A FRAMEWORK FOR AUTOMATED SIMILARITY ANALYSIS OF MALWARE

WEI LONG SONG

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS

SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Wei Long Song**
Entitled:	**A FRAMEWORK FOR AUTOMATED SIMILARITY ANALYSIS OF MALWARE**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Chun Wang	Chair
Amr Youssef	Examiner
Mohammad Mannan	Examiner
Wahab Hamou-Lhadj	Examiner
Amr Youssef	Supervisor

Approved Jamal Bentahar

Chair of Department or Graduate Program Director

________ 20 _____ __________________

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

ABSTRACT

A FRAMEWORK FOR AUTOMATED SIMILARITY ANALYSIS OF MALWARE

Wei Long Song

Malware, a category of software including viruses, worms, and other malicious programs, is developed by hackers to damage, disrupt, or perform other harmful actions on data, computer systems and networks. Malware analysis, as an indispensable part of the work of IT security specialists, aims to gain an in-depth understanding of malware code. Manual analysis of malware is a very costly and time-consuming process. As more malware variants are evolved by hackers who occasionally use a copy-paste-modify programming style to accelerate the generation of large number of malware, the effort spent in analyzing similar pieces of malicious code has dramatically grown. One approach to remedy this situation is to automatically perform similarity analysis on malware samples and identify the functions they share in order to minimize duplicated effort in analyzing similar codes of malware variants.

In this thesis, we present a framework to match cloned functions in a large chunk of malware samples. Firstly, the instructions of the functions to be analyzed are extracted from the disassembled malware binary code and then normalized. We propose a new similarity metric and use it to determine the pair-wise similarity among malware samples based on the calculated similarity of their functions. The developed tool also includes an API class recognizer designed to determine probable malicious operations that can be performed by malware functions. Furthermore, it allows us to visualize the relationship among functions inside malware codes and locate similar functions importing the same API class. We evaluate this framework on three malware datasets including metamorphic viruses created by malware generation tools, real-life malware variants in the wild, and two well-known botnet trojans. The obtained experimental results confirm that the proposed framework is effective in detecting similar malware code.

Acknowledgments

I would like to express my deepest appreciation to all those who gave me the possibility to complete this thesis.

Foremost, I offer my sincerest gratitude to my supervisor, Dr. Amr Youssef, for his patience, support, enthusiasm, and knowledge. I could not have imagined having a better supervisor for my Master's study.

My sincere thanks goes to my friends and lab-mates at Concordia University, especially Honghui, Tengfei, Weiyi, Saurabh and Dhruv. I also owe a debt of gratitude to everyone of my professors for their great teaching, which has also inspired my research.

Last but not the least, special thanks to my family for supporting me emotionally and financially. Without their unconditional love and persistent encouragement, none of this would be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Malicious software, commonly known as malware, is any software that is specifically designed to gain access or disrupt to computer systems without the knowledge of their owners. Examples of malware include viruses, trojan horses, worms, rootkits, and spyware. Hackers are no longer using malware for fun or fame. Over the last few years, cyber criminals, who are actively involved in a rich underground economy, have been using for-profit category of malware to steal sensitive information, spread email spam and launch distributed denial-of-service attacks.

With the heavy reliance of our society on computers and the rising popularity of the Internet, the number of malware has increased significantly. According to recent reports, 20% of all of malware that ever existed was created in 2013 with 30 million new malware threats in one year, or about 82,000 per day [51]. In many cases, hackers do not create new malware from scratch; they develop malware variants from previous ones and use some code obfuscation techniques that allow them to avoid detection by anti-malware and security software. With the large number of malware variants encountered in the wild, efficient analysis of malware is becoming increasingly critical objective for security researchers and organizations. Most often, when performing malware analysis, the analysts only have the non-human-readable malware executables. Interactive disassemblers, such as IDA Pro [2], are commonly used by analysts to dissect and understand malware samples. Despite the fact that these tools can facilitate many tedious steps in the code analysis process, the reverse engineering procedure of malware is still technically very involved and extremely time consuming. When trying to analyze a large number of malware samples, the efficiency of the reverse engineering process can be improved by noting that, in many occasions, malware writers use a copy-paste-modify programming style to accelerate the generation of this large number of malware variants. Thus one way to speed up the analysis of malware is to identify identical or similar code regions in order to save the effort of the analysts and make them focus on new parts of the code that have not been analyzed before. On the other hand, trying to manually search for similar pieces of codes among a large number of malware samples is a time consuming process and successful identification of similar code regions or functions depends heavily on the experience and knowledge of the analysts. In this thesis, we investigate the problem of automated similarity analysis of malware codes. In particular, the objectives of this thesis can be summarized as follows:

- Investigate methods and techniques that can be used to facilitate the process of malware similarity analysis.
- Design and implement a framework to automatically identify similarity between malware samples.
- Develop a tool for visualizing the relationship between malware samples and identifying similarity between their functions.

1.2 Contributions

Our contributions can be summarized as follows:

- We proposed a new metric to measure pair-wise similarity between malware samples. After identifying and extracting the functions within each sample, similarity scores are first computed at the functions level and then these scores are used to determine the degree of containment of each sample within the other. Finally, the containment scores are used to determine an overall similarity between the considered samples.

- We designed and implemented a framework to automate the process of malware similarity analysis. The tools developed within this framework also allow us to visualize the relationship between functions inside malware codes and locate similar functions importing the same API class.

- We evaluated this framework on three malware datasets including metamorphic viruses created by malware generation tools, real-life malware variants in the wild, and two well-known botnet trojans. As confirmed by our experimental results, the proposed framework is effective in detecting similar malware code and in identifying similar functions in malware variants.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides a literature review and background information on obfuscation techniques, reverse engineering and similarity analysis techniques. In Chapter 3, we describe the implementation details of the proposed framework including pre-processing of analyzed malware, similarity detection methods at the function level and metrics used to evaluate the overall similarity between malware samples. Chapter 4 presents our experimental results. Finally, Chapter 5 provides our conclusions and suggestions for some possible future research directions.

Chapter 2

Related Work and Background

In this chapter, we briefly review some related work and provide background knowledge required to understand the work done in this thesis. First, we present a classification of malware based on the techniques used by malware writers to avoid antivirus detection. Then, some code obfuscation techniques are summarized and categorized. After discussing the methods used by malware to escape generic scanning, we introduce steps and tools used for reverse engineering of malware. We also examine metamorphic detection and three intellectual property protection applications: clone detection, plagiarism detection, and birthmark detection, and show how these techniques can be used for malware analysis. Finally, we describe some malware similarity analysis algorithms.

2.1 Malicious Software

2.1.1 Malware Types

In their race with security researchers and anti-malware tools, hackers utilize several obfuscation techniques in order to circumvent detection techniqes. Depending on the concealment techniques, malware can be classified into three categories: encrypted, polymorphic, and metamorphic [58]. These three categories are discussed in the following subsections.

Encrypted Malware

Encryption is one of the simplest methods to avoid signature-based detection which is widely used by antivirus software. An encrypted malware generally has encryption/decryption engine, encrypted malware code, and a decryption key embedded in its code. The malware uses the decryption engine with the associated key to decrypt the encrypted malware code before execution. After the infection process, the decrypted malware code is re-encrypted with a newly generated key to evade simple signature analysis. This new key then replaces the original decryption key in the maleware body. Figure 1 provides a pictorial illustration of an encrypted malware before and after the decryption process. Since the encryption/decryption engine usually remain the same, it is possible to detect this category of malware by creating a signature of the encryption/decryption routines. The

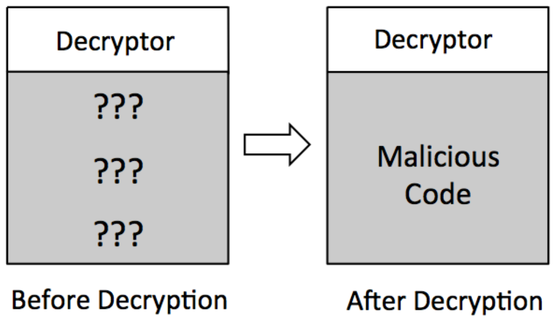


Figure 1: Encryted malware before and after decryption

simplest method to encrypt malware is to XOR the body of the malware with a predefined key stream. This approach is used not only to encrypt (or decrypt) the malware body but also to hide specific strings such as monitored domain names, the attacker's IP address, and the initial communication key. Figure 2 illustrates a simple (non secure) XOR encryption where a single secret byte is used to encrypt/decrypt the malware.

```
decrypt_malware_code:
mov   dh, dl              ; clone the key
mov   ecx, 3810           ; number of bytes to decrypt
loop_xor:
xor   [eax], dl
inc   eax
add   dl,dh               ; update key
loop  loop_xor
retn
```

Figure 2: An example of a decryptor function

Polymorphic Malware

In order to overcome the weaknesses associated with encrypting malware with identical encryption/decryption routines, malware writers developed polymorphic malware that has encrypted malware code and morphed decryption code. Hence it is difficult to be detected by antivirus scanner using a specific signature of the encryption/decryption engine. Variants of a polymorphic malware keep their inherent functionality the same. The first polymorphic malware is a virus, called 1260, which is a .COM infector developed by Mark Washburn in 1990 [71]. Figure 3 illustrates how the structure of polymorphic malware changes from one generation to the other, where G, D, and V donate the generations of polymorphic malware, decryption engine, and malware body. The simplest polymorphic technique is

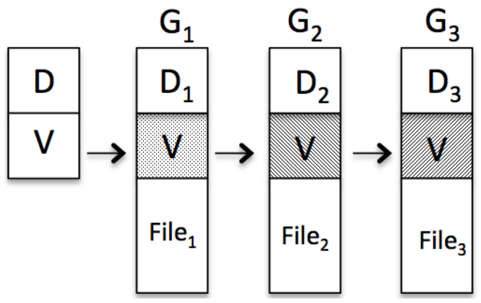


Figure 3: Polymorphic malware replication

to insert junk code or substitute instructions in the decryption engine [43]. However, this category of malware can be detected using code emulation techniques [62]. The encrypted malware code will be revealed when polymorphic malware is executed in code emulators. The decrypted body can then be used to create a signature for detection.

Metamorphic Malware

Unlike encrypted and polymorphic malware, metamorphic malware contains changing body code without any decryptor. This category of malware employs a mutation engine [48] that can change the entire malware body. The mutation engine performs code obfuscation to generate a completely different variant with the same malicious behavior.

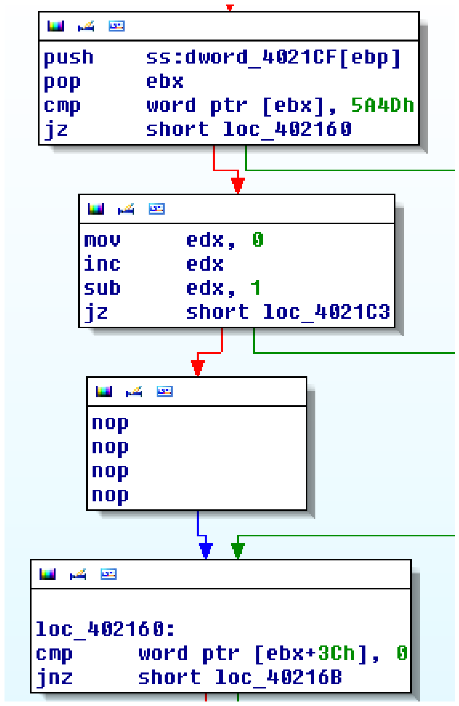
Professional metamorphic malware need to be implemented carefully by malware writer. The metamorphic variants should be unpredictable in size and the operations of mutated functions should remain efficient and correct [73].

2.1.2 Obfuscation Techniques

Metamorphic malware employs obfuscation techniques to generate different variants with the same functionality in order to avoid signature-based detection. This section describes some of the obfuscation techniques used by malware writers [57, 91].

Insertion of Junk Code

The insertion of junk code is simple and yet somewhat effective approach to modify malware body or size without affecting the function or program outcome. Malware writers can insert single or a block of do-nothing operations between malicious instructions to confuse antivirus software and some primitive reverse engineering tools. The junk code can be classified into two categories depending on whether they modify the content of CPU registers or memory [8, 14]. Table 1 shows several examples of junk instructions belonging to the first category in which the instructions are equivalent to no-operation such as *nop*. Figure 4 shows an example for junk code inserted in one variant of the NGVCK metamorphic virus family.



```
push    ss:dword_4021CF[ebp]
pop     ebx
cmp     word ptr [ebx], 5A4Dh
jz      short loc_402160

mov     edx, 0
inc     edx
sub     edx, 1
jz      short loc_4021C3

nop
nop
nop
nop

loc_402160:
cmp     word ptr [ebx+3Ch], 0
jnz     short loc_40216B
```

Figure 4: Example of junk code inserted in the metamorphic virus (A snapshot from IDA Pro)

Instruction	Operation
mov eax, eax	eax ← eax
or ecx, 0	ecx ← ecx | 0
and ebx, -1	ebx ← ebx & -1
add edi, 0	edi ← edi + 0

Table 1: An example of no-operation instructions [14]

The second category of junk code performs operation on registers and memory. However, before it alters the outcome of the target function or program, undo instructions restore the status of these affected registers or memory locations. Table 2 illustrates an example for this category of junk code insertion.

Instruction	
push Reg$_1$	; *push value of Reg$_1$ into stack,*
... ...	; *before it affects outcome of function*
pop Reg$_1$	; *Reg$_1$ must be restored*
inc Reg$_2$	; *increase value of Reg$_2$ first,*
... ...	; *before any usage of Reg$_2$*
sub Reg$_2$, 1	; *it must be restored to previous status*

Table 2: An example for the second category of junk code [57]

Instruction/Subroutine Permutation

The permutation technique is another solution to modify the internal structure of malware while keeping its original malicious functionality. This technique can perform obfuscation in malware at two levels, namely instruction permutation [58] and subroutine permutation [68]. The instruction permutation reorders instructions that have no dependency between them. The re-ordered instructions make the functions or programs look dissimilar but keep the functionality unchanged. Table 3 depicts an example for this technique where the two columns of instructions produce the same result but in different order. Advanced instruction permutation can use *jmp* to create different sequences of instructions.

