, the opcode is extracted and then used by the n-gram method. Other interesting research was recently done by Invincea Labs [51] for malware detection using Neural Networks. The authors achieved 95% detection rate with 0.1% false positive when they used static features in an experiment consisting of 350,016 malicious and 81,910 benign files. The features used were the imported DLL names, each DLL's imported functions, Entropy, byte distribution and other 256 features of metadata extracted from the header.

In general, although static analysis is still considered as prone to evasion if the file is obfuscated/packed, it is much faster and promising if combined efficiently with machine learning.

3.2.3 Detection based on Hybrid Features

There is a number of publications that use both dynamic and static features for malware detection. Yan *et al*. [71] used PE header information and n-gram of instructions opcode as static features. For dynamic features, they used n-gram of instruction traces opcode plus the list of invoked system calls. The authors excluded the packed files from their dataset and considered only those files that were not flagged by PEiD [5] as packed. Ravula *et al*. [40] gathered dynamic features of added/re-

moved/modified registry keys, whether the sample accesses the internet or not, what DLLs and APIs are used during execution and if the sample accesses another directory. For static features they considered the packer name by using PEiD [5], the programming language used, list of unique strings and URLs embedded in the files. The total number of features with list of APIs was 141 features. Santos *et al*. [47] collected the frequency of occurrences of operation codes that were obtained statically. For dynamic analysis, information was gathered about file activities, some protection mechanisms of the sample, whether the sample is persistent, network activities, process manipulation, whether the sample retrieves information about the system, and unhandled exceptions during execution. The number of dynamic features were 63 while the opcode-sequences frequencies led to 144,598 features that were reduced later to 1,000. Other interesting research is [6], in which the authors used the raw byte sequence of the binary file, sequence of instruction traces, sequence of the statically disassembled instructions, the control flow graph, and dynamic system call traces. The authors used 1,556 files (780 malicious and 776 benign) as the training set, whereas the test set contains 20,936 of only malicious files.

3.3 Features Definition

We collect an extensive set of the malicious sample features and integrate them to build a feature vector for machine learning techniques. The features are collected from different aspects of the malicious samples. As every aspect of malware can be prone to obfuscation, data collected based on one sole aspect could be inaccurate. Therefore, we collect several features from many aspects of a malicious file, and use these to build our feature model for describing a malware sample. These feature sets include dynamic analysis of the file when it runs in a controlled environment, static analysis of the file header features, and data based on the instructions that compose the executable file.

Static features consist of both fixed and variable length features. There are 16 fixed length static features typically including the number of file sections, number of imported APIs and whether the file is signed or not. Variable length features are the names of sections, the list of imported APIs,

and the list of imported modules names. Three arrays of variable length features account for 4,500 different values, representing features existing across samples in a certain month. For example, the array of imported module names contains 1,500 entries, which are different names found across 200,000 samples gathered from December 2014. Each value is represented as a bit in the feature vector. The total length of the static feature vector for the month of December 2014 is 4,516.

Dynamic features, on the other hand, consist of 22 scalar-value features, such as whether the file uses TCP connection, uses API hooking (Yes/No). The variable length features consist of 31 long array of values representing, for example, list of Hooking types used, HTTP methods, and runtime loaded DLLs. The total number of features represented in the vector of dynamic features for the month of December 2014 is 27,552. Utilizing the technique presented in [43], a determination is made as to whether the file is obfuscated or not. This information is based on the control flow graph and distribution of instructions. The obfuscation feature is presented as one bit in the feature vector.

Combining these three sets culminates a feature vector containing all the features. For a sample belonging to December 2014, this vector consists of 32,069 features.

In the following subsections, we describe features extracted from each context. For the full list of features, please refer to Table 3.2 and Table 3.3.

3.3.1 Instructions Features

Using the technique discussed in [43], information about the control flow graph of the file is obtained. Based on the control flow graph features, it is determined whether the file is obfuscated, including whether the file is using anti-analysis tricks on the instruction level. A boolean flag is used in the feature vector to indicate whether the file employs any obfuscation techniques. The flag is named `"Is Obfuscated"` in Table 3.2.

3.3.2 Dynamic Features

There are many features extracted from the program's dynamic behavior.

Network Activity

A number of features that indicate network activity initiated by the malware sample are extracted. For example, UDP and TCP requests are logged and both the IP and PORT are extracted. Also, DNS requests are used as features, with both the IP and host name collected. In addition, all HTTP requests are collected, and for each request, the User-Agent, Destination URL and Request Method are considered features.

Services

Service activities are collected, as some malware programs register one or more services to run in the background once they penetrate the system.

Process Manipulation

We collected a number of features related to processes dealt with by the malicious samples during execution, such as whether or not the sample created new processes, process tree name, terminated processes, as well as process injection information.

Process injection is a highly utilized technique that allows malware to task legitimate processes do the work for them. For example, if the malware needs to have network communication, but the firewall on the system will deny its request the malware may look for some known processes that likely have an "allow" rule in the firewall, such as web browsers, and then inject their malicious code to get executed in the context of the legitimate application. A list of injected process names was collected, as well as a number of features related to processes dealt with by the malicious samples during execution, i.e. such as whether or not the sample created new processes, process tree names, terminated processes, as well as process injection information. Additionally, a list of shell commands executed by the sample is collected.

Files

File use is very important for almost every sample. A malicious program could be a virus that infects other executables. In this case it will look for executable files in the system, open them, and write itself into their code. A trojan could also download other malware programs from the internet. Several types of information about the files created, written, deleted or modified by the sample under analysis are gathered.

Registry

The system registry is commonly used by processes to save or modify system configurations. Many malicious applications use registry manipulation to achieve their goal. Registry related behavior such as creation, modification and deletion of registry keys or values is noted.

Mutex Objects

Mutex objects are usually used in multi-threaded applications to control and coordinate shared resources access. Mutexes can have names when created by the application, and the list of the newly created and opened mutexes during the sample runtime are also collected.

Runtime DLL

An executable file can declare the DLLs to be used in the import table, or load the required DLLs during runtime. Usually malware use the runtime DLL technique to hide behavior. The list of DLLs loaded during runtime is collected and considered as part of the feature vector.

Windows Manipulation

The list of searched Windows names and Windows class name are gathered. These values are usually used when the sample calls APIs such as `FindWindow` and `RegisterClass`.

Shell Commands

A list of shell commands executed by the sample is also collected.

Extras

The VirusTotal report provides some extra information regarding the use of device driver, usually whether the file is using the API `DeviceIoControl`, or detecting a debugger presence using `IsDebuggerPresent`.

3.3.3 Static Features

The file structure can be indicative of the type of file, and some research studies are based solely on file structure for malware detection, as mentioned in Section 3.2. Many features from the file header are compiled and used in our final feature vector.

Imported Modules and API

Many research studies are based on collecting imported API names, as mentioned in Section 3.2. The number of modules and the number of APIs are gathered as two fields in our feature vector. In addition, the list of imported module names and APIs names are collected.

File Sections

Every executable file contains a number of sections for different types of contents. Usually *.text* section contains executable code, *.data* section contains data, *.rsrc* for storing program's resources, etc. Some crafted files, usually malicious, can contain unusual section names. In addition, packed files might also get smaller number of sections than unpacked files, and have specific section names. For example, UPX packer [66] almost always puts sections with the names "UPX0", "UPX1", etc. From every file in our set, we collected section names, file alignment [34] and section alignment [34].

EntryPoint

The address of the first instruction to be executed.

Machine Code

The machine code in the file header. This code indicates the type of the machine or the system the file should run on. Usually a file compiled to run on Intel 386 or later processors will have the value of `0x14c` [34].

Image Base

Address in memory where the executable file will be located upon execution.

Compile Timestamp

Timestamp indicating file compilation date.

Link Date

Timestamp indicating file linking date.

Size of Headers

Size of the "optional header" which is, unlike what the name implies, required for PE executable files [34].

Characteristics

Flags that indicate the attributes of the file, such as if the file is a DLL or not, 32 bit or 64 bit, etc.

Number of Data Directories

Number of data-directory entries in the optional header.

File Size

Total file size in bytes.

Is Suspicious

A value that is set to non-zero if any of the following conditions is met:

- The entry point is in the file header before any section.
- There is no .text or CODE section in the file.
- The entry point is in the last section while it is neither .text nor CODE.
- SizeOfRawData = 0 and VirtualSize $>$ 0 for some sections.
- Sum of SizeOfRawData field of all sections is greater than the file size.
- Two or more sections overlap.
- The file has no imports at all or the import table is corrupted.

These checks were used in our previous work [43] and led to improvement of the detection results.

Digital Signature

Whether the file is signed or not in our is also considered for the feature vector. Although some recent malware samples are signed with stolen private keys, the vast majority of signed software is benign.

3.4 Methodology

The feature set consists of two types of features, variable and fixed size. The variable size features are the set of features that do not necessarily exist in every file. For example, HTTP host name that the sample connects to, or API and module names imported. Since not all files have these

Table 3.1: Top Ten Module Names.

	Module Name	Number of occurrences
1	KERNEL32.DLL	85,046
2	USER32.DLL	69,724
3	ADVAPI32.DLL	58,098
4	OLEAUT32.DLL	37,622
5	OLE32.DLL	34,280
6	GDI32.DLL	34,092
7	SHELL32.DLL	31,664
8	MSVCRT.DLL	24,642
9	COMCTL32.DLL	20,677
10	SHLWAPI.DLL	17,412

features, we select an initial set of files, namely the training set which consists of 100,000 files for each malicious and clean category. The files in the training set were first seen by VirusTotal in the specified month. All the variable size features are enumerated from this set and a table is assembled with each different value. For example, Table 3.1 shows a subset of module names extracted from the 100,000 malicious files in our training set for December 2014, with the corresponding occurrence frequency, i.e. number of malicious files where a specific module name is found.