Instruction Order 1	**Instruction Order 2**
mov edx, 0	add ecx, 03h
push eax	mov edx, 0
add ecx, 03h	push eax

Table 3: An example of instruction permutation

The subroutine permutation reorders subroutines instead of instructions. If a malware consists of n subroutines, $n!$ variants of the malware can be generated by this technique. For example, the Win32/Ghost virus [14] with 10 subroutines employs this permutation to generate $10! \approx 3.6$ million variants of itself.

Register Swapping

Register swapping is one of the simplest methods used by mutation engines of metamorphic malware. This method simply replaces registers in the instructions with different equivalent registers, but the mnemonics of the instructions are kept the same for all variants. The W95/RegSwap virus is a good representative for metamorphic malware that employs this technique [71]. Figure 5 shows how register swapping is used in two variants of the NGVCK metamorphic family. Malware variants produced with this technique can be detected with wildcard-based scanning [71], which can skip bytes or byte ranges to detect the common areas of the two code variants.

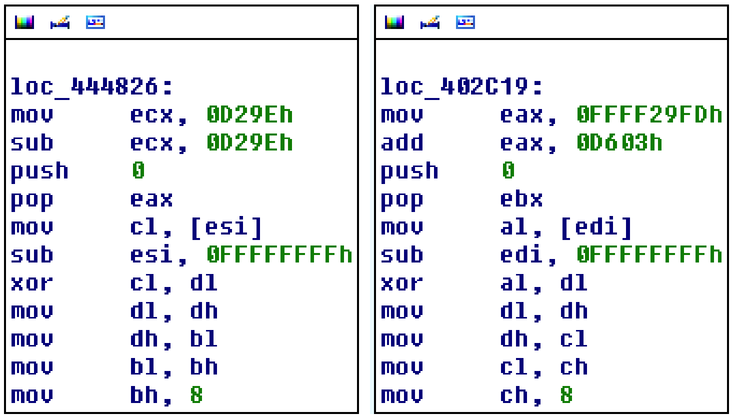

```
loc_444826:
mov     ecx, 0D29Eh
sub     ecx, 0D29Eh
push    0
pop     eax
mov     cl, [esi]
sub     esi, 0FFFFFFFFh
xor     cl, dl
mov     dl, dh
mov     dh, bl
mov     bl, bh
mov     bh, 8
```
```
loc_402C19:
mov     eax, 0FFFF29FDh
add     eax, 0D603h
push    0
pop     ebx
mov     al, [edi]
sub     edi, 0FFFFFFFFh
xor     al, dl
mov     dl, dh
mov     dh, cl
mov     cl, ch
mov     ch, 8
```

Figure 5: Example of register swapping in the metamorphic virus (A snapshot from IDA Pro)

Instruction Substitution

Instruction substitution is another technique used to generate metamorphic malware. This technique substitutes a single instruction or a block of instructions with some of their equivalent instructions. For example, all the four instructions in Table 4 perform the same operation of resetting the content of a register to zero. The malware writer tends to use complicated replacement strategy to morph malware variants because well-constructed substitutions in malicious code can make the process of understanding the malware harder and more time-consuming for security professionals and researchers. Table 5 shows other substitution patterns used in metamorphic malware [14, 88].

Equivalent Instructions
and Reg, 0
mov Reg, 0
xor Reg, Reg
sub Reg, Reg

Table 4: Example of different approaches for resetting a register to zero [57]

Original Instructions	Equivalent Instructions
mov Reg, Imm	push Imm pop Reg
push Reg_1 mov Reg_1, Reg_2	push Reg_1 push Reg_2 pop Reg_1
op Reg_1, Reg_2	mov Mem, Reg_1 op Mem, Reg_2 mov Reg_1, Mem

Table 5: Instruction substitution examples [14]

2.2 Reverse Engineering of Malware

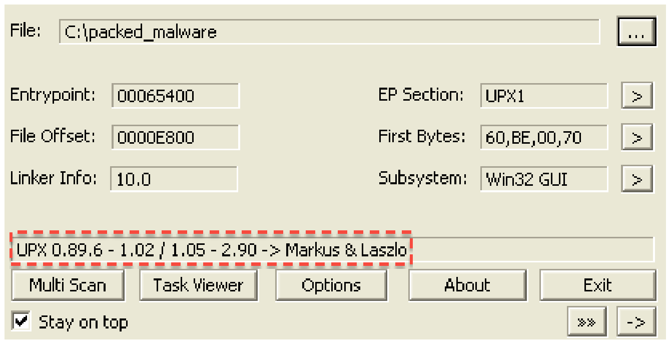
2.2.1 Unpacking

Malware in the wild exists in binary form at the machine code level that computers can run quickly and efficiently. Reverse engineering is one of the strongest weapons used by malware analysts against malware writers. Reverse engineering of malware can help security researchers to analyze malicious code, determine how dangerous they are, and learn how to defeat them. Sometimes, at the beginning of this process, unpacking the malware is necessary if the malware was packed in order to evade signature-based antivirus and prevent simple static analysis.

Identifying the Packer

The first step of unpacking is to detect whether the malware is packed. There are several techniques we can use for this purpose [36]. The simplest way is to notice these signs from the analyzed malware:

- The malware with large size has very small import table, and particularly if the only imports are *LoadLibraryA* and *GetProcAdress*, which can be used to locate other functions.
- After the disassembly process, only a small amount of code is recognized by the automatic analysis and a large amount of data appears inside the executable.
- The disassembled malware shows section names that are strange or indicate a particular packer (such as *UPX0*, *UPX1*, and *UPX2*).
- The string table of disassembled malware is missing or it contains only garbage.
- The disassembled malware has abnormal section size, such as a *.text* section with a Size of Raw Data equals 0.
- The malware is opened in some tools with a warning information related to a specific packer, such as OllyDbg [92] and PEiD [70]. Figure 6 shows an example of packer identification in PEiD, where a *packed_malware* file is packed with UPX version 0.896-1.02 or 1.05-2.90.


File: C:\packed_malware

Entrypoint: 00065400
File Offset: 0000E800
Linker Info: 10.0
EP Section: UPX1
First Bytes: 60,BE,00,70
Subsystem: Win32 GUI

UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo

Figure 6: A snapshot of the packed malware identified in PEiD

Unpacking Techniques

After identifying packed malware, we mainly have two methods for unpacking it: automated unpacking and manual unpacking. Compared to manual unpacking, the automated unpacking is faster and easier but it only targets some specific identified packers.

Automated unpacking programs decompress or decrypt the packed malware and restore it to its original state. These programs only work if the malware writer use packers that are not designed to thwart analysis. One of the most commonly used malware packers is the Ultimate Packer for eXecutables (UPX) [49], which is open source, free, and easy to use. UPX can compress the malware and is designed for performance rather than security.

When automated unpacking fails, manual unpacking has to be performed on packed malware. The two common methods to manually unpack a malware can be summarized as follows:

- Locate the packing algorithm used by the malware and write a program to run it in reverse. When running the algorithm in reverse, the program will undo each of the steps of the packing program. This method is time-consuming since the program written to unpack the malware will be specific to the used packing program.

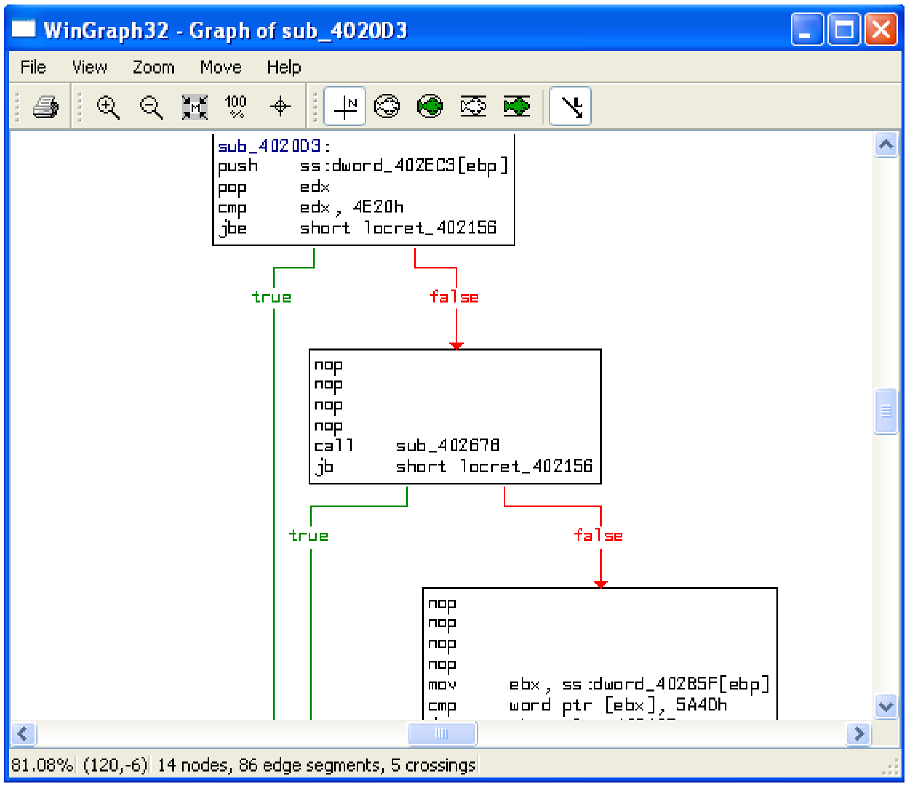
- Run the packed malware in a safe environment, such as a virtual machine, so that the unpacking stub loads the original malware, and then dump the process out of memory, and manually fix the PE header. In order to correct the PE header, we have to reconstruct the import table and change the entry point value in the header for pointing to the Original Entry Point (OEP). Debuggers with the memory dump plug-in can be used for this method, e.g., OllyDbg [92] and Immunity debugger [31].

2.2.2 Disassembly

After obtaining the unpacked malware, different forms of analysis can be performed on the original malicious code. Malware disassembly is the process of taking malware binary as input and converting it back to its assembly code. Disassembly is a processor-specific process, but some disassemblers support multiple CPU architectures. For malware analysts, a high-quality disassembler is an important component in the toolkit to perform the analysis. In this section, we briefly discuss two powerful disassemblers which are commonly used by many malware analysts.

IDA Pro

IDA Pro [2] is probably the most popular disassembler used for reverse engineering of malware. It supports most executable types, such as Portable Executable (PE), Executable and Linking Format (ELF), Dalvik Executable (Dex), and Mach-O. IDA Pro can automatically identify the data and code sections of a program. It is capable of auto-commenting code, displaying graphical relationships between code jumps, and automatically recognizing imported APIs from standard library. A flowchart for each identified function of the malware is produced by IDA Pro, which is essentially a logical graph that shows chunks of disassembled code and provides a visual representation of how each conditional jump in the code affects the function's flow. Figure 7 shows an example for an IDA Pro-generated flowchart for one function instance of the NGVCK metamorphic virus family, where each box represents a code snippet or stage in the function's flow and the boxes are connected by arrows that show the flow of the code based on whether the conditional jump is satisfied or not. As an interactive disassembler, IDA Pro allows analysts to modify, manipulate, and redefine all aspects of the disassembly process. For example, when performing the analysis, the analyst can add comments on instructions related to malicious operations, label data that represents initial encryption key, and name a function that is used to infect DLL files. All the analysis results can be saved in an IDA Pro database (*.idb*) to return to later. IDA Pro also supports scripts or plug-ins written by analysts to extend its functionality.



```
sub_4020D3:
push    ss:dword_402EC3[ebp]
pop     edx
cmp     edx, 4E20h
jbe     short locret_402156
```

```
nop
nop
nop
nop
call    sub_402678
jb      short locret_402156
```

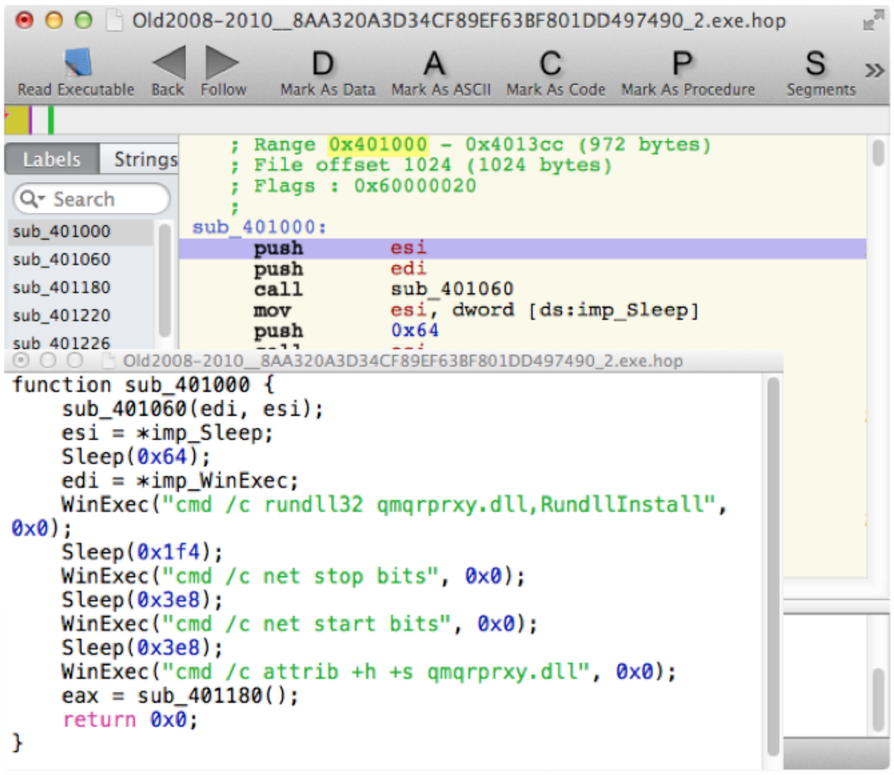
```
nop
nop
nop
nop
mov     ebx, ss:dword_402B5F[ebp]
cmp     word ptr [ebx], 5A4Dh
```

```
81.08% (120,-6) 14 nodes, 86 edge segments, 5 crossings
```