In this example, if we select only the ten module names in Table 3.1, we will have a vector of size 10. Then, every other file we examine will have a binary vector of size 10, the value of 1 in the vector means that the file is using the corresponding module, while 0 means it does not. For instance, if a file uses the modules KERNEL32.DLL, LIBC.DLL, SHELL32.DLL, MSVCRT.DLL, and SHLWAPI.DLL. It will have a feature vector of [1, 0, 0, 0, 0, 0, 1, 1, 0, 1]. The module LIBC.DLL was used by the file, but it is not on our base features, so it will not be a part of the vector. The same process is done for the other features. The value of each feature in the resulting vector is a boolean value. For the full list of variable features, please refer to Table 3.3.

On the other hand, fixed length features will not be matched to any base features. Typically, fixed length features include $EntryPoint$, $NumberofSections$, $FileAlignment$, etc. The value of each feature in the resulting vector is an integer or boolean value. For the full list of fixed features, please refer to Table 3.2.

Table 3.2: Fixed Length Extracted Features.

	Feature	Report		Feature	Report
1-	Num of imported modules	Anatomist	2-	Num of imported APIs	Anatomist
3-	Entry point	Anatomist	4-	Machine	Anatomist
5-	Image Base	Anatomist	6-	Compile Timestamp	Anatomist
7-	Size of Headers	Anatomist	8-	Section Alignment	Anatomist
9-	File Alignment	Anatomist	10-	Num of Data Directories	Anatomist
11-	File Size	Anatomist	12-	Num of Sections	Anatomist
13-	Is Suspicious	Anatomist	14-	Is Obfuscated	Anatomist
15-	Link Date	VirusTotal	16-	Is Signed	VirusTotal
17-	Has Hooks	VirusTotal	18-	Has UDP	VirusTotal
19-	Has HTTP	VirusTotal	20-	Has DNS	VirusTotal
21-	Has TCP	VirusTotal	22-	Does Service actions	VirusTotal
23-	Does Process Termination	VirusTotal	24-	Does Process Injection	VirusTotal
25-	Does Process Creation	VirusTotal	26-	Does Modify hosts file	VirusTotal
27-	Does Window Search	VirusTotal	28-	Has Runtime DLLs	VirusTotal
29-	Does Open Mutex	VirusTotal	30-	Does Create Mutex	VirusTotal
31-	Does Delete Registry	VirusTotal	32-	Does Set Registry	VirusTotal
33-	Does Open a file	VirusTotal	34-	Does Move a file	VirusTotal
35-	Does Download a file	VirusTotal	36-	Does Replace a file	VirusTotal
37-	Does Delete a file	VirusTotal	38-	Does Copy a file	VirusTotal

3.5 System Architecture

In this chapter, we focus only on Windows PE executable files. Due to hardware and time constraints, we chose one million files on which to conduct our experiment, which is, to the best of our knowledge, the largest number of files used in academic literatures for the topic of malware detection. We excluded Adware from consideration. Adware is defined as computer programs that display unwanted ads without user consent or permission. Adware could be manufactured by legitimate companies, such as Lenovo [11], so detecting adware programs accurately needs a different approach and should not be treated in the same fashion as other malicious applications. We obtain both malicious and benign binary files, along with the scan report of each file from VirusTotal. The malicious files in our experiment is defined as those files that were flagged by 20 or more antivirus programs, including Microsoft, Kaspersky, Mcafee and Bitdefender anti-malware. Since Kaspersky, Mcafee and Bitdefender were among the best tools in regards to detection rate [7], and

Table 3.3: Variable Length Extracted Features.

	Feature	Report		Feature	Report
1-	Modules	Anatomist	2-	APIs	Anatomist
3-	Section Names	Anatomist	4-	Hook Type	VirusTotal
5-	Hook Method	VirusTotal	6-	HTTP URL	VirusTotal
7-	HTTP Method	VirusTotal	8-	HTTP User Agent	VirusTotal
9-	DNS IP	VirusTotal	10-	DNS hostname	VirusTotal
11-	UDP IP:PORT	VirusTotal	12-	TCP IP:PORT	VirusTotal
13-	Opened srvc names	VirusTotal	14-	Opened srvc mngr db	VirusTotal
15-	Opened srvc mngr machine	VirusTotal	16-	Extra	VirusTotal
17-	Shell cmd	VirusTotal	18-	Proc Tree Name	VirusTotal
19-	Injected Proc Name	VirusTotal	20-	Created Proc Name	VirusTotal
21-	Terminated Proc Name	VirusTotal	22-	Searched Windows Class	VirusTotal
23-	Searched Windows Name	VirusTotal	24-	Runtime DLLs	VirusTotal
25-	Opened Mutex Name	VirusTotal	26-	Created Mutex Name	VirusTotal
27-	Set Reg Entry Type	VirusTotal	28-	Set Reg Entry Key	VirusTotal
29-	Set Reg Entry Value	VirusTotal	30-	Deleted Reg Entry Key	VirusTotal
31-	Opened Files path	VirusTotal	32-	Read Files path	VirusTotal
33-	Written Files path	VirusTotal	34-	Deleted Files path	VirusTotal

Microsoft was the best in regards to false-positive rates at the time of writing [7], we made sure all malicious files in our experiment were detected by all four anti-malware programs. On the other hand, benign files are considered those that are not flagged by any of the 56 antivirus systems at VirusTotal. We understand that even though each file was tested by numerous antivirus systems and found to be clean, there is still a small possibility that the file is obfuscated or a zero-day malicious program. However, multi-antivirus scanning is the only feasible and automated way we found to check whether the file is benign given our large data set.

The files were first seen by VirusTotal during the period from December 2014 to April 2015. We collected 100,000 samples of both malicious and benign files each month from December 2014 through April 2015, giving us a total of *1,000,000* files.

The scan report from VirusTotal is a JSON file containing static and dynamic analysis data of the file. The static analysis data is obtained using `pefile` [37] tool, while the dynamic analysis is done by a modified version of Cuckoo [46]. VirusTotal lets the binary run for up to 60 seconds before it is terminated [21, 32].

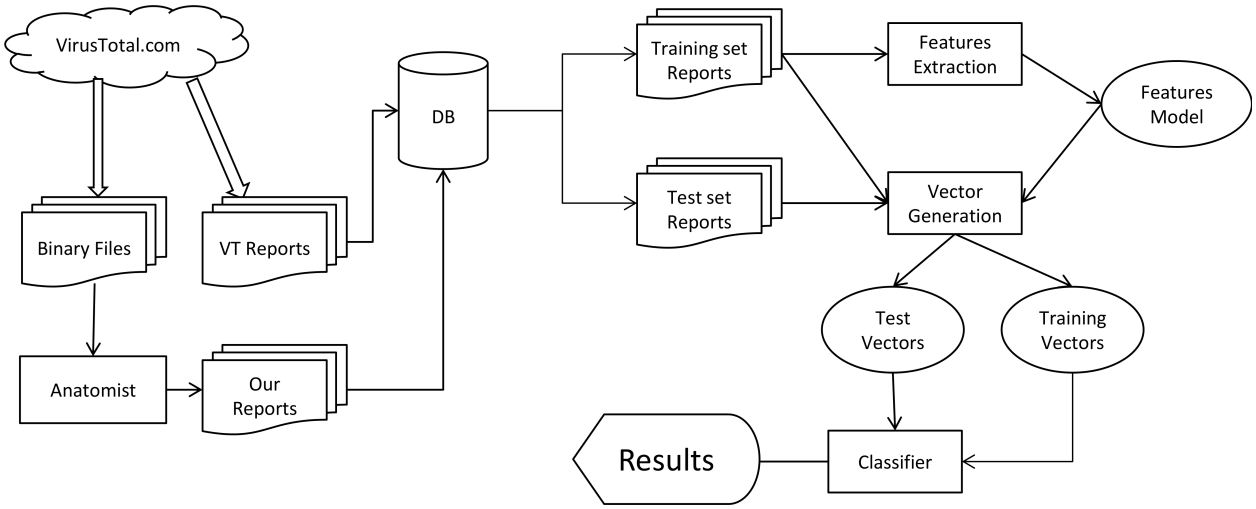

Figure 3.1: Our system architecture.

We developed a PE file scanner called Anatomist in C++. It extracts different types of information from the header of the PE file as well as a list of instructions and its control flow graph, it is also resilient to many tricks usually used by malware to hinder the process of extracting information from the header. An earlier version of Anatomist was used in a previous work to detect obfuscation in binary files [43]. Anatomist outputs a report in JSON format. Both Anatomist and VirusTotal reports are stored in a MySQL database.

Next, both the training and test set reports will be fed to the `Vector Generation` component which will match each report to the `Features Model` and get a feature vector for each file. This step is described in detail in Section 3.4. Thus, `Vector Generation` will generate a list of training and test set vectors. The generated vectors will be the input to the machine learning classifier. We use WEKA [70] as our classification tool.