Figure 7: A snapshot of the IDA Pro-generated flowchart for one function in the metamorphic virus

Hopper

Hopper [11] is a reverse engineering tool which can handle 32-bit or 64-bit programs for both Windows and OS X. Hopper is perfectly adapted to the Mac OS X environment and hence it is a desirable option for the analysis of Mac malware. Control flow graphs (CFGs) can be generated for identified functions during the disassembly process. Based on an advanced understanding of the executable, Hopper provides a pseudo-code representation of the functions identified in the executable. This pseudo-code can help the analyst to better understand the malware behavior.



```
        ; Range 0x401000 - 0x4013cc (972 bytes)
        ; File offset 1024 (1024 bytes)
        ; Flags : 0x60000020
        ;
sub_401000:
        push        esi
        push        edi
        call        sub_401060
        mov         esi, dword [ds:imp_Sleep]
        push        0x64
```

```
function sub_401000 {
    sub_401060(edi, esi);
    esi = *imp_Sleep;
    Sleep(0x64);
    edi = *imp_WinExec;
    WinExec("cmd /c rundll32 qmqrprxy.dll,RundllInstall",
0x0);
    Sleep(0x1f4);
    WinExec("cmd /c net stop bits", 0x0);
    Sleep(0x3e8);
    WinExec("cmd /c net start bits", 0x0);
    Sleep(0x3e8);
    WinExec("cmd /c attrib +h +s qmqrprxy.dll", 0x0);
    eax = sub_401180();
    return 0x0;
}
```

Figure 8: A snapshot of the disassembly code of BISCUIT backdoor obtained using Hopper

Figure 8 shows the pseudo-code of one function that is generated when we disassembled and analyzed the BISCUIT malware which is a kind of Advanced Persistent Threat (APT) backdoor. This malware provides the attackers with full access to infected hosts and is capable of launching an interactive command shell, enumerating processes and servers on Windows networks, and transferring files. In the pseudo-code, we can see that function *sub_401000* uses a command line to installs *qmqrprxy.dll* which replaces the legitimate BITS (Background Intelligent Transfer Service) file, restarts BITS, and sets the attributes of *qmqrprxy.dll* to system hidden.

2.2.3 Features of Binary Code

Malicious codes in malware can be accessed after the disassembly process. A disassembler with the capability of binary parsing can provide different representations of binary code such as byte-level, instruction-level, and structure-level representation. In order to analyze the relationship between different malware, we need to define features of malicious codes. Based on these three representations, different code features can be extracted from malicious binaries for related tasks such as detecting similarity between malware and identifying malicious programs in the wild. In this section, we discuss some of the features that are commonly extracted to be used for malware analysis from these three binary code representations.

Byte-level Features

Most antivirus applications detect malware using signatures extracted from the byte level representation of malicious binaries. A simple, and yet powerful, feature that we can extract from the byte level representation is *n*-grams, which are sequences of bytes of length *n*. For example, for $n = 2$ and 3, the sequence of bytes in malware binary(F6 E0 00 56 66 C3) can be represented, respectively, as:

$$\text{Bigrams: } \langle \text{F6 E0} \rangle, \langle \text{E0 00} \rangle, \langle \text{00 56} \rangle, ...$$
$$\text{Trigrams: } \langle \text{F6 E0 00} \rangle, \langle \text{E0 00 56} \rangle, \langle \text{00 56 66} \rangle, ...$$

The frequency of the n-gram features extracted from binaries can be used to distinguish different file types [42] and detect malware [44]. The main problem with this type of byte code analysis is the lack of generalizations. For example, polymorphic and metamorphic malware can change their byte level content due to mutation, recompilation, and code modification [66]. Another limitation of *n*-gram features is the loss of long-range information within analyzed binaries. In other words, significant relationships between disparate code cannot be extracted by short *n*-grams.

Instruction-level Features

Compared to byte-level representation, the instruction-level representation of binary code is more semantically meaningful for the human analyst. In the Intel IA-32 instruction set, a single instruction can span up to fifteen bytes [23]. One analysis method based on instructions is to fingerprint malware using opcode distribution [12]. N-gram analysis can also be applied to opcode sequences [35] in order to detect similarity between malware instances. The code normalization process is always applied on disassembled instructions before performing malware analysis. For example, in [90], the X86 instructions are normalized by classifying them into 15 groups based on their functionalities when computing similarity between functions inside malware code. In order to take into account the variations in operands of instructions, a normalization process is implemented on constants, memory addresses and registers of operands. Table 6 illustrates an example for the instructions normalization process used in [19].

Disassembled Instructions	**Normalized Instructions**
mov edi, edi	mov REG, REG
push ebp	push REG
push ebp, ebp	push REG, REG
mov exa, dword ptr [ebp+8]	mov REG, MEM

Table 6: An example of operands normalization

Structure-level Features

Although the normalization process can be done on instructions to reduce binary code variations, instruction-level representations are still sensitive to code modification such as instruction reordering and substitution. Using higher-level representation of the binary is one solution. A method that uses control flow characteristics of functions to detect differences between two versions of the same program was proposed in [28]. As a further extension to this method, new structural characteristics such as block size and specific instruction information are incorporated in [63]. In order to detect polymorphic and metamorphic malware using structural features of binaries, the authors in [40] proposed to fingerprint malware based on *k-subgraphs* of program control flow graphs. For the generated subgraph, a fingerprint is derived using an adjacency matrix (see Figure 9), which is then used for searching for similar structures in polymorphic worms. Besides the structural characteristic, the fingerprint from [40] also includes information about the instructions by coloring each node based on the instructions in the corresponding basic block.

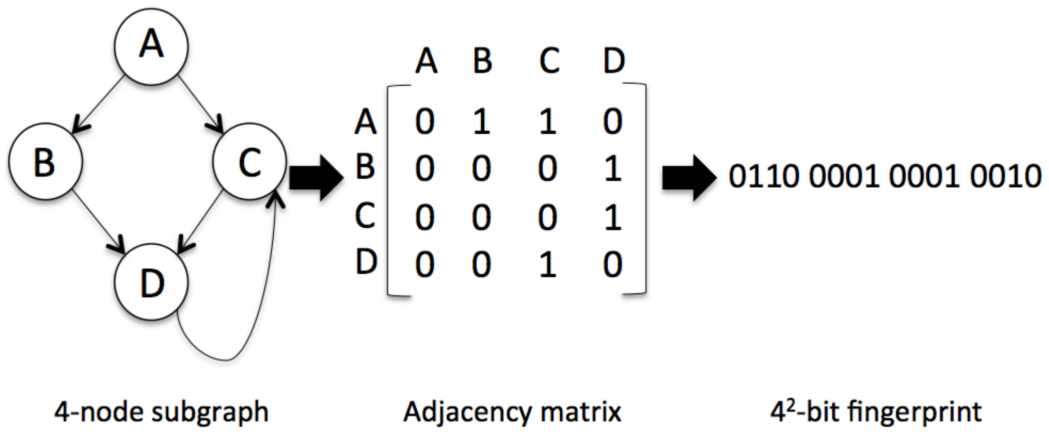


Figure 9: Generating a fingerprint from a subgraph [40]

A malware classification technique based on dynamically constructed system call dependence graphs was employed in [52]. The behavioral graph is extracted by tracing system calls at runtime, which represents a set of dependent system call sequences. The similarity between behavioral graphs is computed using the maximal common subgraph (MCS).

2.3 Similarity Analysis

2.3.1 Applications

In this section, we first discuss three applications related to software protection where the adversary copies important pieces of code from the victim's program and incorporates original or modified code into her program. In the three applications, the fundamental operation is determining how two programs are similar to each other and whether one program is contained within the other. In malware research, related techniques are applied to determine similarity between malicious programs. Finally, some techniques that are used to solve the metamorphic detection problem are presented.

Clone Detection

Code duplication and then reuse by pasting with or without modification exist both in software development and in malware development. For these two cases, clone detection is employed for different purposes.

In the software development industry, the duplicated code is often the result of a *copy-paste-modify*-style of programming; a programmer searches for existing code on the Internet, copies a desirable one, specializes it as necessary, and pastes the copy to her program. Clone detection is to identify similar pieces of code in given program(s).

Sæbj∅rnsen et al. [13] proposed a practical clone detection algorithm for binary executables. First, the input binaries are disassembled and transformed to the intermediate representation. Then, a normalization process creates abstract format for instructions. In the clone detection process, the authors use exact clone detector that only returns identical normalized instruction sequences and inexact clone detector that tolerates certain difference. In inexact clone detection, they adapted the basic approach implemented by Jiang et al. [32], where the detector extracts feature vectors from normalized instruction sequences and groups similar vectors to find clones.

Compared to code duplication in software, the code reuse in the production of malware happens when malware writers exchange source code among them to develop different malware variants. Here, a copy-paste-modify programming can help malware writers achieve similar malicious functionality in their own malware. Clone detection targeting malware can help the analysts save their effort in reanalyzing similar pieces of code belonging to new malware variants by identifying the similarity between the new samples and the old, already analyzed, ones.

Charland et al. [19] developed a prototype for a clone search system for malware analysis, which expands on the previous work presented in [13]. They implemented a flexible normalization scheme and a new inexact clone detection method that can efficiently identify inexact clone pairs. The experimental results suggested that the implemented system can effectively identify both exact and inexact clones in the assembly code of malware.

Rahimian et al. [59] performed clone-based analysis on Citadel and Zeus to quantify the similarity between them. The analysis results show that these two malware share 526 exact binary clones and Citadel duplicates a significant amount of codes from Zeus. An inexact clone is also detected in the RC4 function modified from the original one in Zeus, which is used for encrypting the C&C traffic.

Plagiarism Detection

Plagiarism in programming most often happens in assignments for computer science classes where a student may copy an already existing program and make changes to it in order to hide the origin of the code. Common modifications consist of renaming of identifiers, reordering of functions, and replacing of equivalent statements. As depicted in Figure 10, in plagiarism detection, the detector makes pair-wise comparisons between all of the input programs and orders them from the most to least similar.

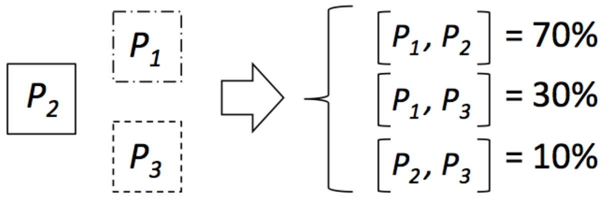


Figure 10: Plagiarism detection process

Most approaches used in plagiarism detection relied on the concept of *n*-grams that can be considered as a group of *n* consecutive items from a sequence. This technique is also applied to malware detection. Oprisa et al. [50] presented a unified approach to detect the plagiarism cases in programming assignments and to cluster malicious samples for making the analysis considerably simpler. As illustrated in Figure 11, they extracted features in the form of opcode sequences from the binary code, where Σ represents a sequence of symbols transformed from opcode. Then they employed four similarity metrics including descriptional entropy, normalized compression distance, common *n*-grams, and weighted common *n*-grams, to compute the similarity between two sequences. The proposed system showed good results for the task of clustering malware.

Kim et al. [38] proposed a malware detection system which combines a genetic algorithm (GA) and a metric-based method. Metric-based methods use a numerical vector to represent the characteristics of a program and work well in code plagiarism detection [41]. The proposed system is illustrated in Figure 12 where *Decision Algorithm* determines whether the input program is malicious or not, *Malicious Core Finder* uses a genetic algorithm and extracts the malicious part of the program which is the most similar to the given malware, *Metric Calculator* converts programs to numerical vectors containing various metrics, and finally *Distance Calculator* measures the distance between the vectors.

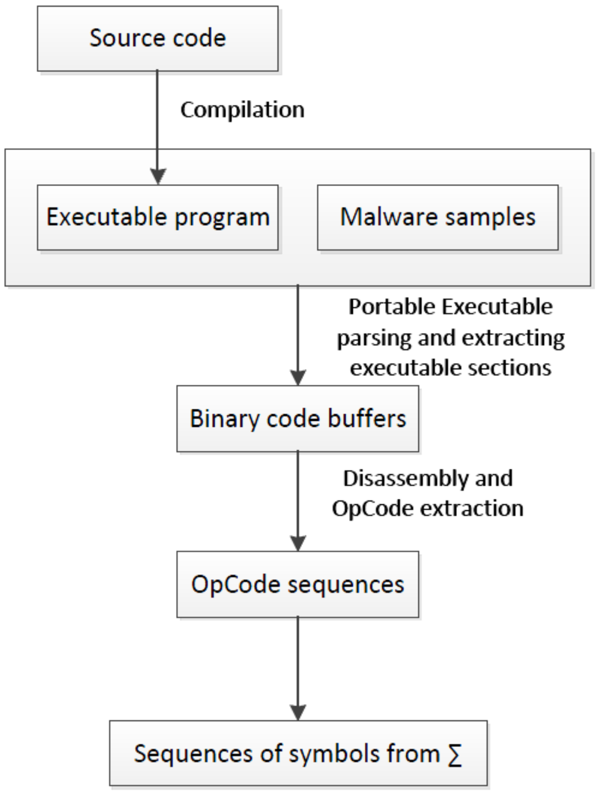

Source code
Compilation
Executable program
Malware samples
Portable Executable parsing and extracting executable sections
Binary code buffers
Disassembly and OpCode extraction
OpCode sequences
Sequences of symbols from ∑

Figure 11: Transforming a program into a set of symbols [50]

Birthmark Detection

Birthmarking is also concerned with detecting similarities between programs, but it differs in some respects from clone detection and plagiarism detection. First, we generally extract birthmarks from executable code, such as X86 binary or bytecode. Second, birthmark detection assumes of a much more active and competent adversary. In particular, in birthmark detection, the assumption is that the adversary uses the following steps to copy code from some program P into her own program Q [22]:

1. Copy one or more code sections from P into Q.

2. Compile Q, as necessary, into binary code or bytecode.

3. Apply semantics-preserving transformations, such as obfuscation and optimization, to Q and distribute the resulting program.