3.6 Experiment

An experiment E consists of a training and test set. Every month i has its set of files S_i. For each Experiment E_i, the training set is S_i and the test set is $\frac{1}{2}S_{i+1}$. That is, the files involved in the experiment are divided as 66% training and 33% test. So $E_i(S_i, \frac{1}{2}S_{i+1})$ is an experiment of learning from a training data set for month i, and testing the knowledge on data from the next

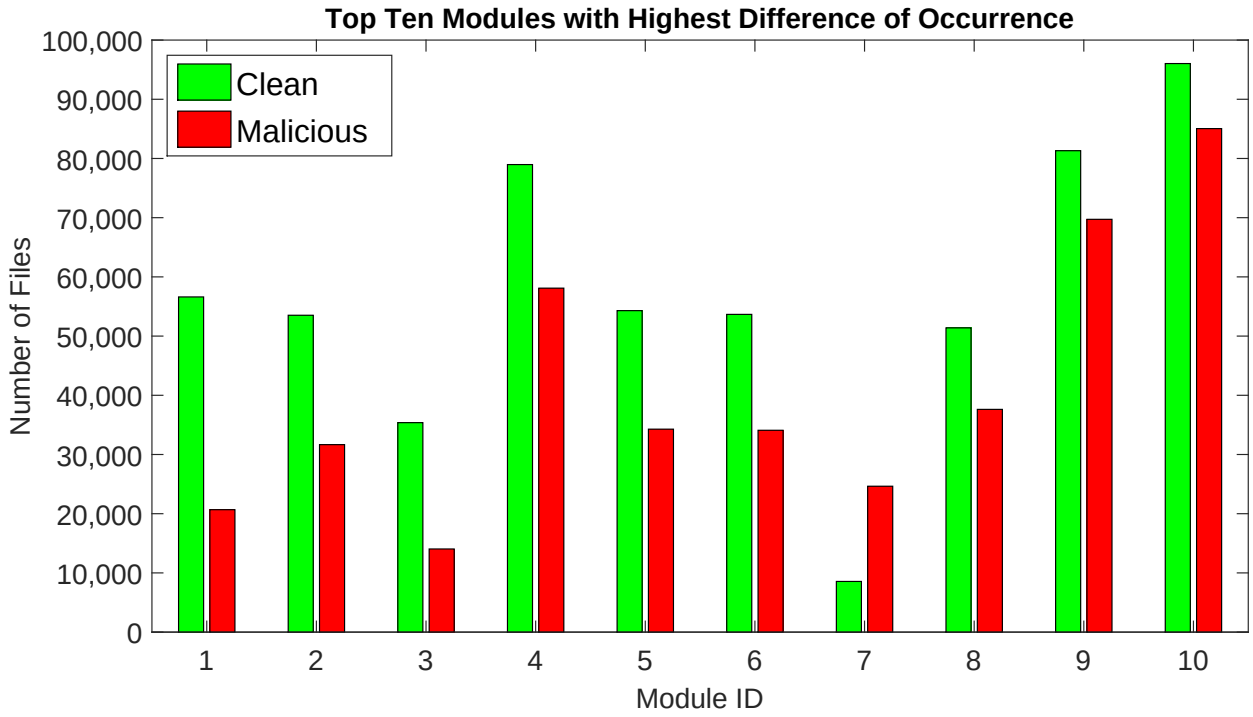

Figure 3.2: Top ten modules with highest difference of usage.

month. The size of each training set S_i = 200,000 files. In the experiment, we have 5 months of data.

Practically speaking, we have a *training set* from which we extracted the variable length base features. The training set is 200,000 files distributed equally between clean and malicious files. We create a feature vector for each file to train the model. After that we use a test set of 100,000 files distributed equally between clean and malicious files, where we match their properties against the features extracted from the training set, and form a feature vector for each file. So in total we have a training set of 200,000 files and a test set of 100,000 files from next month both distributed equally between clean and malicious files.

As an example of our base features, Table 3.4 and Figure 3.2 show the top ten modules that have the highest difference of occurrences in files. For example, the module "COMCTL32.DLL" is used by 56,615 clean files and 20,677 malicious ones, with difference of 35,938. The x-axis in figure 3.2 represents the module ID, which can be found in Table 3.4 that shows the number of occurrences of each module in files.

Table 3.4: Top Ten Modules with Highest Difference of Occurrences between Clean and Malicious Files.

ID	Module Name	# of Clean files	# of Malicious files
1	COMCTL32.DLL	56,615	20,677
2	SHELL32.DLL	53,521	31,664
3	VERSION.DLL	35,379	14,044
4	ADVAPI32.DLL	78,972	58,098
5	OLE32.DLL	54,298	34,280
6	GDI32.DLL	53,667	34,092
7	MSVCRT.DLL	8,559	24,642
8	OLEAUT32.DLL	51,389	37,622
9	USER32.DLL	81,312	69,724
10	KERNEL32.DLL	96,034	85,046

For each performance metric in the columns of the following tables, TP (True Positive) indicates the number of malicious files that were correctly classified as malicious, whereas FP (False Positive) indicates the number of benign files that were misclassified as malicious. Accuracy is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

FN is False Negative, which is the number of malicious files that were incorrectly classified as benign, and TN (True Negative) is the number of correctly classified benign files.

A commonly used metric to evaluate the classification performance in the field of data mining is the area under the receiver operating characteristic curve (AUC), which is utilized here as a performance evaluation criterion.

We chose two machine learning algorithms to conduct our experiments, Naive Bayes and Random Forest. Both belong to a different algorithmic family. Three scenarios were considered in our experiment. First, running the experiment with only static features. Second, we use only dynamic features. Finally, we utilize all the features. In all scenarios, we use those files first seen by VirusTotal in December 2014 as the training set, and test the classifier on each of the subsequent months until April 2015.

Table 3.5: Applying December set to four months' data with different features sets.

		Static Only		Dynamic Only		Combined Features	
		RF	NB	RF	NB	RF	NB
January 2015	TP	95.6%	91.9%	85.1%	42.1%	95.5%	92.1%
	FP	2.8%	40.9%	0.6%	8.5%	2.1%	40.3%
	Acc	96.4%	75.5%	92.3%	66.8%	96.7%	75.9%
	AUC	99.3%	77.5%	95.0%	85.9%	99.5%	78.5%
February 2015	TP	89.6%	92.4%	85.8%	66.5%	94.3%	92.7%
	FP	4.8%	37.2%	2.1%	15.2%	2.8%	36.8%
	Acc	92.4%	77.6%	91.8%	75.6%	95.7%	78%
	AUC	96.9%	78.8%	96.7%	86.3%	98.7%	80.1%
March 2015	TP	94.0%	92.4%	65.8%	65.1%	94.9%	92.6%
	FP	2.5%	34.1%	1.4%	12.1%	1.5%	33.7%
	Acc	95.7%	79.1%	82.2%	76.5%	96.7%	79.5%
	AUC	98.4%	82.0%	94.2%	82.8%	98.8%	82.9%
April 2015	TP	86.3%	90.9%	75.4%	80.1%	88.5%	91.2%
	FP	2.2%	24.4%	1.5%	35.3%	1.2%	24.1%
	Acc	92.0%	83.3%	87.0%	72.4%	93.6%	83.5%
	AUC	98.5%	84.1%	96.9%	77.7%	99.4%	84.8%

Static Features Only

In this scenario, only static features were considered, which are the features from 1 through 16 in Table 3.2 and from 1 through 3 in Table 3.3. The features include `"Is Obfuscated"` which is based on the control flow graph of the instructions. Figure 3.3(a) and Table 3.5 show the performance of Naive Bayes and Random Forest algorithms, referred to as RF and NB in Table 3.5 respectively.

It can be noted from the figure that the accuracy of Naive Bayes is improving on later months. However, the column "Static Only" in Table 3.5 shows that the TP rate has decreased in April respective to January.

Dynamic Features Only

In this scenario, we considered dynamic features; these are the features from 17 through 38 in Table 3.2 and from 4 through 34 in Table 3.3. Figure 3.3(b) and Table 3.5 illustrate the performance of Naive Bayes and Random Forest.

Combined Features

In the final scenario, we considered all features, both static and dynamic. This led to the best classification results. Figure 3.3(c) and Table 3.5 show the performance of the Naive Bayes and Random Forest algorithms.

It is obvious that the Random Forest algorithm outperforms Naive Bayes in most cases. Figure 3.3(d) shows a comparison of Random Forest performance in the three scenarios.

We also conducted an experiment where every month's data is considered as training set and tested on itself. Table 3.6 shows the results of this experiment with both Random Forest and Naive Bayes algorithms. In this experiment, all features were used. Random Forest achieved superior results compared to Naive Bayes. It showed very good results in December, plus it scored 100% accuracy with no errors in the last four months. On the other hand, Naive Bayes scores were not close to Random Forest ones. This is a strong indication on the effectiveness of Random Forest.

Table 3.6: Applying each month to itself (Combined Features).

Classifier	Period	TP	FP	Acc	AUC
Random Forest	Dec 2014	100%	0.07%	99.6%	100%
	Jan 2015	100%	0%	100%	100%
	Feb 2015	100%	0%	100%	100%
	Mar 2015	100%	0%	100%	100%
	Apr 2015	100%	0%	100%	100%
Naive Bayes	Dec 2014	82.2%	40.4%	70.9%	73.6%
	Jan 2015	84.6%	36.4%	74.1%	78.3%
	Feb 2015	92.1%	42.5%	74.8%	75.9%
	Mar 2015	89.5%	33.4%	78.1%	80.7%
	Apr 2015	87.1%	23.8%	81.7%	84.2%

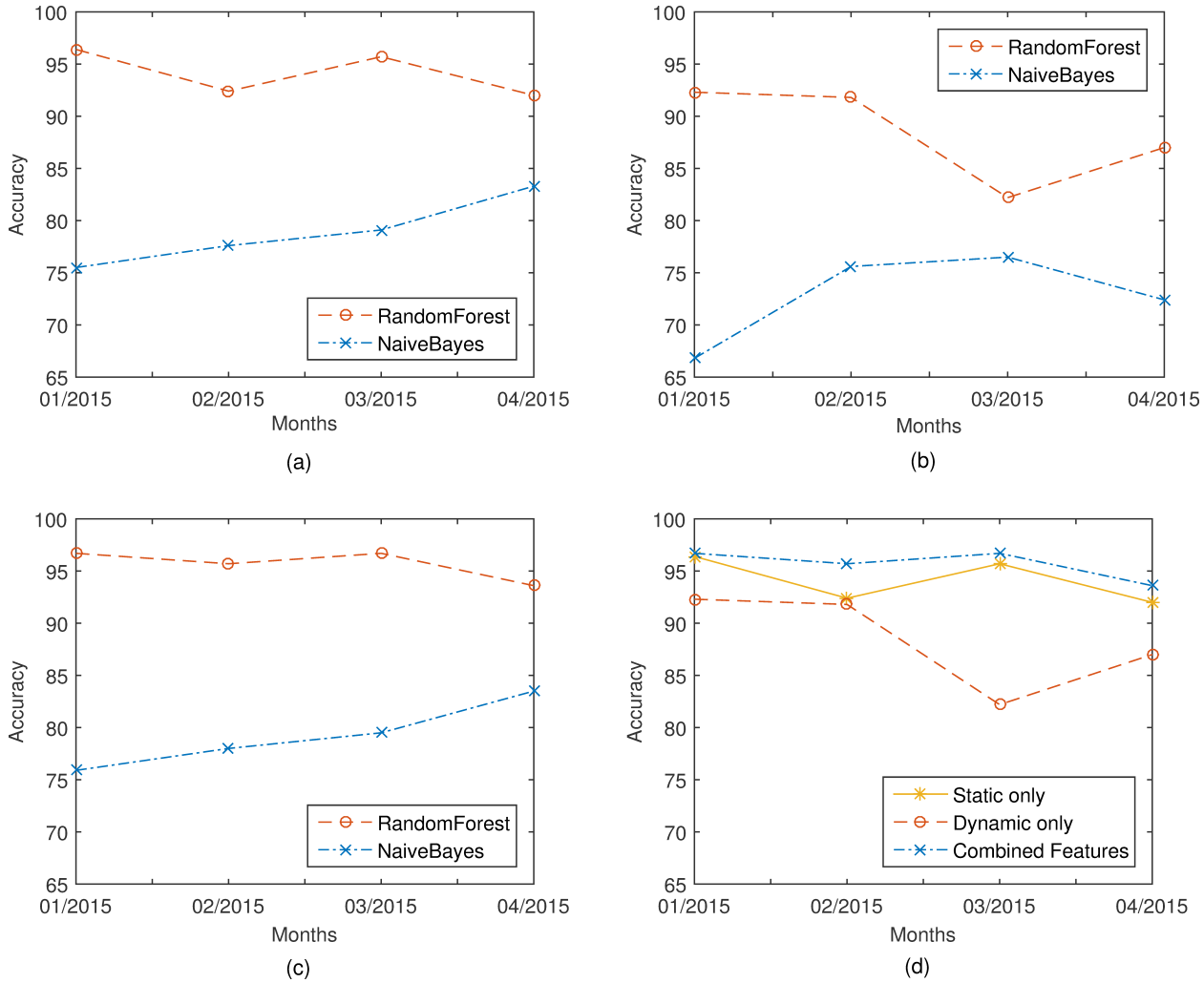

Figure 3.3: (a) Accuracy using static features only. (b) Accuracy using dynamic features only. (c) Accuracy using combined features. (d) Accuracy of Random Forest algorithm with different features sets.

An additional experiment was conducted to check the result of testing each month's training data on the month immediately following utilizing all features. Table 3.7 illustrates the results of the experiment. The experiment showed the different results when using training sets from different months. Again, this experiment shows that Random Forest outperforms Naive Bayes.

3.7 Discussion

As demonstrated in the previous section, the Random Forest algorithm outperformed Naive Bayes in every experiment. It is expected that the accuracy will gradually decrease when predicting future

Table 3.7: Applying each month to the one after (Combined Features).

Classifier	Period	TP	FP	Acc	AUC
Random Forest	Dec14-Jan15	95.5%	2.1%	96.7%	99.5%
	Jan15-Feb15	98%	1.3%	98.3%	99.8%
	Feb15-Mar15	96.3%	1.1%	97.6%	99.7%
	Mar15-Apr15	96.4%	0.9%	97.8%	99.7%
Naive Bayes	Dec14-Jan15	92.1%	40.3%	75.9%	78.5%
	Jan15-Feb15	90.3%	29.1%	80.6%	85.3%
	Feb15-Mar15	96%	42.5%	76.8%	78.2%
	Mar15-Apr15	90.1%	24%	83.1%	83.9%

months, since new malware applications could emerge that differ from those in the initial training set. However, as can be noticed in Table 3.5, some months show a gradual drop in performance, while others show a gain. For example, Random Forest performed well using only dynamic features until 3/2015, followed by a sudden drop of performance as shown in Fig. 3.3(b). There are two primary reasons. The first is due to the fact that new campaigns emerged in March 2015 with a significant population represented in the file data. The second reason is the surge data from malware families that were underrepresented in the training data. For example, the malware family `Win32/Loring` accounts for 2,147 files in December 2014, but the number of files increased in March 2015 to 11,763 files, representing the top family at almost 23.5% of the March 2015 test files. Conversely, new malware families emerged in March, such as `Win32/Rofin`, with 228 files and `Win32/Delf` with 118 files. The total number of new families in March is 265 with 1033 different files. Table 3.8 shows the top ten malware families and the ratio of each family to the total number of files in each month. Family names as shown in the table are obtained from Microsoft anti-virus labels. The first column in the table represents the family distribution in training data (100,000 malicious files), whereas the other four columns represent distribution in the test data (50,000 files each).

The training data from December 2014 contains 1,259 different malware families, with 3,049

different variants. Some families have only one variant present in the training data, such as `Win32/Claretore`, while other families contain over a hundred variants, such as `Win32/Vobfus`. The top family in December 2014, shown in table 3.8 and in regards to the total file number of variants is `Win32/Virut`, where the family has 16,666 files. Its two most common variants are `Win32/Virut.BR` and `Win32/Virut.BN` with 6,728 and 4,841 files, respectively.

Table 3.8: Top Ten Malware Families in Each Month

Training December 2014		Test January 2015		Test February 2015	
Families	Count	Families	Count	Families	Count
Virut	16.66%	Eggnog	34.22%	Upatre	14.85%
Vflooder	4.60%	Virut	7.50%	Nabucur	8.34%
Elkern	4.55%	Upatre	5.01%	Soltern	7.69%
Parite	4.39%	Simbot	4.11%	Virut	7.47%
Jadtre	4.13%	Loring	3.89%	Vobfus	4.31%
Sality	3.26%	Parite	3.74%	Expiro	4.15%
Ramnit	2.44%	Viking	2.96%	Loring	3.96%
Almanahe	2.40%	Beaugrit	2.70%	Sality	3.48%
Loring	2.15%	Vflooder	2.67%	Comame!gmb	3.39%
Gupboot	1.99%	Sality	2.39%	Berbew	3.33%
		Test March 2015		Test April 2015	
		Families	Count	Families	Count
		Loring	23.53%	Virut	17.17%
		Expiro	14.69%	Lydra	9.57%
		Vobfus	7.57%	Soltern	8.94%
		Virut	5.95%	Vflooder	7.48%
		Delf	5.59%	Loring	4.23%
		Mydoom	5.14%	Rofin	4.17%
		Worm:Win32/VB	5.12%	Trojan:Win32/VB	3.64%
		Vflooder	4.53%	Sality	3.53%
		Comame!gmb	3.57%	Worm:Win32/VB	3.30%
		Beaugrit	2.32%	Morefi	3.16%

3.7.1 Features Evolution

With a feature vector of length 32,000+ and over a million files, it is computationally difficult to apply feature reduction algorithms to ascertain the importance of individual features. In order to get a sense of features importance, and for the purpose of illustration, we examined the occurrence of features in the benign and malicious files. A feature is rated as important when it can distinguish between each class. We consider the disparity of occurrence, rather than the mere number of occurrence, to determine if a feature is more distinguishing than other features.

Three classes of features were selected to illustrate how the feature evolution progresses month to month. Tables 3.10, 3.11, and 3.9 show section names, evolution of APIs, and injected processes, respectively, observed in the data of each month. Each month contains 200,000 files divided evenly between clean and malicious categories, resulting in one million files after five months.

Each column in Tables 3.10, 3.11, and 3.9 shows the month of observation, feature name, and number of files observed having that feature. The features listed are the top five distinguishing features in that class. For example, in Table 3.11 the API `Sleep` observed in 65,573 and 30,414 clean and malicious files, respectively, providing a difference of 35,159 additional files in the *Clean* category. This was the highest difference among APIs, and thus the top distinguishing API in the API features. Likewise, the *Malware* row in the table lists the most distinguishing features, with the highest contrast favoring malicious files.

An interesting observation in Table 3.11 is the high number of `_controlfp`, `__p__commode` and other C-runtime floating point functions in four months, but not in January 2015. This can be understood when we look at January's malware families in Table 3.8. In January 2015, there was a surge in the number files of the `Eggnog` family, utilizing registry APIs extensively, and accounting for 34.22% of files in that month. This made its API dominate the top five malware APIs that month.

In `Section Names` category, it is evident from Table 3.10 that UPX packer was heavily used by malicious files since most distinguishing section names are UPX0, UPX1, etc. UPX is a free

and an easy to use packer that is frequently abused by adversaries to obfuscate executables and evade string signature. Behind UPX is "ASPack" packer, which is a commercial packer and also used for similar obfuscation.

Table 3.9: Top five injected processes with highest differences of usage between malicious and clean files.

	Clean		Malware	
	Value	Files	Value	Files
Dec 2014	SELF	2,750	explorer.exe	1,228
	dwwin.exe	2,407	python.exe	763
	msiexec.exe	717	services.exe	658
	wmiprvse.exe	325	cmd.exe	627
	MiniThunderPlatform.exe	81	poskAAUk.exe	547
Jan 2015	msiexec.exe	369	explorer.exe	1,616
	SELF	2,207	iexplorer.exe	805
	drwtsn32.exe	70	cmd.exe	746
	DTLService.exe	14	net.exe	703
	TenioDL.exe	12	pAAMUcMg.exe	678
Feb 2015	msiexec.exe	351	SELF	6,904
	tmp1.exe	24	iexplorer.exe	1,829
	DTLService.exe	22	explorer.exe	1,746
	LMIGuardianSvc.exe	19	python.exe	1,233
	LMI_Rescue_srv.exe	10	biudfw.exe	1,026
Mar 2015	SELF	2,027	explorer.exe	1,679
	msiexec.exe	935	services.exe	1,490
	dwwin.exe	2,061	cmd.exe	853
	HssInstaller.exe	96	net.exe	711
	af_proxy_cmd_rep.exe	66	com7.exe	611
Apr 2015	SELF	2,281	explorer.exe	1,322
	msiexec.exe	893	services.exe	898
	GameCenter@Mail.Ru.exe[1]	81	dwwin.exe	2,593
	drwtsn32.exe	97	python.exe	629
	wmic.exe	55	iexplorer.exe	527

[1] "GameCenter@Mail.Ru.exe". Obviously the file has an interesting name that could give the impression that it is malicious. However, we analyzed the file and it was found to be clean and legitimately signed. For more information, see https://goo.gl/FCEPLh.