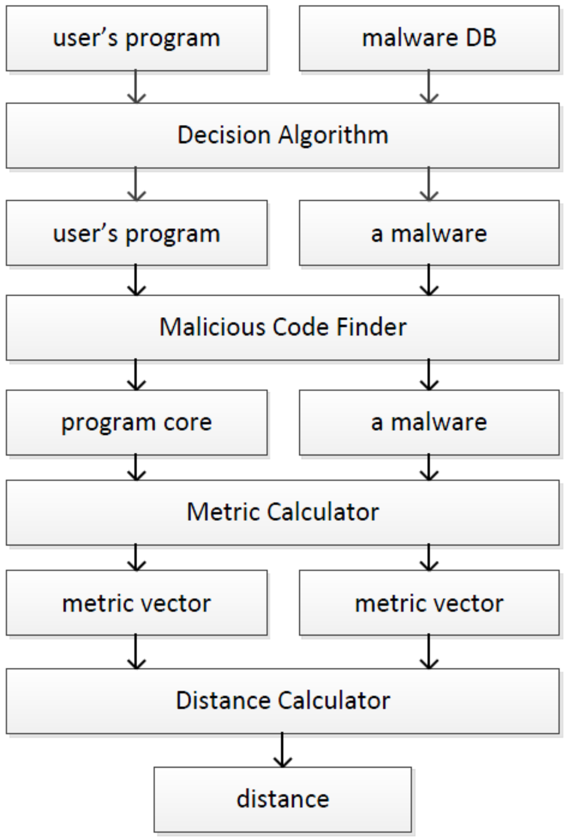


Figure 12: Overview of the malware detection system [38]

Birthmark detection is to extract properties of code that are invariant under obfuscation or other transformations and determine if two programs are similar. As shown in Figure 13, the signatures bm_P and bm_Q are extracted from two programs P and Q to determine their similarity.

Choi et al. [20] proposed a static API birthmark for Windows executables that utilizes API calls identified by the disassembler. The birthmark system computes the birthmark of the target function using call graphs of the underlying program. The birthmark is a set of API calls of the function and its descendant functions, which correspond to the descendant nodes of the function in the call graph. Based on the birthmarks extracted from functions, the system computes similarity between functions using Dice's coefficient and

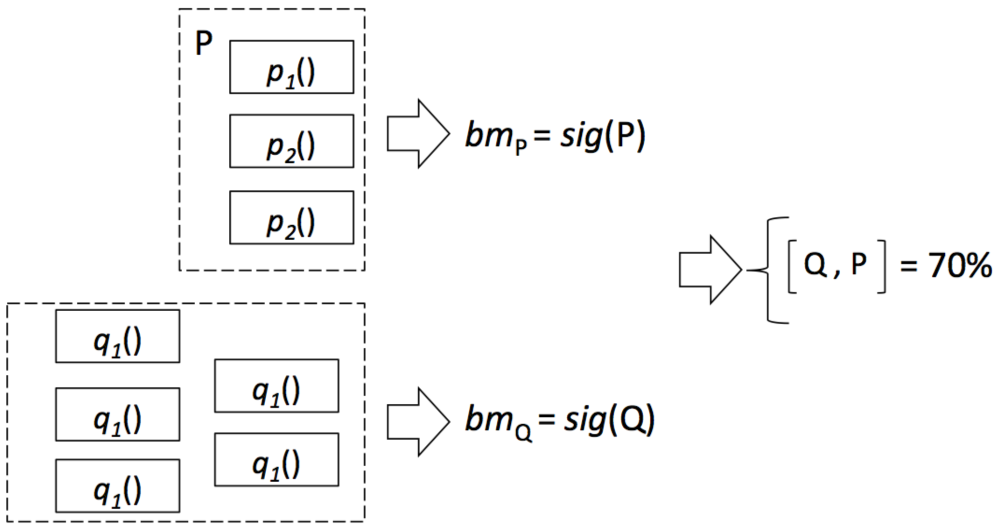
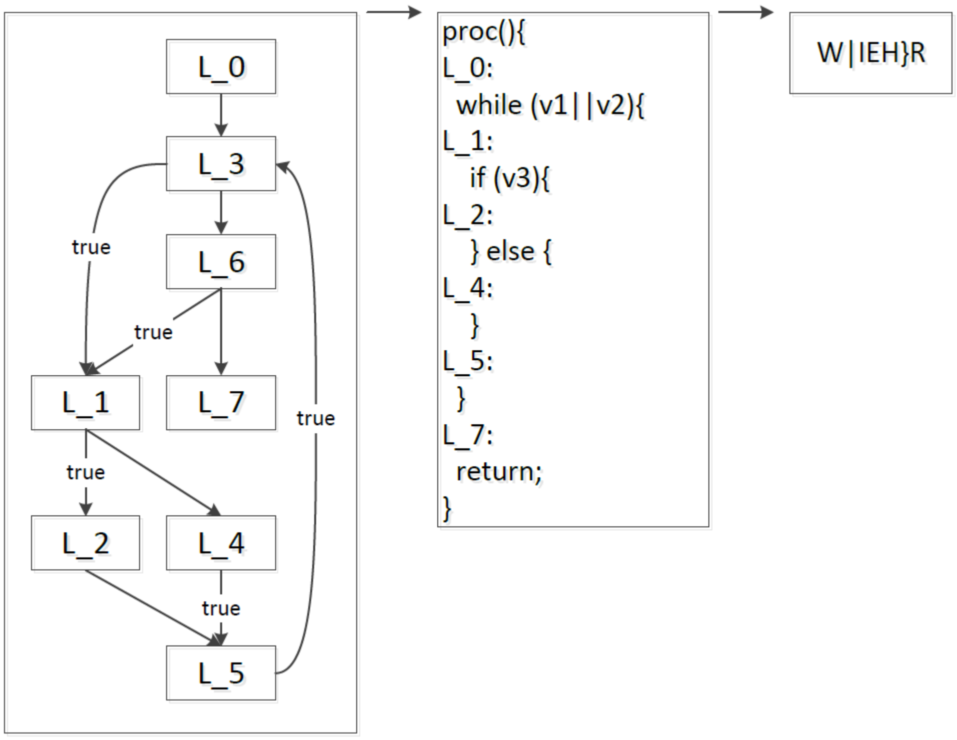


Figure 13: Birthmark detection process [22]

concludes similarity between programs by finding the maximum weighted bipartite matching. The proposed system was evaluated by comparing 49 Windows executables and the results showed that it can distinguish similar programs and detect copies.

The similar birthmark technique is also used in malware detection. Cesare et al. [18] developed a malware classification system which builds a birthmark of malware based on the set of control flow graphs it has and compares birthmarks using distance metrics. In order to match malware variant to existing malware, they first used structuring algorithm to transform the control flow graphs into a structured graph that is used to obtain string representation, as illustrated in Figure 14. Afterwards, the pre-filtering algorithm is applied to provide a list of potentially related malware by computing distance between feature vectors. Cesare et al. used k-subgraphs and q-grams of structured control flow to extract features. The authors concluded that the q-gram feature based on structuring string is more efficient and accurate.



```
proc(){
L_0:
 while (v1||v2){
L_1:
  if (v3){
L_2:
  } else {
L_4:
  }
L_5:
 }
L_7:
 return;
}
```

Figure 14: Structuring algorithm transforms CFG into string [18]

Metamorphic Detection

Metamorphic detection is applied to detect malware that employ metamorphic techniques that make signature-based detection virtually impossible [14]. In what follows, we briefly describe several techniques that were used to detect metamorphic malware with some success. In [33], a technique based on Singular Value Decomposition (SVD) was applied to the problem of metamorphic detection. The proposed method generates Singular Values and Singular Vectors from the training malware set. The Euclidean distance between the weights of each testing file and weights of all training files is computed by projecting files on to singular space. This SVD-based technique proved to be effective in detecting three metamorphic malware families (NGVCK, MWOR and G2).

A Hidden Markov model (HMM) approach for malware classification is proposed in [7]. HMMs were trained for a variety of malware generators, including MWOR and

NGVCK, and a variety of compilers. The scores generated from HMMs for over 9000 malware samples were used as input to a *k*-means clustering algorithm. The experimental results showed that HMM is an effective tool to automatically classify malware.

A graph-based approach which uses function call graph for the metamorphic malware detection is proposed in [25]. This method detects similar function pairs from two malware variants by computing the Cosine similarity based on opcodes and a graph coloring technique. Similarity between two function call graphs with the common vertex pairs is calculated based on common edges. The detection technique is tested on the NGVCK and MWOR virus families and the results showed that it is better than the other graph based technique proposed in [64].

Based on software similarity, Shanmugam et al. [68] proposed a method for detecting metamorphic malware which use simple substitution ciphers. The opcode sequences of malware samples are extracted for constructing digraph distribution matrices, which are input to a hill climbing heuristic optimization search. The method computes similarity score with recovered keys. Baysa et al. [9] applied a technique based on structural entropy to metamorphic detection. This technique, which analyzes variations in the complexity of data within a file, consists of two steps. First, it uses entropy measurements and Wavelet analysis to segment the file. Then, the similarity of files is measured by computing the edit distance between segmented sequences.

In [53], metamorphic detection was carried out using a similarity index technique based on edit distance and pairwise sequence alignment. The edit distance between two opcode sequences extracted from files is computed by replacing each opcode with a corresponding symbol. In pairwise sequence alignment, a scoring matrix [45] is used to compute similarity score between two files by aligning the corresponding opcode sequences.

Code emulation is a dynamic analysis technique in which malware is allowed to execute in a simulated environment without actually impacting the host machine. In [56], a code emulator is designed specifically to detect dead code in virus files. The output of the code emulator can be used to enhance HMM-based metamorphic detection by removing the unexecuted instructions/subroutines and the instructions that are produced by code obfuscation techniques. The metamorphic viruses, which previously cannot be detected using HMMs technique, are detected by un-morphing in the code emulator. In order to improve HMM techniques, Toderici and Stamp [73] combined HMMs with the statistical framework of the Chi-squared test to build a new method for metamorphic detection. The hybrid detection model uses probabilistic scores from both the HMM and the Chi-squared distance (CSD) detectors and optimizes the weight for these detectors by performing a grid search. In the CSD detector, they modeled the behavior of the compiler that produces benign programs by analyzing the instruction frequencies. Then, the Pearson χ^2 distance is computed between the frequencies of opcodes used by the complier and the frequencies of opcodes in the target file to determine whether it is infected. The results showed that the proposed hybrid model was able to beat the scores of both the HMM and CSD detectors.

2.3.2 Metrics and Measures

The most common method to determine similarity between programs is to extract signals from the target programs and then compare these signals. These signals take different forms such as string sequences, feature vectors, sets of features, trees and graphs representing the structure of programs. Based on the used form, we can compare two instances using a suitable similarity metrics or measures.

String Similarity

String sequences can be compared using similarity metrics. Since sequences are common feature structures in many areas, such as in computing similarity between documents, the literature is rife with proposals for string similarity measures. In what follow we present three of the most commonly used similarity metrics and measures.

- **Levenshtein (Edit) Distance** The Levenshtein distance between two strings defines the number of edit operations that must be performed to transform one string to the other. An edit operation includes character insertion, deletion, and substitution. This metric is suitable for comparing two strings with unequal length. Given two strings, p and q, the Levenshtein distance is given by $distance_{p,q}(|p|, |q|)$ [81] where

$$distance_{p,q}(i,j) = \begin{cases} max(i,j) & \text{if min(i,j)=0,} \\ min \begin{cases} distance_{p,q}(i-1,j)+1 \\ distance_{p,q}(i,j-1)+1 \\ distance_{p,q}(i-1,j-1)+1_{(q_i \neq q_j)} \end{cases} & \text{otherwise} \end{cases}$$

$1_{(q_i \neq q_j)}$ is the indicator function which is equal to 0 when $p_i = p_j$ and equal to 1 otherwise. The similarity (in $[0, 1]$) between p and q using Levenshtein distance is defined as [22]:

$$\textit{similarity}(p, q) = 1 - \frac{distance_{p,q}}{max(|p|,|q|)}$$

- **Longest Common Subsequence (LCS)** The LCS [82] is used to find the longest subsequence common to all sequences in given two strings $P = \{p_1, p_2, p_3 ... p_m\}$ and $Q = \{q_1, q_2, q_3 ... q_n\}$. The prefixes of P are $P_{1,2,...m}$; the prefixes of Q are $Q_{1,2...n}$. Let $LCS(P_i, Q_j)$ represent the set of longest common subsequence of prefixes P_i and Q_j. This set of sequence is given by

$$LCS(P_i, Q_j) = \begin{cases} \emptyset & \text{if i = 0 or j = 0} \\ LCS(P_{i-1}, Q_{j-1}) + 1 & \text{if } p_i = q_j \\ longest(LCS(P_i, Q_{j-1}), LCS(P_{i-1}, Q_j)) & \text{if } p_i \neq q_j \end{cases}$$

The similarity between P and Q using LCS is defined as [26]

$$similarity(P, Q) = \frac{|LCS(P,Q)|}{max(|P|,|Q|)}$$

- **Normalized Compression Distance (NCD)** The NCD is another method for measuring the similarity between two strings. This metric approximates *normalized information distance* which is an information-theoretic measure for quantifying the length of the shortest program that computes string p from string q. It is defined as [21]:

  $$NCD(p, q) = \frac{C(p,q)-min(|C(p)|,|C(q)|)}{max(|C(p)|,|C(q)|)}$$

  where C is a compressor using any real world compression algorithm such as the Lempel-Ziv-Welch [80] or the Huffman algorithm [76].