Table 3.10: Top five section names with highest differences of usage between malicious and clean files.

	Clean		Malware			Clean		Malware	
	Value	Files	Value	Files		Value	Files	Value	Files
Dec 2014	.rdata	76,451	UPX0	26,181	Jan 2015	.data	81,421	DATA	25,239
	.reloc	44,030	UPX1	26,140		.rdata	76,326	BSS	24,040
	.idata	28,673	UPX2	5,011		.text	80,127	CODE	24,165
	.tls	21,951	.aspack	3,141		.ndata	18,308	.tls	26,025
	.data	72,863	.adata	3,274		.rsrc	94,772	.aspack	4,779
Feb 2015	.rdata	86,894	UPX0	12,209	Mar 2015	.rdata	80,807	UPX0	27,235
	.reloc	54,451	UPX1	12,087		.data	81,372	UPX1	27,209
	.idata	38,278	rsrc	2,785		.text	80,992	.aspack	14,808
	.tls	26,122	uinC	2,777		.ndata	22,246	.adata	15,017
	.ndata	16,994	.vmp0	2,685		.reloc	39,069	ExeS	13,326
Apr 2015	.rdata	85,326	UPX0	49,548					
	.data	82,194	UPX1	49,460					
	.text	82,612	UPX2	15,673					
	.ndata	31,978	.adata	3,585					
	.rsrc	98,094	.aspack	3,258					

Code injection is commonly used by malware to hide malicious code inside trusted processes. However, process code injection is still used by legitimate software. Table 3.9 illustrates the injected process names used by both benign and malicious files. It can be observed that "SELF" is the most injected process with respect to the number of clean files. SELF describes the same running process. This is done usually when a plugin or a piece of code is received and loaded by the running process, then injected into the current context as a thread. On the other hand, it is obvious from the table that "explorer.exe" is the mostly injected process by malware. This behavior was illustrated by `Win32/Sality` and `Win32/Madang`.

3.8 Conclusion

Extensive analysis is presented and applied to a recent and diverse set of one million files, both malicious and clean, where thousands of features were extracted. We defined and integrated fea-

tures from three contexts to describe a malicious file and used Random Forest and Naive Bayes machine learning algorithms to create and train classifiers to detect malicious programs. In one experiment, a classifier was trained on a single month's data and used to detect malware from future months, up to four months ahead with a high detection accuracy up to 96.7% and false positive rate of 2.1%. The evolution of features and malware families over the course of five months were also studied and illustrated. As future work, we plan to apply feature selection algorithms to get the most important features. In addition, we plan to leverage the system capabilities to cluster the samples into families as well as consider other file types.

Table 3.11: Top five APIs with highest differences of usage between malicious and clean files.

	Clean		Malware	
	Value	Files	Value	Files
Dec 2014	Sleep	65,573	_controlfp	19,860
	DestroyWindow	50,603	__p__commode	21,630
	WriteFile	61,018	__p__fmode	22,112
	ReadFile	54,884	_except_handler	17,022
	SetFilePointer	53,836	_c_exit	12,265
Jan 2015	Sleep	65,729	RegFlushKey	19,511
	GetCurrentProcess	66,667	RegSetValueExA	28,365
	MultiByteToWideChar	59,977	RegCreateKeyExA	26,161
	GetLastError	66,898	RegQueryValueExA	33,492
	FreeLibrary	53,781	RegOpenKeyExA	33,096
Feb 2015	MultiByteToWideChar	69,448	_controlfp	21,698
	SetFilePointer	63,675	__p__commode	23,281
	GetSystemMetrics	50,013	__p__fmode	23,809
	SetWindowPos	47,909	_except_handler3	18,823
	WriteFile	68,699	exit	28,770
Mar 2015	DestroyWindow	60,392	_controlfp	20,745
	SetFilePointer	64,695	__p__commode	24,323
	GetExitCodeProcess	51,651	__p__fmode	23,392
	ReadFile	65,025	_c_exit	16,114
	Sleep	74,499	_cexit	23,383
Mar 2015	DestroyWindow	65,527	VirtualProtect	34,450
	Sleep	78,919	_controlfp	15,408
	GetExitCodeProcess	57,957	__p__commode	16,601
	GetCurrentProcess	75,535	__p__fmode	16,906
	MultiByteToWideChar	66,688	_c_exit	10,331

Chapter 4: A CONTROL FLOW GRAPH-BASED SIGNATURE FOR PACKER IDENTIFICATION

4.1 Introduction

Zero-day malware detection is a persistent problem. Hundreds of thousands of new malicious programs are produced and published on the Internet daily. Although conventional signature-based techniques are still widely relied upon, they are only useful for known malware. Many research efforts have aimed at helping flag and detect unknown suspicious and malicious files. All of these techniques can be categorized into three types: sandbox analysis, heuristic static analysis or code emulation. Among the three, heuristic static analysis is the fastest, yet the weakest against obfuscation techniques.

Code obfuscation includes packing, protecting, encrypting or inserting anti-disassembly tricks, and is used to hinder the process of reverse engineering and code analysis. About 80% to 90% of malware use some kind of packing techniques [30] and around 50% of new malware are simply packed versions of older known malware [58]. While it is very common for malware to use code obfuscation, benign executable files rarely employ such techniques. Thus, it has become a common practice to flag an obfuscated file as suspicious and then examine it with more costly analysis to determine if it is malicious or not.

In this chapter, we present a new method for packer or obfuscator detection and identification that builds upon our instructions-based technique presented in Chapter 2. The prior work builds a recursive traversal disassembler that extracts the control flow graph of binary files, and then computes various statistical features of this graph to distinguish between obfuscated and normal files. Once the file is confirmed to be obfuscated, the new work presented in this chapter constructs a compact signature of the control flow graph at the entry point which is resilient to dummy instructions insertion and a number of graph manipulation methods. This signature can be compared to a set of signatures in the database to determine the packer. If the packer is new or unknown, the sig-

nature can be automatically constructed from a set of packed files and used to update the database with no human intervention.

More specifically, the contributions of the chapter are:

- We introduce a compact signature for the control flow graph that can be used to identify the obfuscator or packer used in a file.
- The process of CFG signature construction is done automatically with no human intervention, so the system can store and be updated with new signatures to detect different types of obfuscators or packers.
- The signature is resilient to some instructions modifications and shuffling, which makes a single signature efficient against mildly different versions of the same code.
- We achieve a fast scanning speed of 0.5 ms per file on average, given the fact that our method encompasses disassembly, control flow graph extraction, signature creation and matching.
- There is a strong potential that the same technique can be used to detect malware variants.

4.2 Related Work

Most of the current work of detecting obfuscated files is based on executable file structure characteristics, such as file entropy, signature, or file header analysis.

4.2.1 Entropy-based Detection

Lyda and Hamrock presented the idea of using entropy to find encrypted and packed files [30]. The method became widely used as it is efficient and easy to implement. However, some non-packed files can have high entropy values and thus lead to false-positives. For example, the `ahui.exe` and `dfrgntfs.exe` files from Windows XP 32-bit have an entropy of 6.51 and 6.59, respectively for their `.text` section [68, 69] (our system detects these files correctly as non-packed). While entropy-based methods can be effective against encryption or packing obfuscation, it is ineffective

against anti-disassembly tricks, such as the one mentioned in Section 2.3, and even against simple byte-level XOR encryption. Finally, the entropy score of a file can also be deliberately reduced to achieve an entropy value similar to that of a normal program [64].

4.2.2 Signature-based Detection

A popular signature-based tool to find packed files is *PEiD*, which uses around 620 packer and crypter signatures [5]. The obvious drawback of this tool is that it can identify only known packers, and sophisticated malware usually uses custom packing or crypting routines. Moreover, even if a known packer is used, the malware writer can change a single byte of the packer signature to avoid being detected. Finally, it is a time consuming job to add a signature of a new packer since it usually requires manual analysis to extract a reliable signature.

4.2.3 File Header-based Detection

File header-based techniques include those proposed by [38, 50, 53, 63]. These techniques can get good results only when the packer changes the PE header in a noticeable way. Indeed, many public packers exhibit identifiable changes in the packed PE file. However, this is not always the case with custom packers and self-encrypting malware. Moreover, if the packing avoids the PE file header completely, and instead focuses on instruction sequence or program execution flow, there will be absolutely no trace in the header. Lastly, even if a packer introduces some identifiable artifacts in the header, some of these can be easily removed by the malware author without affecting the integrity of the executable. For example, section names and some strings could be manually restored to match the original file header.

Besides the specific shortcomings of each technique described here, none of them can statically detect the presence of anti-disassembly tricks or other forms of control flow obfuscation. Unfortunately, these tricks are now commonly used in a wide range of advanced malware. This major shortcoming is the first motivation for our work. Because our proposed system does not depend on a coarse-grained entropy score of the file or section, byte signature of packers, or file header

features, it overcomes the techniques' individual shortcomings while also addressing the need for detecting files containing anti-disassembly tricks and control flow obfuscation.

4.3 Background

4.3.1 Packer Identification

If a program is determined to be obfuscated, it is helpful to determine the obfuscator or packer that was used. Identifying the packer will accelerate the automatic unpacking process, since the file will be fed to the designated unpacker. If the unpacker was not determined, usually the file is fed to a heuristic/universal unpacker that will emulate the instructions and guess the original entry point of the unpacked file. Universal unpackers are not as reliable as those unpackers that were made specifically for known packers and obfuscators. A universal unpacker has to make some guesses and assumptions to determine the end of unpacking process that might deem wrong. Because of the shortcomings with packer detection, reliable packer identification is also problematic. This fact represents the second motivation for our work.

4.3.2 Control Flow Graphs

For a control flow graph (CFG) G consisting of a set of vertices $\mathbb{V}$ and edges $\mathbb{E}$, the following function maps a node to its 2-tuple of right and left children:

$$Children(x) = (x_r, x_l) \,|\, x, x_r, x_l \in G$$

In addition, parents of a node x are those nodes that have a direct connection to x. The set of parents is defined as:

$$Parents(x) = \{y \in G\} \,|\, x \in Children(y)$$

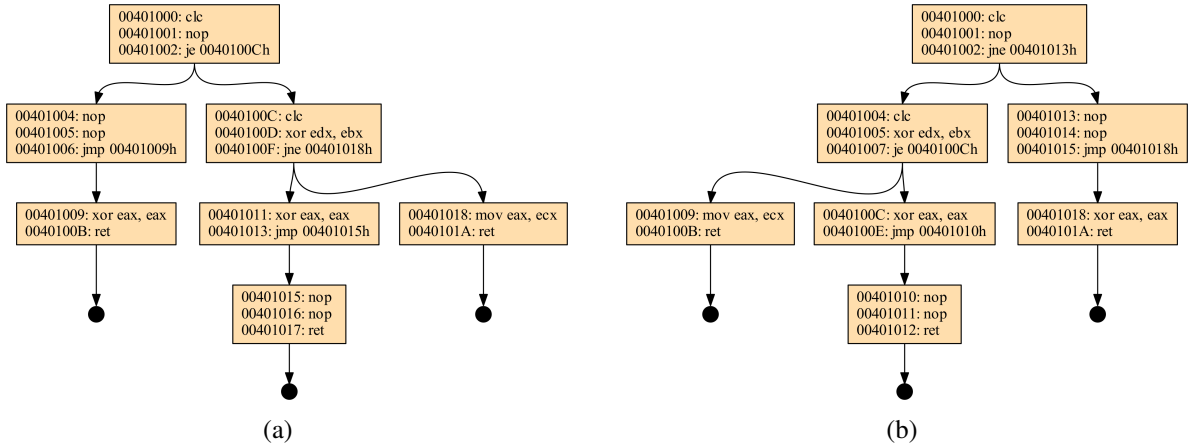

Figure 4.1: (a) An example CFG, (b) a mirrored version of the same CFG.

Each node in a CFG has up to two children[1]. A node x is an exit node $\iff Children(x) = \varnothing$, and x is the entry node $\iff Parents(x) = \varnothing$.

CFG Manipulation

CFGs can be manipulated in a number of ways without affecting the overall logic of the program. These manipulations can be due to normal compiler optimization processes or, as in the case of malware, polymorphism. The shape of the CFG changes while the original logic does not. For example, a control flow instruction such as `JE` can be changed to `JNE`. In this case, the left and right children will be swapped which results in a mirrored version of the original CFG. Figure 4.1 shows a CFG and its mirrored version, whereby each control flow instruction with two children in Figure 4.1(a) has been negated so as to transform it into the mirrored version shown in Figure 4.1(b).

Another way that CFGs can be manipulated without affecting program logic is by prepending a node at the entry node. For example, putting a `JMP` instruction that points to the first instruction of the code will create an extra basic block at the beginning of the CFG while keeping the logic unchanged. Similarly, nodes can be appended to exit nodes or inserted in the middle of the graph. Finally, a node may be split into two to increase the number of nodes.

[1]Considering programs written in Intel x86 or x86-64 instructions set

The rest of the chapter is structured as follows: Section 4.4 explains our methodology of constructing a reliable signature from the control flow graph. Section 4.5 describes our experiments and results. Section 4.6 discusses the results and potential limitations. Section 4.7 concludes the chapter.

4.4 Control Flow Graph-based Signature

4.4.1 Graph Preprocessing

As discussed earlier, polymorphic codes usually insert dummy instructions to evade detection by string signature. These instructions could be either instructions that do not affect the execution flow of the program, such as arithmetic or memory operations, or control flow instructions. In the first case, the control flow graph of the program will not be affected, and the CFG signature will thus stay the same. In the second, inserting control flow instructions will add more basic blocks to the CFG and change the CFG signature. For instance, a malicious program could add a series of `JMP` instructions between basic blocks, with each `JMP` pointing to the next. The malicious program could also divide each basic block into smaller pieces and connecting them with `JMP` instructions in between.

In all these cases, the logic of the program will be intact, but the presence of dummy `JMP`s will change the CFG. Thus, we preprocess the CFG in order to alleviate the effects. Specifically, we preprocess the graph before creating the signature such that nodes with only one child are combined into one, as follows:

$$Merge(x, y) = v \iff x_r = x_l = y \,|\, x, y, v \in G \; and \; x_r, x_l \in Children(x)$$

To mitigate the effect of mirroring, we traverse the graph and for each node we determine the shortest distance to root, and assign this value to each node. Then every input CFG is topologically sorted so that for each node, the deepest node on its right branch has a shorter distance to root than the deepest node on the left branch. The sorting operation is defined as:

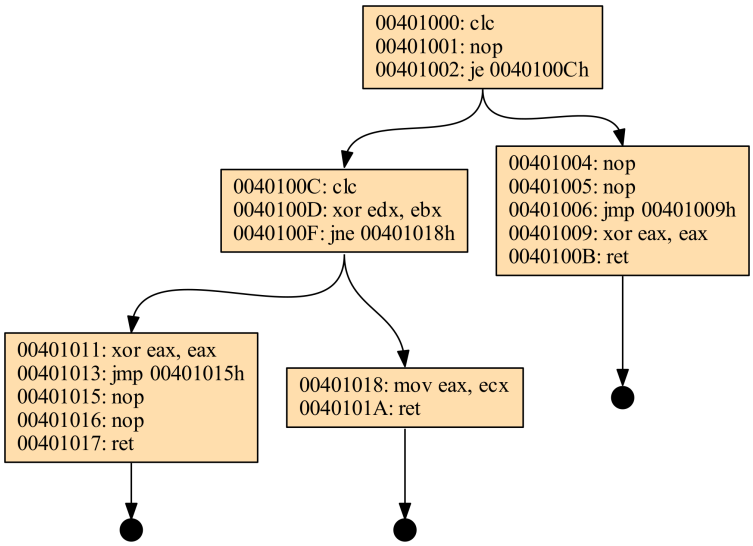

Figure 4.2: A normalized and sorted version of the CFG in Figure 4.1(a).

$$\forall x, x_r, x_l \in G, swap(x_r, x_l) \iff x_{rd} > x_{ld}$$

$$where\ x_{rd}, x_{ld}\ are\ the\ depth\ of\ right\ and\ left\ branch\ of\ x,\ respectively.$$

Figure 4.2 shows the normalized and sorted version of the program in Figure 4.1(a).

4.4.2 Exact Signature

In order to compare two CFGs, we use two types of signatures, exact and approximate. The exact signature identifies two CFGs that look identical. However, if one of the CFGs was subjected to minor manipulation, the signatures will fail to match. In this case, an approximate signature is used in an attempt to withstand minor manipulation of the CFG, although this may come at the cost of possible false-positives.

There are a number of known representations of graphs. In this chapter, we represent the control flow graph as a series of integer values. Nodes are visited in breadth-first fashion, and each node is given a sequential ID. Since each node in the control flow graph can have a maximum of two children, an ID is followed by two IDs for the children, then the child with the least ID is listed

followed by its children, etc. Each node is represented as 3-elements tuple of *(Node ID, Child1 ID, Child2 ID)*. If a node has only one child, the second nonexistent child is represented by zero. In case of an exit node that has no children, both children are represented by two zeros. For example, the following signature represents the graph in Figure 4.2.

```
1 2 3 2 4 5 3 0 0 4 0 0 5 0 0
```

The number of nodes in the graph equals the length of the signature divided by 3. With this representation, the original graph can be fully restored from the signature.

Obviously, the node ID is repeated when it is listed as a parent and as a child. Thus, the signature can be shortened by removing *Node ID*, then each two consecutive IDs represent the two children of a node. The previous signature can be rewritten as:

```
2 3 4 5 0 0 0 0 0 0
```

In this version, the number of nodes equals the length of the signature divided by 2 and still the original graph shape can be fully restored with this signature.

4.4.3 Approximate Signature

As mentioned earlier, the CFG can be modified in various ways without affecting the original program logic. Thus, exclusively relying on finding exact signature matches will miss many possibly isomorphic graphs and increase the rate of false negatives. In order to overcome this problem and match similar CFGs, we construct an approximate representation of the graph, as follows.

Each node is labeled with two numbers, x, y. These values represent the number of parents and the number of children of each node, respectively. Then, visiting each node in the graph in a breadth-first first fashion, the numbers are listed in the order of the visited nodes. Since each node cannot have more than two children, the possible values for x are *0*, *1* and *2* and thus x can be represented with just 2 bits. On the other hand, the possible number of parents is not limited to a specific range of values. However, we scanned 300,000 files and found that the average number of

parents for all CFG nodes is 1.3. We also found that only 130 files in this samples set have a CFG containing one or more nodes with more than 63 parents.

Since the number of children can be represented in 2 bits, there are 6 remaining bits available for the number of parents if the entire label is to be contained in one byte. Thus, a one-byte label can represent a maximum of 63 parents, which is sufficient for the vast majority of CFGs, based on our sampling.

To illustrate this, we refer again to the graph in Figure 4.2 and present the following three hexadecimal sequences representing the number of children, parents, and the combined value, respectively. For each byte in the combined sequence, the two low order bits represent the number of children, and the high order six bits represent the number of parents.

```
02 02 00 00 00
00 01 01 01 01
02 06 04 04 04
```

There are many properties that each node in the CFG can have, such as, number of parents, number of children, shortest distance to exit node, shortest distance from root, number of instructions and level. However, with the exception of the number of children and parents, examining these properties shows that they are susceptible to change even if a single node is inserted or removed. For example, if we define a signature based on the distance of each node to the root, and then a new node is inserted as a new root, the entire signature will be completely different. The same can happen if we used a shortest distance to exit node measurement. Although the number of instructions in each node will not be affected by CFG manipulation, it will change if dummy instructions are inserted or removed, something very common in polymorphic code, and usually easier to accomplish than changing the CFG.