Vector Similarity

The most widely used method for detecting similarity between two objects is to compare their corresponding feature vectors. The objects are mapped onto feature vectors in an appropriate multidimensional feature space. Then the similarity between the two objects is defined as the proximity of their feature vectors in the feature space.

- **Minkowski Distance** The Minkowski distance between two n-dimensional vectors, A and B is given by

  $$distance(A, B) = \left(\sum_{i=1}^{n} \mid A_i - B_i \mid^{\lambda} \right)^{\frac{1}{\lambda}}$$

  The Minkowski Distance can be considered as a generalization of both the Euclidean distance and the Manhattan distance. When $\lambda = 1$, it corresponds to the Manhattan distance and when $\lambda = 2$, it corresponds to the Euclidean distance.

- **Cosine Similarity** Cosine similarity is a measure of similarity between two vectors based on the cosine of the angle between them. The vectors *A* and *B* are usually the term frequency vectors. The Cosine similarity between vectors *A* and *B* is given by

$$similarity(A, B) = \frac{A \cdot B}{||A||\ ||B||} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2}\sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

  This similarity score ranges between -1 to 1.

Set Similarity

The features extracted from documents or programs can also be treated as sets, such as the sets of *n*-grams. Two sets can be compared using the following set measures:

- **Jaccard Index** The Jaccard index [78] is often used for comparing similarity between two data sets. Given two sets *A* and *B*, the Jaccard index is the result of division between the number of features that are common to all, and the number of properties as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

- **Containment** Broder [15] defines the *containment* for comparing two documents. The function $f()$ computes sets of features from two document *p* and *q*, such as fingerprints of "shingles" [86]. The *containment*(*p*, *q*) of *p* within *q* is defined as:

$$containment(p, q) = \frac{|f(p) \cap f(q)|}{|f(p)|}$$

Graph Similarity

Function call graphs are common features used for comparing two programs. The edit distance also works on graphs, where the distance between two graphs is defined as the minimum number of basic edit operations necessary to transfer one graph to another.

Given two graphs G_1 and G_2, G is *common subgraph* if there exists subgraph isomorphisms from G to G_1 and from G to G_2. If there exists no other common subgraph *G'* of G_1 and G_2 that has more nodes than *G*, *G* is the *maximal common subgraph*(*mcs*) [16]. The similarity between G_1 and G_2 is defined as:

$$\textit{similarity}(G_1, G_2) = \frac{|mcs(G_1,G_2)|}{max(|G_1|,|G_2|)}$$

where |*G*| is the number of nodes in *G*. The *containment*(G_1, G_2) of G_1 within G_2 is defined as:

$$\textit{containment}(G_1, G_2) = \frac{|mcs(G_1,G_2)|}{|G_1|}$$

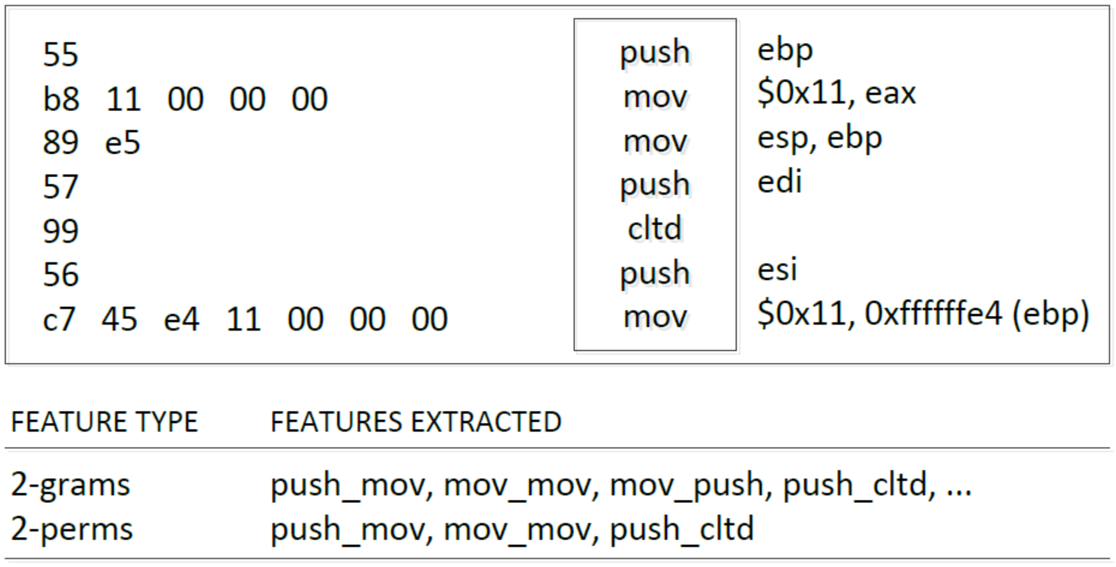
2.3.3 Similarity Detection Algorithms

Algorithms for identifying similarity between malware can be applied directly on the source or binary code. More common, however, is to first convert the malware code to a more convenient representation or characterize the code with features such as string sequences or graphs and then compare them. Four primary algorithms for malware similarity analysis are discussed in the following sections.

n-gram-Based Analysis

Comparing sets of *n*-grams of two document is a popular technique for detecting their similarity. This basic method has been used for plagiarism detection of text documents and source code, for clone detection of programming statements [69], and for birthmark detection of executable code.

Wong and Stamp [88] presented a simple detection method based on a similarity index and a detector based on hidden Markov models for metamorphic viruses. First, they employed the method in [47] which extracts the sequences of opcodes from two metamorphic variants, compares two sequences by matching all subsequences of trigram opcodes regardless of order, and computes similarity scores by determining the fraction of opcodes that are covered by line segments in matching graph. Then they trained hidden Markov models by using the assembly opcode sequence of viruses of metamorphic families to detect virus variants from same family. The results suggested that both of these methods detect all testing metamorphic viruses accurately.



```
55                          push   ebp
b8  11  00  00  00          mov    $0x11, eax
89  e5                      mov    esp, ebp
57                          push   edi
99                          cltd
56                          push   esi
c7  45  e4  11  00  00  00  mov    $0x11, 0xffffffe4 (ebp)
```

FEATURE TYPE	FEATURES EXTRACTED
2-grams	push_mov, mov_mov, mov_push, push_cltd, ...
2-perms	push_mov, mov_mov, push_cltd

Figure 15: The *n*-gram features extracted from malware [74]

Walenstein et al. [74] described a method for searching database of malware for a match. For searching for previous malware that match a new variant, they applied a feature comparison approach where *n*-gram or *n*-perm features are extracted from mnemonic sequences of assembly instructions as shown in Figure 15. *n*-perms are exactly like *n*-grams except that the ordering of the characters is not considered during the matching. Similarity score between *n*-gram or *n*-perm feature vectors of malware are calculated using Cosine similarity with inverse document frequency.

API-Based Analysis

Programs interact with the system on which they run through a set of standard library types and calls. There are several birthmarking algorithms that use standard library functions as signatures. The main idea is that the way a program uses the standard libraries or system calls is not only unique to that program but also difficult for the adversary to forge. The work in [67] is an example for using API sequences to measure similarity between malware.

Han et al. [29] proposed a detection method for malware variants by measuring similarity between control flow graphs related to API calls. The proposed method first extracts API calls related graphs from malware samples and stores it in a database. Then, the API call related graphs corresponding to the suspicious files are extracted. In order to compare with the existing malware graphs, they used Jaccard Index [78] to measure the similarity. This method was evaluated on 200 malware samples from different families and the obtained results verified that API call related control flow graphs are different even if corresponding malware variants have identical API calls.

Shankarapani et al. [67] presented two general malware detection methods: Static Analyzer for Vicious Executables (SAVE) which uses API call sequence for analysis, and Malware Examiner using Disassembled Code (MEDiC) that uses assembly code for the analysis. In MEDiC, the authors created a signature for each sequence of instruction based on a procedure that matches malware with a threshold. In SAVE, they first mapped the extracted API sequence to a string formed by 32-bit integers. In order to compare the API sequences of malware, sequence alignment is applied on API sequence strings, and then three similarity algorithms, including Cosine similarity, extended Jaccard coefficient, and Pearson correlation [84], are used to measure similarity between sequences. SAVE calculates similarity score using the mean value of the three measures and makes a decision whether the test file is malware.

Graph-Based Analysis

A program can be modeled by a graph-like structure. Functions can be represented as control flow graphs (CFG), dependencies between statements within a function as dependence graph, and calls between functions as call graphs. The similarity between programs can then be computed over their corresponding graph representations.

Xu et al. [90] proposed a similarity computing method between malware variants using function-call graphs and their opcodes. The function matching process uses function opcode information and function-call graph to locate all of the common function pairs between two function-call graphs. This process consists of four steps: matching external functions that have identical name, matching the local functions that call the same external functions, matching the local functions based on their opcodes, and matching the local functions according to their matched neighbors. The authors also used the maximum common vertices and edges to compute the similarity between two function-call graphs of malware with all matched vertices obtained in function matching. The limitation of this method is that the incomplete function-call graph constructed from encrypted malware makes the similarity score incorrect.

Runwal et al. [65] presented a method for measuring similarity of executables based on opcode graphs. This technique was applied to detect metamorphic malware and the obtained results show that it outperforms the detection method based on hidden Markov models in [88]. For construction of opcode graphs, this technique extracts the opcode sequence in which each distinct opcode is a node in a directed graph. The directed edge is then inserted from a node/opcode to each possible successor node/opcode with a weight that represents the probability of the successor node. Based on constructed opcode graph, Runwal et al. mapped the weighted directed graphs to the edge-weight matrices and computed similarity score between executable files using the following formula:

$$\text{score(A, B)} = \frac{1}{N^2}\left(\sum_{i,j=0}^{N-1} \mid a_{ij} - b_{ij} \mid\right)$$

where A = $\{a_{ij}\}$ and B = $\{b_{ij}\}$ are the edge-weight matrices corresponding to the executable files.

Tree-Based Analysis

A program in source form is hierarchical, i.e., tree-structured. Nested statements in structured programming languages form trees, classes without multiple inheritance form a tree in languages, and operators and operands in an expression form a tree. An abstract syntax tree (AST) is a preferable program representation when transforming a program code into a form that is close to source. The AST abstracts away from any parsing process and only keeps the information that is in the original program. *Revolver* [34], proposed by

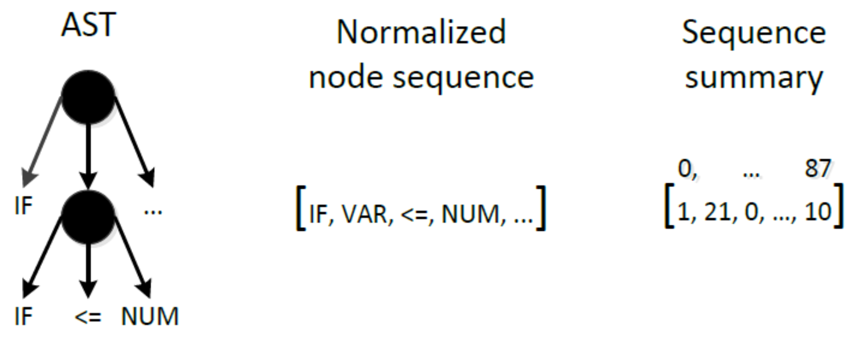


Figure 16: Data structures used by *Revolver* [34]

Kapravelos et al., is a tool to identify similarities between malicious JavaScript programs and to interpret their difference in order to detect evasions. *Revolver* extracts ASTs of the JavaScript code contained in benign and malicious web pages to generate AST representations that are later transformed into a normalized node sequence. As shown in Figure 16, the normalized node sequence is the sequence of node types obtained by performing a pre-order visit of the tree, which is used for sequence summary by storing the number of times each node type appears in the corresponding AST. The similarity measurement used by *Revolver* is based on the pattern matching algorithm proposed by Ratcliff et al. [60] which can find the longest contiguous common subsequence(LCCS) between two node sequences.

Curtdinger et al. [24] presented Zozzle which is a detector for JavaScript malware using classification of hierarchical features of JavaScript abstract syntax tree. Zozzle used naïve Bayesian classifier that is trained with extracted features including the context in which it appears and the text of the AST node. To limit the number of features and improve the performance of the classifier, Zozzle extracts features only from the nodes of expressions and variable declarations in the AST, and uses the χ^2 statistic to perform feature selection. Curtdinger et al. evaluated Zozzle using 1.2 million pre-categorized code samples and their results suggest that it is fast and accurate tool for detecting JavaScript malware.

Chapter 3

Automated Malware Similarity Analysis

In this chapter we present a framework for automated malware similarity analysis. As shown in Figure 17, the proposed framework includes four main components: (i) a pre-processing module, (ii) a function-level similarity detection module, (iii) a similarity scoring module for the whole maleware sample, and (iv) a visualization module. First, the pre-processing module disassembles the binary samples into assembly files and generates functions with imported APIs. The similarity detection is performed on abstract function regions to determine the similarity between functions. Then, similarity scores between samples are calculated based on the proposed similarity metric. Finally, the interactive visualizer is used to illustrate the relationship between cloned functions of analyzed samples.