Overall for a given CFG, the number of direct children and parents are resilient to these modifications. Only the node that is a direct children or parent of a modified node will have its value in the signature changed. The rest of the CFG is not affected in terms of parents or children and thus the rest of the signature will be the same.

This makes the choice of these two properties a good fit to detect mildly polymorphic or optimized code when the signature of a similar variant is known.

4.4.4 Signature Matching

To compare two signatures, we use a difference score based on edit distance as a metric for comparison. The difference score between two signatures x and y is defined as the number of insertion, deletions and substitutions needed to convert one string to another. We define a value δ as the number of allowed distance between two signatures to establish a match. Each signature x in the database has its own value of δ to make sure we match variants of x correctly. Hence, a match between two signatures x and y is defined as:

$$\textit{x and y are matched} \iff 0 <= EditDistance(x, y) <= \delta_x$$

Where *x* is the signature stored in the database and *y* is the signature of the input file.

4.5 Experiment and Evaluation

To evaluate the utility of our signature and matching algorithm, we conducted an experiment using 7 publicly-available packers commonly abused by malware authors to obfuscate their code: *UPX*, *Execryptor*, *Themida*, *FSGv1.33*, *FSGv2.0*, *eXpressor* and *Yoda's Protector*. For each packer, one obfuscated file was sufficient to get a CFG signature able to detect all files in each test set of 20 obfuscated files for each packer.

For each file in the data set, the control flow graph is extracted from the function at the entry point. The extraction is stopped at 200 basic blocks limit. In the case of Yoda's protector, the test set needed 3 signatures in PEiD to detect all of them, as the test set was packed by 3 different versions of the packer. However, we only needed one signature of the CFG to detect files packed by the 3 versions. Each signature successfully detected 100% of the test set. Table 4.1 shows the δ of each signature, where each δ in the table is obtained by scanning the test set and obtaining the

Table 4.1: Value of δ for each signature.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
δ	10	1	1	1	1	1	2

Table 4.2: The distance of each packer signature from the other 6 packers.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
UPX	0	37	156	38	30	29	31
Execryptor	37	0	184	12	31	24	17
eXpressor	156	184	0	182	159	167	175
Themida	38	12	182	0	35	25	18
FSG v1.33	30	31	159	35	0	12	31
FSG v2.0	29	24	167	25	12	0	24
Yoda	31	17	175	18	31	24	0

maximum difference score.

Table 4.2 shows the distance of each signature from the other 6 packers. Note that the distance between a signature of packer x and the other 6 packers, is higher than δ_x, which indicates that files belong to a packer x will not be misidentified as packer y, for any two different packers x and y.

To measure the false-positive rate for each signature, we scanned 324 non-packed files taken from a clean Windows 7 machine. We had zero false-positive rate with all the signatures. In fact, the difference score was much greater than the δ value of each signature. Table 4.3 shows the lowest difference score for each signature when scanning the non-packed files.

Table 4.3: Minimum score for each packer against non-packed files.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
δ	29	12	103	12	24	17	17

Table 4.4: Edit distance scores of UPX file set.

Score	0	3	6	10
Number of files	7	11	1	1

4.6 Discussion and limitations

During the experiment, we noted that UPX has a number of slightly different CFGs for some files even if the same version is used. We found that during packing, UPX will add code blocks in the unpacking stub based on the input file's structure. For example, if the file has a relocation section, UPX unpacking stub will contain some code that deal with unpacking the relocation section, and this code will not be present in the stub of other files that do not have a relocation section. This is in fact the reason UPX have higher value of δ than the other packers. That is, the more difference among the CFGs belong to the same packer, the higher the value of δ needed to match all of them. Table 4.4 shows the different edit distance scores we obtained when scanning UPX file set.

Typically, a PEiD signature and a CFG signature for UPX are as follows:

PEiD
```
60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? C7 87
?? ?? ?? ?? ?? ?? ?? ?? 57 83 CD FF EB 0E
?? ?? ?? ?? 8A 06 46 88 07 47 01 DB 75 07
8B
```

CFG
```
01 09 0A 05 05 0E 0E 0A 05 06 0A 06 0A 06
0A 05 05 05 0A 05 0D 0A 05 0A 0A 0A 0E 06
05 05 0A 05 06 06 05 09 06 0A 09 05 05 05
05 0A 0A 05 04 06 05 05
```

The symbol "`??`" in PEiD signature is a wildcard symbol refers to matching any byte value.

Yoda's protector file set is obfuscated using three versions of the packer which are v1.02b, v1.03.2 and v1.03.3. PEiD is using three signatures to detect all the files in the set, one signature for each version. Nevertheless, only one CFG signature with δ value of 2 was needed to detect

the three versions. On the other hand, we chose to separate the signatures for FSG packer, so we have two different signatures to detect the FSG v1.33 and FSG v2.0, this is why we have δ of value 1 for each. However, if one signature is needed to match both versions, the δ is determined by experiment to be 6. Therefore, it is up to the system administrator to choose whether to have lower value of δ and more signatures, or higher value of δ with less number of signatures, considering the risk of false-positive with high δ.

We set a minimum and maximum size of nodes of 30 and 200 nodes, respectively. So an input CFG with a number of nodes less than the minimum is skipped, as considering CFG with size less than this minimum could produce high number of signature mismatches. On the other hand, the maximum number of 200 is thought to be enough to detect different CFGs, and so if the input file has more nodes than the maximum, it is trimmed down to 200 nodes.

Regarding the speed of the system, the process of scanning a file involves disassembly, control flow graph extraction, signature generation and matching. This process takes on average 0.5 ms per file on a machine with 8GB and Xeon processor with four cores.

It worth to mention that one limitation of the system is imperfect disassembly due to indirect addressing. However, this problem exists in all disassemblers since the problem of perfect disassembly is undecidable problem [22]. Nevertheless, this should not be a problem in our case since the same rules of disassembly are applied on both the training and test files.

4.7 Conclusion

Hundreds of thousands of malware are produced every year, the vast majority of them are obfuscated and packed. Unpackers are used to reveal the malicious code of the file by unpacking its contents. It is important to identify the packer used in the malware in the triage process, so the appropriate packer can operate on the file. Current techniques of packer identification mostly rely on byte signature, which can be easily evaded by the attacker. In this chapter, we introduce a new technique for packer/obfuscator identification based on the control flow graph of the file. We introduced two types of signatures to compare and match two CFGs, an exact signature and ap-

proximate one. The CFG is normalized and sorted and then the approximate signature is generated in a short runtime of 0.5 ms per file on average. The approximate signature with the preprocessing of the graph is able to withstand minor code modification usually done by attackers. The method showed very good performance when tested against seven common packers with no false-positive.

We believe that the technique has a strong potential to be efficient in detecting malware variants and polymorphic code, which will be implemented in our future work.

Chapter 5: CONCLUSION AND FUTURE WORK

5.1 Conclusion

This dissertation addresses the problem of proactive automatic malware detection. Hundereds of thousands of malware samples are produced every year, and most of them are obfuscated. This research tackles the problem of obfuscated malware detection and classification by introducing three techniques. The dissertation makes the following three contributions:

- *A novel approach for detection of sophisticated obfuscation and packing*: Obfuscation is a common way to hide the intent of malicious files. Several ways exist to scramble the code and make it difficult to analyze. We presented a technique for sophisticated obfuscation detection in Chapter 2. The key idea is to extract and utilize information from the Control Flow Graphs (CFGs) of malware programs to determine whether the file employs code obfuscation techniques. The technique overcome limitations of other techniques used for the same purpose while maintaining a fast execution runtime. Experimental results show that the new technique can detect a variety of obfuscation methods (e.g., packing, encryption, and instruction overlapping) in a short runtime, which makes the technique feasible and practical for deployment on critical systems. This patent-pending technique paves the way for developing the two other techniques presented in the dissertation.

- *Multi-context features for automatic malware detection*: Given a suspicious file detected by the previous method, our second technique presented in Chapter 3 aims at automatically classifying whether the file is malicious or not, by utilizing several types of information about the suspicious file (e.g., file structure, runtime behavior, and instructions). The technique is efficient against both plain and obfuscated malware samples since it does not depend solely on static features. Random Forest and Naive Bayes classifiers were used to build the machine learning models. A key contribution of this technique is the definition and utilization of over 32,000 features of files, including file structure, runtime behavior, and instructions. To the

best of our knowledge, this is the first effort that defines and uses such a comprehensive feature set.

- *A novel technique for packer and obfuscator identification*: The third technique leverages the first one to automatically identify and construct a signature for packers and malware obfuscators based on the control flow graph on the file. The method presented in Chapter 4 constructs a signature based on the control flow graph of the file, which makes the new signature has many advantages over the conventional byte signature. While it is sometimes enough for an attacker to change only one or more bytes to evade detection with byte signature, it is more difficult to change the control flow graph of the file. In addition, the signature is resilient to some instructions modifications and shuffling, which makes a single signature efficient against mildly polymorphic versions of the same code. Last but not least, the signature extraction of new obfuscators is done automatically with no human analyst intervention.

5.2 Future Work

In this section, we present some potential research directions which might guide our future works.

5.2.1 Control flow graph-based signature for polymorphic variants detection

Packed malware files are dominant nowadays, and the problem of unpacking or deobfuscating the malware sample is insisting due to the high number of obfuscated samples. Obfuscation is very effective in evading detection by anti-virus software. A malware family can stay active in the wild for a long period as long as it has different mutating variants that bypass the conventional byte signature detection. We plan to leverage our work presented in Chapter 4 to detect mutating or polymorphic variants of malware with the control flow graph signature. To address the limitation of perfect disassembly discussed in Chapter 4, we plan to integrate an emulator to resolve indirect addressing. The emulation part is planned to be minimal so it does not have a major impact on the processing runtime. With this addition, we believe the signature will be more descriptive and accurate, and thus will help detecting multiple variants of polymorphic malware with minimal

number of signatures.