3.1 Pre-Processing Module

The pre-processing step consists of disassembling the binary samples, normalizing function regions, and recognizing respective imported APIs.

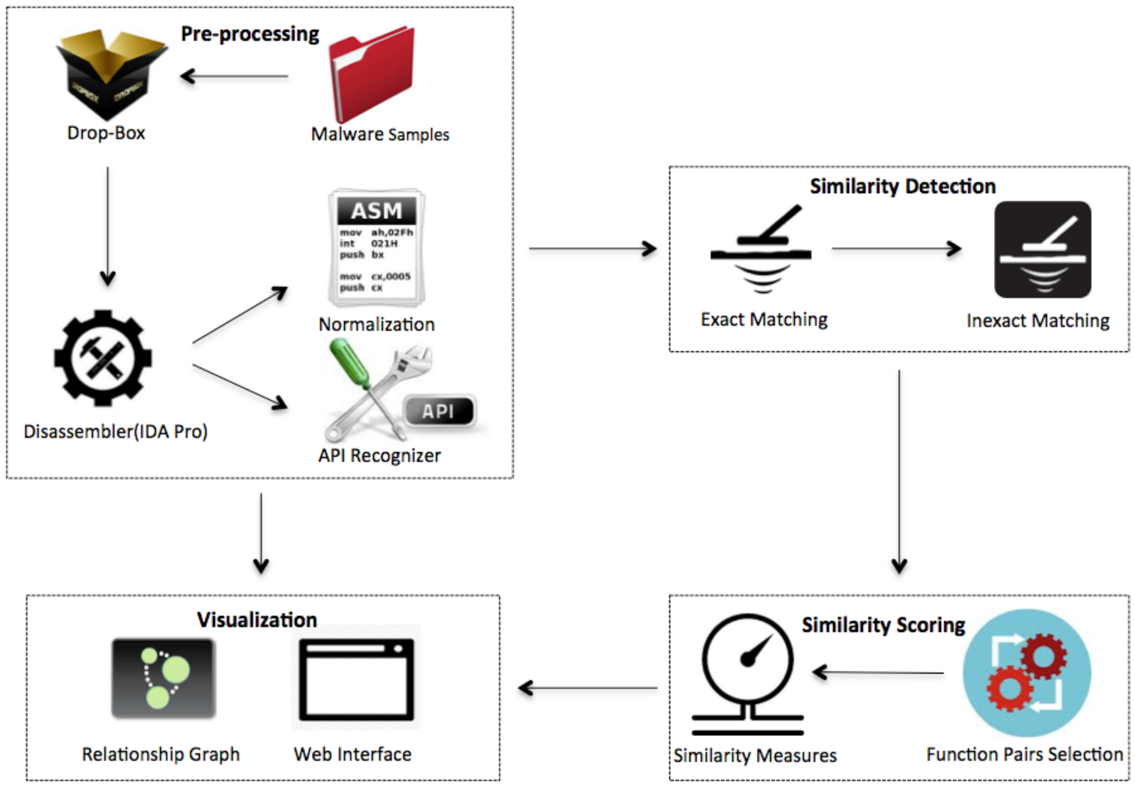


Figure 17: Overview of the proposed framework

3.1.1 Disassembly

We use an interactive disassembler, namely IDA Pro [2], to handle the disassembly process of all input malware samples with the aid of a plug-in that we developed to facilitate the automation of the rest of the process. Since IDA Pro does not involve unpacking functionality, we expect that every malware sample is unpacked before submission to our framework. An assembly language instruction usually consists of a *mnemonic* followed by a sequence of *operands*. The mnemonic represents the specific operation performed by the instruction. The operands can be partitioned into three categories: memory references (*e.g.,* "[edi+4Ch]"), register reference (*e.g.,* "ebx"), and constant values (*e.g.,* "20h"). The developed plug-in extracts the instructions of all functions in each analyzed sample. The normalization process is applied to *function regions* which are sequences of instructions inside the disassembled functions.

3.1.2 Normalization

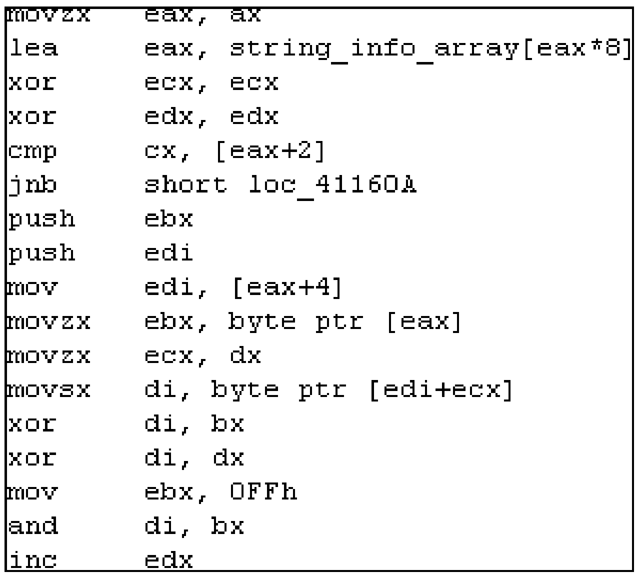
The work in [61] does not involve a normalization step prior to applying fuzzy hashing [39] on function regions. The lack of this process makes it somewhat difficult to recognize the similarity between two function regions that might be identical except for some of the used operands. Overcoming this problem is particularly important since, as described in chapter 2, malware writers can apply obfuscation on assembly instructions to avoid detection by simple signature-based anti-malware. For example, register in the operands of an assembly language instruction can be substituted with equivalent ones. In order to account for this situation, we normalize the instructions before applying similarity detection on function regions.

Reg32
eax ebx ecx edx esi edi esp ebp eip eflags
Reg16
ax bx cx dx si di sp bp ip flags cs ds ss es fs gs
Reg8
ah al bh bl ch cl dh dl
Mem
[0x805b634] [ebx] [bp+598h] [esp+24h+var_24] etc.
Value
0 3 455h 1F0001h etc.
Dummy
word_4022EF dword_44475B sub_444808 loc_4456FE locret_40109E unk_402564 etc.

Table 7: An example for operands normalization

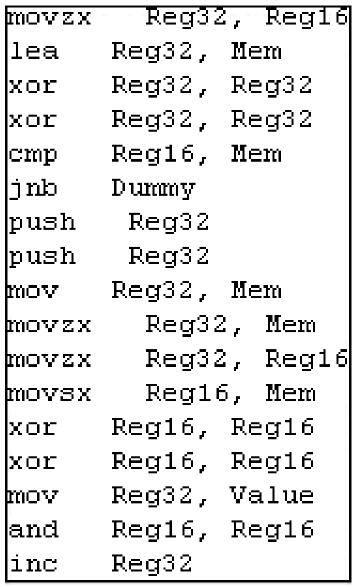
One basic method for code obfuscation is to insert random number of "nop" instructions in the program's assembly code. For this reason, the first step in the normalization process is to discard the "nop" instructions. Then, we normalize the operands of the other mnemonics on the basis of three categories including memory, constant value, and register.

For memory references, we normalize them to *Mem* which abstracts specific memory references information. For constant values, *Value* is used to substitute for them. For registers, the abstract replacement depends on the number of bits they can hold. Accordingly, *Reg*8, *Reg*16, and *Reg*32 are used to substitute for the registers. In addition to general operands, there are dummy names that are used to denote subroutines, program locations and data which are automatically generated by IDA Pro [2]. We generalize these names to *Dummy*. Table 7 illustrates some examples for our operand normalization process. Figure 18 shows an example for a function region before and after the normalization step.



```
movzx   eax, ax
lea     eax, string_info_array[eax*8]
xor     ecx, ecx
xor     edx, edx
cmp     cx, [eax+2]
jnb     short loc_41160A
push    ebx
push    edi
mov     edi, [eax+4]
movzx   ebx, byte ptr [eax]
movzx   ecx, dx
movsx   di, byte ptr [edi+ecx]
xor     di, bx
xor     di, dx
mov     ebx, 0FFh
and     di, bx
inc     edx
```

(a) Before Normalization



```
movzx   Reg32, Reg16
lea   Reg32, Mem
xor   Reg32, Reg32
xor   Reg32, Reg32
cmp   Reg16, Mem
jnb   Dummy
push   Reg32
push   Reg32
mov   Reg32, Mem
movzx   Reg32, Mem
movzx   Reg32, Reg16
movsx   Reg16, Mem
xor   Reg16, Reg16
xor   Reg16, Reg16
mov   Reg32, Value
and   Reg16, Reg16
inc   Reg32
```

(b) After Normalization

Figure 18: An example for a normalized function produced by our system

3.1.3 API Class Recognizer

Windows APIs are crucial to any program, including malware, which runs on Microsoft Windows systems. Without APIs, hackers need to write tremendous lines of codes in malware. Examination of imported APIs is a basic component in malware analysis which provides significant clues about malicious activities performed by the malware. The analysts can gain useful information about the functionality of malware by studying the list of imported APIs. For example, *GetTickCount* is a very common API used for detecting debuggers. *LookupPrivilegeValueA* and *AdjustTokenPriviledges* are generally used for accessing the Windows security tokens. *RegSetVauleExA*, *RegCreateKeyA* and *RegCloseKey* are used to access and modify registry keys.

API Classes	**API Examples**
Cryptography	EncryptFileA, CryptGenKey
Hashing	CryptCreateHash, CryptSignHashA
File	DeleteFileA, fwrite
OSInformation	GetUserNameA, GetComputerNameA
String	lstrcmpA, _stricmp
Registry	RegDeleteKeyA, RegOpenKeyA
Winsock	GetAddrInfoW, gethostname
WinINet	InternetCreateUrlA, InternetOpenW
Directory	GetSystemDirectoryA, RemoveDirectoryA
Memory	LocalAlloc, memcpy
Bitmap	BitBlt, LoadBitmapA
Thread	CreateThread, ExitThread

Table 8: Examples of API classes

The API class recognizer is designed to determine probable malicious operations by functions in malware. This process provides extra information that indicates similarity between behaviors of functions in order to augment the other syntactic similarity measures. Based on previous experimental result [54, 55] on top maliciously used API calls and basic API classification [30], we manually clustered 2231 APIs that are frequently used by malware into 64 groups according to their functionality and malicious usage. Table 8 provides an example for our API classification. Based on basic imported APIs that are identified by IDA Pro, we implement an API class recognizer as a plug-in script for IDA Pro in which generic functions are automatically renamed with the corresponding API class names. The disassembled function regions are labeled with the class names of their corresponding imported APIs. The function format after normalization and API labeling is shown in Table 9 where the malware sample *A* is dissected into five function regions donated by F$_{A-1}$ to F$_{A-5}$. After pre-processing and normalization, each *abstract function region* contains recognized API classes and normalized instructions.

Abstract Function Regions of Malware Sample *A*
F$_{A-1}$(None){normalized instructions}
F$_{A-2}$(API$_{class2}$,API$_{class3}$){normalized instructions}
F$_{A-3}$(API$_{class1}$){normalized instructions}
F$_{A-4}$(None){normalized instructions}
F$_{A-5}$(API$_{class1}$,API$_{class2}$,API$_{class4}$){normalized instructions}

Table 9: The format of abstract function regions

3.2 Similarity Detection at the Function Level

We have two approaches to find similar functions among malware samples: exact matching of abstract function regions, and inexact matching of feature vectors that represent structural characteristics of the abstract function regions.

3.2.1 Exact Matching

The exact matching method, adapted from [13], identifies the exact cloned function pairs among the abstract functions by comparing the normalized assembly language instructions. Two functions are considered as exact match if all normalized instructions in the two function regions are identical and follow the same sequence. Functions that have identical hash values are efficiently detected by this exact matching module.

Algorithm 1 illustrates the process of exact matching. At first, the algorithm initializes an empty hash table H for each malware. Each entry in the hash table contains a hash value

v with a corresponding unique identifier of the function. In Lines 3-8, the algorithm iterates through each abstract function region f, generates a hash value v and adds v to H for the corresponding abstract function region. In Lines 9-12, the algorithm iteratively compares the hash values in each hash table and builds an array of pairs of exact matching functions, denoted by EF.

Algorithm 1 Exact Matching on Abstract Fucntion Regions (EMAFR)

```
Input: Set of abstract function regions in malware1 $F_1$
       Set of abstract function regions in malware2 $F_2$
Output: Set of exact abstract function region pairs $EF$
1: $EF \leftarrow \emptyset$;
2: $H_1 \leftarrow \emptyset$;
3: $H_2 \leftarrow \emptyset$;
4: foreach function $f \in F_1$ do
5:   $v \leftarrow hash(f)$;          // hash() computes hash value $v$ for function $f$
6:   $H_1(f) \leftarrow H_1(f) \cup v$;
7: foreach function $f \in F_2$ do
8:   $v \leftarrow hash(f)$;
9:   $H_2(f) \leftarrow H_2(f) \cup v$;
10: for $i = 0 \leftarrow | H_1(f) |$ do
11:     for $j = 0 \leftarrow | H_2(f) |$ do
12:        if $v_i == v_j$
13:           $EF \leftarrow EF \cup (f_i, f_j)$;
14: return $EF$;
```

3.2.2 Inexact Matching

Hackers may modify codes in malware functions to avoid signature-based detection or to add new malware functionalities. Inexact match aims to find cloned function pairs by computing a similarity score between feature vectors that represent their structural properties. The similarity score between the pairs of function regions is then used to derive similarity between malware samples.

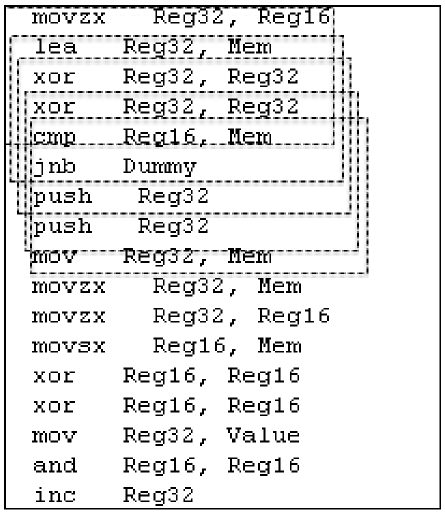
The first step of inexact matching is to collect features and build a feature vector for each abstract function region. The features used to characterize abstract function regions can be classified into five categories. The first category includes all distinct mnemonics of instructions. The second category includes types of operands. The combination of the mnemonic and the type of the first operand is the third category. The fourth category includes the combinations of the type of the first and second operands. The last category includes the combinations of the mnemonics and the type of the first and second operands. Algorithm 2 shows how to collect possible features from abstract function regions.

Algorithm 2 Feature Collection for an Abstract Function Region (FCAFR)

```
Input: Abstract function region $F$
Output: Set of features $S$
1: $S \leftarrow \emptyset$;                                  // initialize unique set of feature
2: foreach instruction ins in function $F$ do
3:    $S \leftarrow S \cup mnemonic(ins)$;                    // mnemonic() extracts mnemonics
4:    foreach operand $o \in operands(ins)$ do
5:       $S \leftarrow S \cup type(o)$;                       // type() get type of operand
6:    $ops \leftarrow operands(ins)$;                         // operands() extracts operands
7:    if $length(ops) \geq 1$ then
8:       $S \leftarrow S \cup (mnemonic(ins), type(ops_0))$;
9:    if $length(ops) \geq 2$ then
10:      $S \leftarrow S \cup (type(ops_0), type(ops_1))$;
11:      $S \leftarrow S \cup (mnemonic(ins), type(ops_0), type(ops_1))$;
12: return $S$;
```

As depicted in Figure 19, the normalized function is scanned by sliding a fixed length window. For each region contained inside the sliding window, we count the number of occurrences of each feature and construct a feature vector for the abstract function region as shown in Algorithm 3.

Throughout our experiments, we set the default size of the window to 5, which is smaller than the size of most analyzed functions in our malware dataset. Functions with less than 5 lines are ignored when calculating the overall similarity between large malware samples. The reasons for skipping these short functions is that the similarity scores



```
movzx   Reg32, Reg16
lea     Reg32, Mem
xor     Reg32, Reg32
xor     Reg32, Reg32
cmp     Reg16, Mem
jnb     Dummy
push    Reg32
push    Reg32
mov     Reg32, Mem
movzx   Reg32, Mem
movzx   Reg32, Reg16
movsx   Reg16, Mem
xor     Reg16, Reg16
xor     Reg16, Reg16
mov     Reg32, Value
and     Reg16, Reg16
inc     Reg32
```

Figure 19: A sliding window applied to a function

between the functions with very few number of lines are relatively high because the generated feature vectors from these functions tend to be almost the same, which can affect overall scores between samples.

Algorithm 3 Construct Feature Vector (CFV)

```
Input: Abstract function region $F$
       Window size $w$ (default 5)
       Stride $s$ (default 1)
Output: Feature vector $V$
1: $S \leftarrow$ FCAFR($F$);                    // see Algorithm 2
2: $V \leftarrow \{0\}$;                          // initialize $V$ with size $\|S\|$
3: if length($F$) $\geq w$ then
4:    for $k$ = 0 to length($F$) - $w$ do
5:       currentRegion $\leftarrow F_{[k,k+w)}$;
6:       foreach feature $f$ in currentRegion
7:          $V \leftarrow V$ + Count($f$);          //Count() computes feature frequence
8:       $k \leftarrow k + s$;
9: return $V$;
```

After generating feature vectors for abstract function regions, the inexact matching module calculates the Bray-Curtis dissimilarity [75], d^{BCD}, which is a modified Manhattan measure [83]. Based on our experimental results, this similarity function can better measure the actual relationship between functions. The similarity score between two functions is then given by 1-d^{BCD}, where the outcome is neatly bounded in [0,1]. More precisely, given two vectors, **X** = (x_1, x_2, ..., x_n) and **Y** = (y_1, y_2, ..., y_n) constructed from two abstract function regions, the Bray-Curtis dissimilarity and similarity score between them can be calculated as follows:

$$d^{BCD}(\mathbf{X},\mathbf{Y}) = \frac{\sum\limits_{k=0}^{n-1} \mid x_k - y_k \mid}{\sum\limits_{k=0}^{n-1} (x_k - y_k)} \tag{1}$$

$$\mathit{SimilarityScore}(\mathbf{X},\mathbf{Y}) = 1 - d^{BCD} \tag{2}$$

3.3 Similarity Scoring between Malware Samples

In this step, we aggregate similarity scores calculated on the function level (for functions contained in two malware samples) to evaluate the similarity between the two samples. This process starts by pairing functions from the two samples using the results obtained from the function similarity matching explained in the previous sections. Then we calculate an overall similarity score between the two malware samples.

3.3.1 Pairing of Similar Functions

Given two malware samples, *A* and *B*, with *N* and *M* functions, respectively, the number of compared function pairs is $N \times M$. Thus, $N \times M$ similarity scores can be generated from the function similarity detection module as explained in the previous section. High similarity scores between functions (e.g., above 0.5) may indicate that these functions share similar pieces of codes but does not always justify that the two functions should be paired together when comparing the similarity between malware samples. For example, in some situations, one function inside sample *A* may have a large similarity score to several functions inside malware sample *B*. Using simple measures such as taking the average of all function level pair-wise scores as a measure of the similarity between the two samples does not lead to good results because such scoring can make similarity between samples high even if they are not similar. In order to reduce the influence of these similarity scores on concluding similarity between samples, we first filter similarity scores between functions by selecting the appropriate function to be paired together. If two samples are malware variants, they share similar or identical functions and therefore these functions can be detected as appropriate function pairs in which two functions have mutually maximum similarity score among all function comparisons. In other words, similar functions belonging to two malware variants are detected in our selection process by identifying the best matching function pairs as illustrated by the following example.

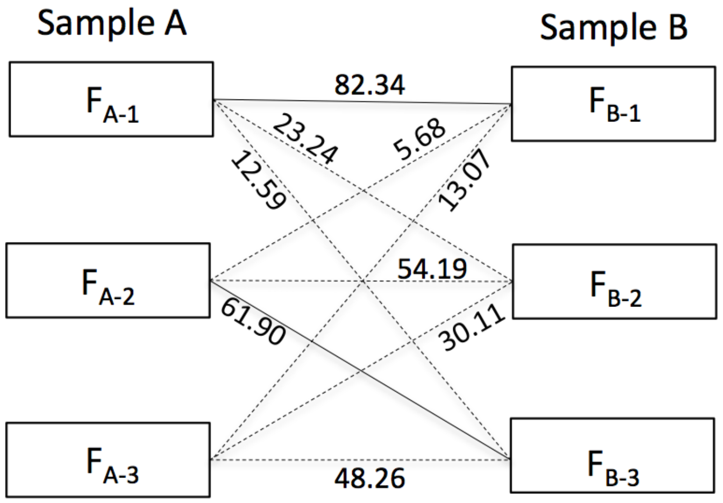
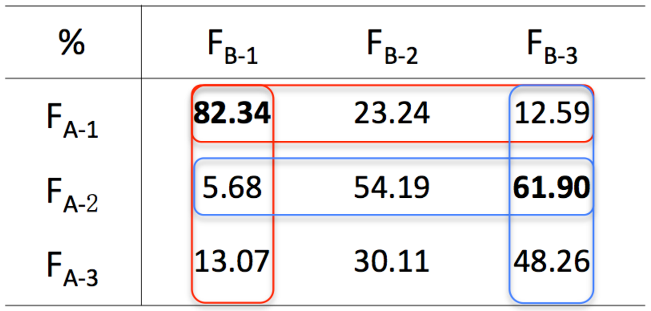


Figure 20: An example of functions comparisons

Example 1 Figure 20 shows a comparisons between three functions of sample *A* and three functions of sample *B*. The similarity scores between these functions are calculated using the inexact matching algorithm and given in Table 10. In Figure 20, the 3×3 function comparison pairs are indicated by the dotted lines with the corresponding similarity scores.



%	$F_{B\text{-}1}$	$F_{B\text{-}2}$	$F_{B\text{-}3}$
$F_{A\text{-}1}$	**82.34**	23.24	12.59
$F_{A\text{-}2}$	5.68	54.19	**61.90**
$F_{A\text{-}3}$	13.07	30.11	48.26

Table 10: An example of similarity score matrix

Among the three comparisons between F_{A-1} and the other three functions of sample *B*, the maximum score is 82.3% and corresponds to F_{B-1}. Among the three function comparisons between F_{B-1} and the other three functions of sample *A*, the maximum score is also 82.3%, and corresponds to F_{A-1}. Thus, F_{A-1} and F_{B-1} are selected to be paired together when calculating the overall similarity between *A* and *B* as indicated by the solid line in the figure. Following same process, the pair (F_{A-2}, F_{B-3}) is selected. As shown in Table 10, we can transform the 3×3 similarity scores to a matrix in which the similarity scores of the selected function pairs are marked at the intersection of red rectangles and the intersection of blue rectangles. The score for the selected pair in the intersection area of rectangles of the same color is higher than any score in the corresponding row and column. If the above function pairing approach leads to a situation where one function is included into more than one function-pairs, our algorithm proceeds by choosing only one pair from these function pairs at random.

3.3.2 Similarity Measures between Malware Samples

The objective of this process is to determine similarity between samples using scores calculated by inexact matching on the function level. We start by calculating some measures that indicate that containment relationship between the analyzed sample pairs.

Containment Score This score reflects the reuse of the code of one sample into the other. For example, given two malware samples *A* and *B*, the containment score for *A*, $\textit{ContainmentScore}_A$, is used to reflect how much code of *A* is used in B. With similarity scores of selected function pairs, we can use the following formula to calculate *ContainmentScore* for *A* and *B* with *n* selected function pairs:

$$\textit{ContainmentScore}_A = \sum_{k=1}^{n} S_k \cdot W_{A_k} \tag{3}$$

$$\textit{ContainmentScore}_B = \sum_{k=1}^{n} S_k \cdot W_{B_k} \tag{4}$$

where S_k represents the similarity sore of the k^{th} selected function pair. W_{A_k} and W_{B_k} donate the weight values for the function of *A* and *B* in the k^{th} selected pair respectively. Here, the weight value is proportional to the size of the corresponding function divided by the total size of functions of the sample. In this way, longers functions contribute more to the containment score.

Similarity Score

Based on the containment score calculated for each sample, the last step of similarity scoring is to derive a single numerical value which determines the overall similarity between samples. Since the containment score is influenced by the size of the sample, taking the average between the containment scores of two samples with very different number of functions does not accurately reflect their similarity. Instead, we use weighted average of the containment scores using the following formula:

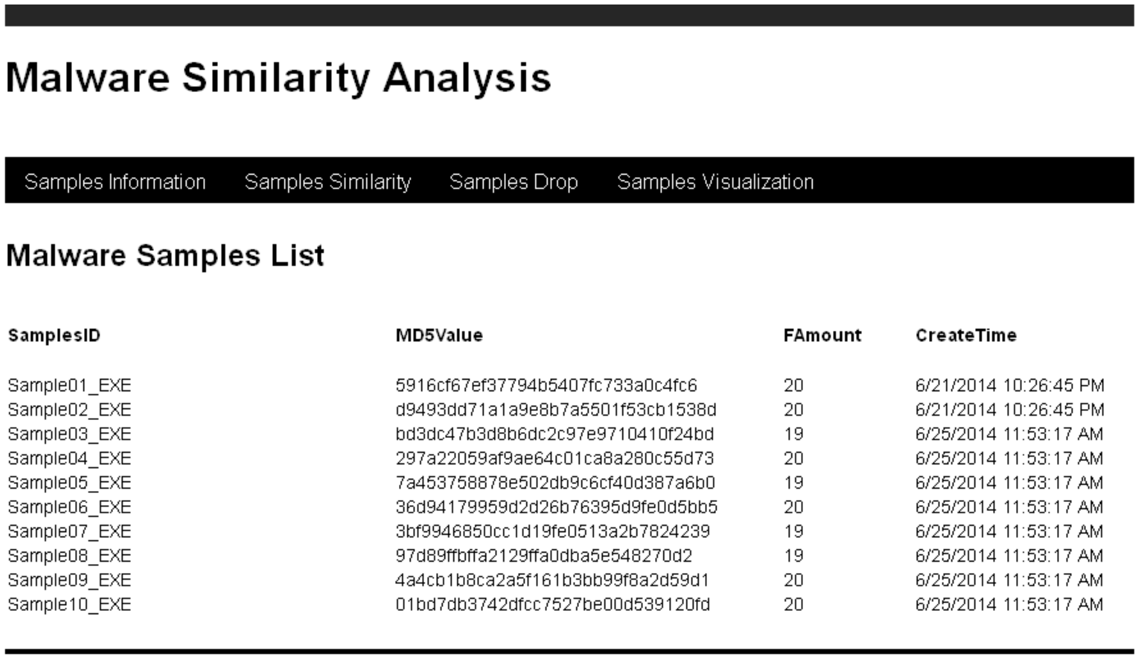
$$\textit{SimilarityScore}_{AB} = \frac{\textit{ContainmentScore}_A \cdot N_B + \textit{ContainmentScore}_B \cdot N_B}{N_A + N_B} \tag{5}$$

where N_A and N_B represent the number of functions of sample *A* and sample *B*, respectively.