5.2.2 A fast malware detection using machine learning

Acquiring behavioral information of a malware sample is the slowest step in any malware detection system that uses this type of information. Thus, requiring the behavioral information to be present makes the system relatively slow. We plan to extend our work presented in chapter 3 by increasing the number of static features, plus adding some features based on partial emulation of the instructions. In addition, removing those features that requires running the sample ahead of the scanning time. We believe that this will lead to obtaining result similar or better than the one presented in chapter 3, with a significant boost to the runtime speed. This will make the system more suitable for deployment on user machines or systems that cannot wait for a malware sample to run in a controlled environment to monitor its behavior.

Bibliography

[1] Zheng Bu Abhishek Singh. Hot knives through butter: Evading file-based sandboxes. https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/fireeye-hot-knives-through-butter.pdf. Accessed: March. 27th, 2016.

[2] M. Ahmadi, G. Giacinto, D. Ulyanov, S. Semenov, and M. Trofimov. Novel feature extraction, selection and fusion for effective malware family classification. *ArXiv e-prints*, November 2015.

[3] Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, and Muddassar Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence*, AISec '09, pages 55–62, New York, NY, USA, 2009. ACM.

[4] Jordi Vazquez Aira. Emulate virtual machines to avoid malware infections. https://kasperskycontenthub.com/kasperskyacademy/files/2013/12/aira.pdf. Accessed: March. 27th, 2016.

[5] aldeid.com. PEiD. http://www.aldeid.com/wiki/PEiD. Accessed: Feb. 8th, 2014.

[6] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 3–14. ACM, 2012.

[7] AV-Comparative. File detection test of malicious software. March 2015.

[8] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[9] BeaEngine. BeaEngine. http://www.beaengine.org/. Accessed: Apr. 3rd, 2014.

[10] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? PowerPoint presentation at Black Hat USA, 2006. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

[11] CNET. Lenovo hit by lawsuit over superfish adware. http://www.cnet.com/news/lenovo-hit-by-lawsuit-over-superfish-adware/. Accessed: December. 9th, 2015.

[12] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. 1(9), December 2015.

[13] Mike Czumak. Pecloak.py - an experiment in av evasion. http://www.securitysift.com/pecloak-py-an-experiment-in-av-evasion/. Accessed: March. 27th, 2016.

[14] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *SIGARCH Comput. Archit. News*, 41(3):559–570, June 2013.

[15] Yuxin Ding, Wei Dai, Shengli Yan, and Yumei Zhang. Control flow-based opcode behavior analysis for malware detection. *Computers & Security*, 44:65 – 74, 2014.

[16] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley & Sons, 2008.

[17] Mojtaba Eskandari and Sattar Hashemi. Ecfgm: enriched control flow graph miner for unknown vicious infected code detection. *Journal in Computer Virology*, 8(3):99–108, 2012.

[18] Mojtaba Eskandari and Sattar Hashemi. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*, 23(3):154–162, 2012.

[19] Peter Ferrie. Principal anti-virus researcher at Microsoft. Personal Communication. Jan. 22nd, 2014.

[20] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[21] Karl Hiramoto. Technical account manager at VirusTotal. Personal Communication. Sept. 24th, 2014.

[22] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[23] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[24] BooJoong Kang, Kyoung Soo Han, Byeongho Kang, and Eul Gyu Im. Malware categorization using dynamic mnemonic frequency analysis with redundancy filtering. *Digital Investigation*, 11(4):323 – 335, 2014.

[25] Akshay Kapoor and Sunita Dhavale. Control flow graph based multiclass malware detection using bi-normal separation. *Defence Science Journal*, 66(2):138–145, 2016.

[26] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM.

[27] Sarat Kompalli. Using existing hardware services for malware detection. In *2014 IEEE Security and Privacy Workshops (SPW)*, pages 204–208. IEEE, 2014.

[28] Kaspersky Labs. The great bank robbery: the carbanak apt. `http://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/`. Accessed: Mar. 25th, 2015.

[29] McAfee Labs. Mcafee labs threats report for february 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf. Accessed: Mar. 25th, 2015.

[30] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *Security Privacy, IEEE*, 5(2):40–45, 2007.

[31] M0SA. Syp.01: Bypassing online dynamic analysis systems. Valhalla ezine, issue #4, November 2013. http://vxheaven.org/lib/vmo04.html.

[32] Emiliano MartÃnez. Software engineer at VirusTotal. Personal Communication. Dec. 25th, 2014.

[33] Qiguang Miao, Jiachen Liu, Ying Cao, and Jianfeng Song. Malware detection using bilayer behavior abstraction and improved one-class support vector machines. *International Journal of Information Security*, pages 1–19, 2015.

[34] Microsoft. Microsoft pe and coff specification. https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx. Accessed: Nov. 20th, 2015.

[35] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*, pages 204–215. Springer, 2008.

[36] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[37] pefile. https://github.com/erocarrera/pefile. Accessed: Jun. 6th, 2015.

[38] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett.*, 29(14):1941–1946, October 2008.

[39] Danny Quist, Val Smith, and Offensive Computing. Detecting the presence of virtual machines using the local data table. *Offensive Computing*, 2006.

[40] Ravinder R Ravula, Kathy J Liszka, and Chien-Chung Chan. Learning attack features from static and dynamic analysis of malware. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 109–125. Springer, 2013.

[41] Ethan Rudd, Andras Rozsa, Manuel Gunther, and Terrance Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *arXiv preprint arXiv:1603.06028*, 2016.

[42] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. `http://repo.hackerzvoice.net/depot_ouah/Red_%20Pill.html`. Accessed: March. 27th, 2016.

[43] M. Saleh, E.P. Ratazzi, and Shouhuai Xu. Instructions-based detection of sophisticated obfuscation and packing. In *Military Communications Conference (MILCOM), 2014 IEEE*, pages 1–6, Oct 2014.

[44] Moustafa E Saleh, A Baith Mohamed, and Ahmed Abdel Nabi. Eigenviruses for metamorphic virus recognition. *Information Security, IET*, 5(4):191–198, 2011.

[45] Zahra Salehi, Ashkan Sami, and Mahboobe Ghiasi. Using feature generation from api calls for malware detection. *Computer Fraud & Security*, 2014(9):9–18, 2014.

[46] Cuckoo Sandbox. Cuckoo Sandbox: Automated Malware Analysis. `http://www.cuckoosandbox.org/`. Accessed: June. 6th, 2015.

[47] Igor Santos, Jaime Devesa, Félix Brezo, Javier Nieves, and Pablo Garcia Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS 12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, pages 271–280. Springer, 2013.

[48] Igor Santos, Yoseba K Penya, Jaime Devesa, and Pablo Garcia Bringas. N-grams-based file signatures for malware detection. In *ICEIS (2)*, pages 317–320, 2009.

[49] Igor Santos, Borja Sanz, Carlos Laorden, Felix Brezo, and Pablo G Bringas. Opcode-sequence-based semi-supervised unknown malware detection. In *Computational Intelligence in Security for Information Systems*, pages 50–57. Springer, 2011.

[50] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, CEAS '11, pages 23–30, New York, NY, USA, 2011. ACM.

[51] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. *arXiv preprint arXiv:1508.03096*, 2015.

[52] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[53] M Shafiq, S Tabish, and Muddassar Farooq. PE-probe: leveraging packer detection and structural information to detect malicious portable executables. In *Proceedings of the Virus Bulletin Conference (VB)*, pages 29–33, 2009.

[54] M Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using Markov n-grams. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107. Springer, 2008.

[55] M.Zubair Shafiq, S.Momina Tabish, Fauzan Mirza, and Muddassar Farooq. PE-Miner: Mining structural information to detect malicious executables in real-time. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 121–141. Springer Berlin Heidelberg, 2009.

[56] Farrukh Shahzad and Muddassar Farooq. Elf-miner: Using structural knowledge and data mining methods to detect new (linux) malicious executables. *Knowl. Inf. Syst.*, 30(3):589–612, March 2012.

[57] Sanjam Singla, Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. A novel approach to malware detection using static classification. *International Journal of Computer Science and Information Security*, 13(3):1, 2015.

[58] Adrian Stepan. Improving proactive detection of packed malware. *Virus Bulletin*, pages 11–13, March 2006.

[59] C. Storlie, B. Anderson, S. Vander Wiel, D. Quist, C. Hash, and N. Brown. Stochastic identification of malware with dynamic traces. *ArXiv e-prints*, April 2014.

[60] P. Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, 2005.

[61] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. *CoRR*, abs/1403.1631, 2014.

[62] Ronghua Tian, M.R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 23–30, Oct 2010.

[63] S. Treadwell and Mian Zhou. A heuristic approach for detection of obfuscated malware. In *Intelligence and Security Informatics, 2009. ISI '09. IEEE International Conference on*, pages 291–299, June 2009.

[64] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P.G. Bringas. Countering entropy measure attacks on packed software detection. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 164–168, 2012.

[65] Xabier Ugarte Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *SSP 2015,*

IEEE Symposium on Security and Privacy, May 18-20, 2015, San Jose, CA, USA, San Jose, UNITED STATES, 05 2015.

[66] UPX. Upx: The ultimate packer for executables. `http://upx.sourceforge.net/`. Accessed: December. 7th, 2015.

[67] VirusTotal. `http://www.VirusTotal.com/`. Accessed: Jun. 6th, 2015.

[68] VirusTotal.com. `ahui.exe`. `http://goo.gl/kbbJKi`. Accessed: Feb. 7th, 2014.

[69] VirusTotal.com. `edfrgntfs.exe`. `http://goo.gl/XCqUcF`. Accessed: Feb. 7th, 2014.

[70] Weka. Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`. Accessed: Jun. 6th, 2015.

[71] Guanhua Yan, Nathan Brown, and Deguang Kong. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

[72] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.

[73] Kim Zetter. *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon*. Crown Publishing Group, New York, NY, USA, 2014.