3.4 Visualization Module

To facilitate the use of the developed framework, we implemented a web interface that can be used to show all the data generated by the three analysis modules and visualize the relationship between analyzed samples. Malware samples can be uploaded via a Drop-Box in the web interface to perform similarity analysis. The similarity results are stored in a database that allows the analysts to query for a specific sample or function and retrieve a list of similarity scores. Finally, using the information stored in the graph database, we visualize relationships between similar functions of the samples in the system.

3.4.1 Querying Similarity Results

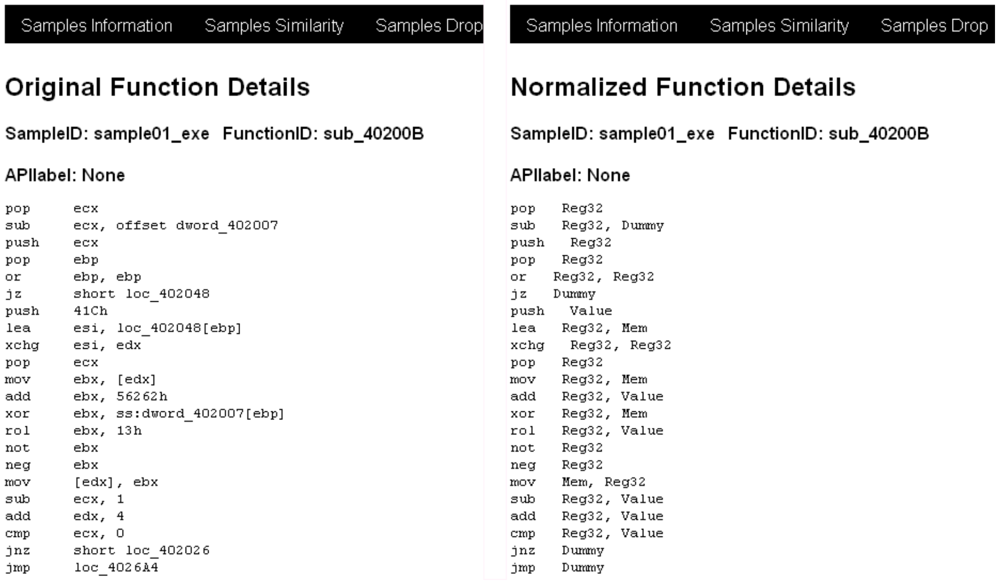


Malware Samples List

SamplesID	MD5Value	FAmount	CreateTime
Sample01_EXE	5916cf67ef37794b5407fc733a0c4fc6	20	6/21/2014 10:26:45 PM
Sample02_EXE	d9493dd71a1a9e8b7a5501f53cb1538d	20	6/21/2014 10:26:45 PM
Sample03_EXE	bd3dc47b3d8b6dc2c97e9710410f24bd	19	6/25/2014 11:53:17 AM
Sample04_EXE	297a22059af9ae64c01ca8a280c55d73	20	6/25/2014 11:53:17 AM
Sample05_EXE	7a453758878e502db9c6cf40d387a6b0	19	6/25/2014 11:53:17 AM
Sample06_EXE	36d94179959d2d26b76395d9fe0d5bb5	20	6/25/2014 11:53:17 AM
Sample07_EXE	3bf9946850cc1d19fe0513a2b7824239	19	6/25/2014 11:53:17 AM
Sample08_EXE	97d89ffbffa2129ffa0dba5e548270d2	19	6/25/2014 11:53:17 AM
Sample09_EXE	4a4cb1b8ca2a5f161b3bb99f8a2d59d1	20	6/25/2014 11:53:17 AM
Sample10_EXE	01bd7db3742dfcc7527be00d539120fd	20	6/25/2014 11:53:17 AM

Figure 21: Main page of the system GUI

As depicted in the example shown in Figure 21, after pre-processing of submitted samples, the basic information including names of samples, MD5 hash values for samples, number of functions, and the creation times of the samples will be shown in the page of samples information. All the information regarding functions of each sample can be examined by clicking the name of the sample on the web page. The page of malware sample detail provides number of line for each function and the API class name identified by the API Class Recognizer. The analysts can also access the original and normalized instructions of the corresponding functions as shown in Figure 22.



Original Function Details

SampleID: sample01_exe FunctionID: sub_40200B

APIlabel: None

```
pop      ecx
sub      ecx, offset dword_402007
push     ecx
pop      ebp
or       ebp, ebp
jz       short loc_402048
push     41Ch
lea      esi, loc_402048[ebp]
xchg     esi, edx
pop      ecx
mov      ebx, [edx]
add      ebx, 56262h
xor      ebx, ss:dword_402007[ebp]
rol      ebx, 13h
not      ebx
neg      ebx
mov      [edx], ebx
sub      ecx, 1
add      edx, 4
cmp      ecx, 0
jnz      short loc_402026
jmp      loc_4026A4
```

Normalized Function Details

SampleID: sample01_exe FunctionID: sub_40200B

APIlabel: None

```
pop   Reg32
sub   Reg32, Dummy
push   Reg32
pop   Reg32
or   Reg32, Reg32
jz   Dummy
push   Value
lea   Reg32, Mem
xchg   Reg32, Reg32
pop   Reg32
mov   Reg32, Mem
add   Reg32, Value
xor   Reg32, Mem
rol   Reg32, Value
not   Reg32
neg   Reg32
mov   Mem, Reg32
sub   Reg32, Value
add   Reg32, Value
cmp   Reg32, Value
jnz   Dummy
jmp   Dummy
```

Figure 22: A snapshot of the original and normalized function

The containment scores and similarity scores between samples are listed on the page of malware samples similarity in descending order of the similarity score. The similarity scores between individual functions may in many cases be more valuable to the analyst than the overall similarity score between samples. The function similarity details can help analyst identify which functions are similar in the two samples. The function pairs with inexact matching scores higher than a prespecified threshold (default is 70%) are listed in descending order and the exactly matched ones are marked in the list. The name of the function in the listed pairs can be clicked to access the original instructions of the corresponding function.

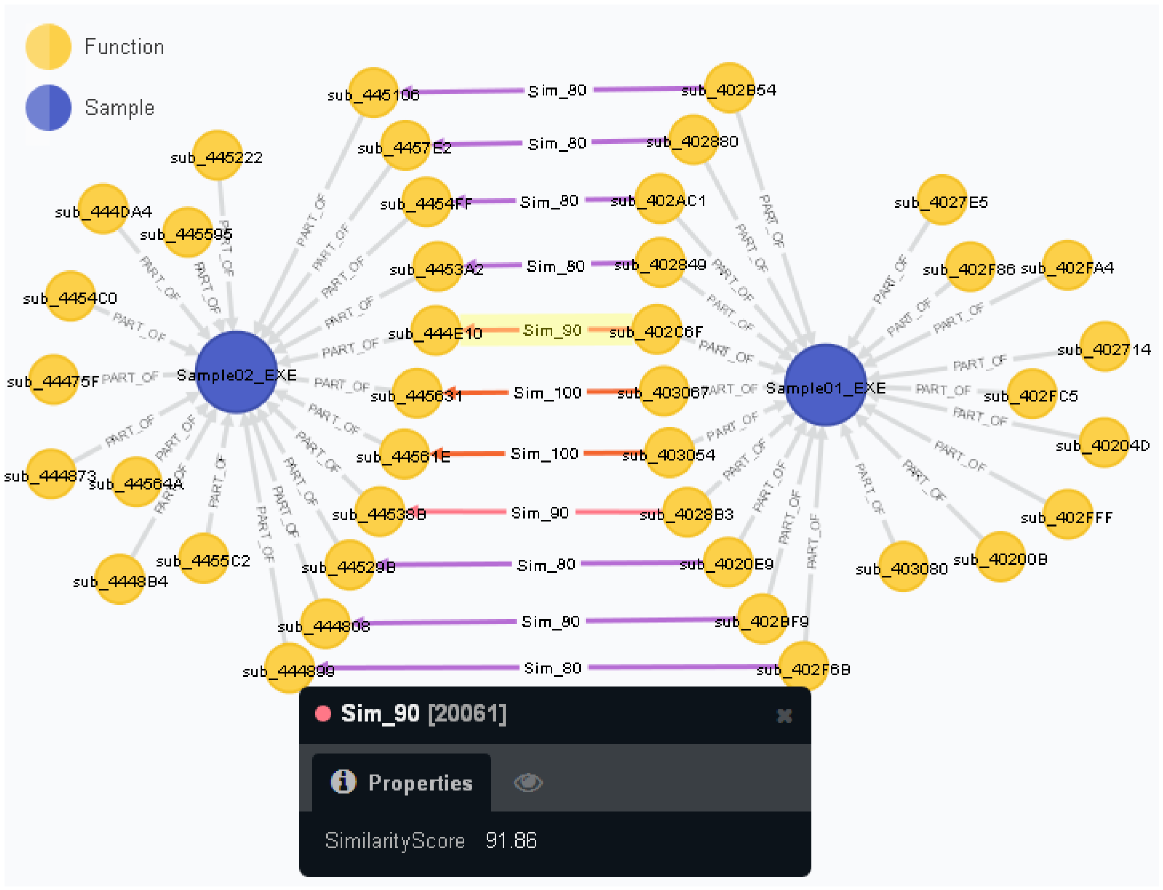
3.4.2 Relationship Graph for Malware Samples

We not only store all sample information and similarity detection results in traditional database, but also upload this data to a graph database (Neo4j [3]) that is used to visualize the relationships between samples and functions. The analysts also can locate specific functions with high similarity scores, or functions importing unusual API classes by using existing functionality of the graph browser and the underlying graph query language.

As indicated in Table 11, we use three labels for the nodes of samples, the nodes of functions and the nodes of API classes in the graph. There are three types of relationship between nodes defined in the associated graph. The *PART_OF* label represents the relationship between the sample and the function that belongs to it. The *CALL* label represents the relationship between the function and the API class imported by it. The *Sim_Score* label represents the range of similarity score between similar functions.

Node labels		
Sample	Function	API
Relationship types		
PART_OF	CALL	Sim_Score

Table 11: Node labels and relationship types of graphs



Sim_90 [20061]

Properties

SimilarityScore 91.86

Figure 23: An example of a graph showing relationships between two samples

The Relationship Visualizer can be used to show all nodes and relationships between nodes in the graph database. For example, using the result of similarity analysis on two malware samples, the nodes of samples and functions are constructed with different colors and the relationships with labels showing the similarity scores are linked between similar functions. As shown in Figure 23, the exact similarity score between any two functions can be displayed by clicking the corresponding link between them.

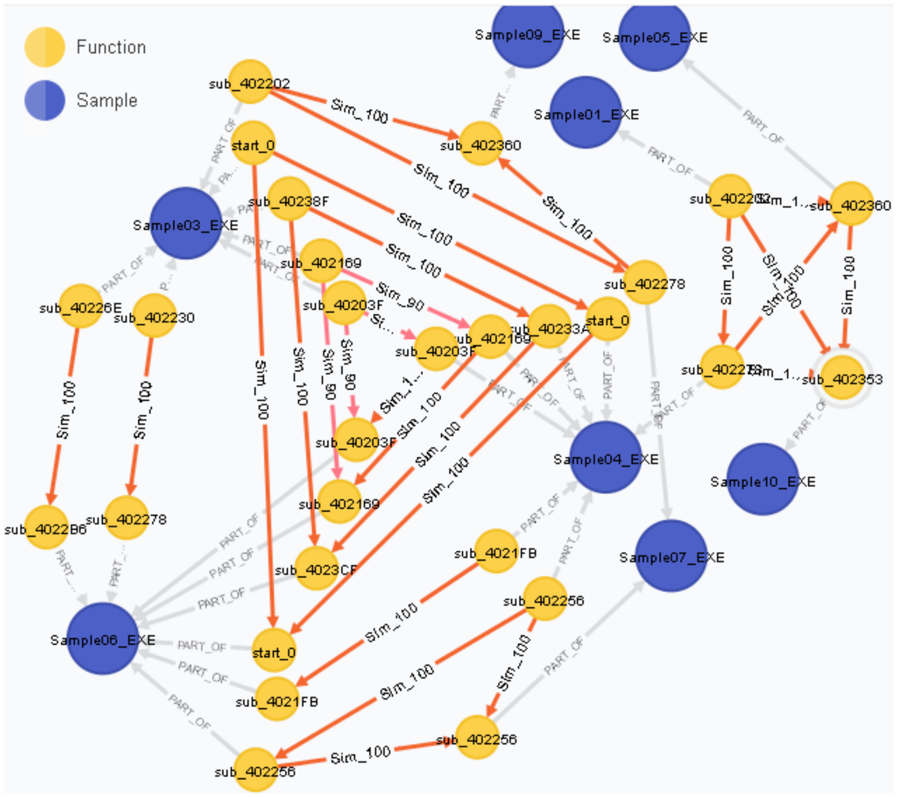


Figure 24: An example of a graph showing relationships between eight samples

When performing similarity analysis on samples with a lot of functions, we can hide the functions without similarity relationship by using graph query language. In Figure 24, only the nodes of functions that have similar functions to match and the nodes of samples that the functions belong to are shown. Using the web interface and the graph application, the analyst is able to locate the functions shared by malware variants and perform manual analysis on assembly code to understand the common malicious functionality of malware variants.

Chapter 4

Experimental Results

In order to evaluate the effectiveness of our proposed framework, we performed three experimental studies using metamorphic virus families created by malware generators, malware variants downloaded from the Internet, and two botnet Trojans. In particular, in the first experiment, our framework is applied to a metamorphic malware detection problem in which metamorphic viruses created by the malware generators are detected and analyzed. In order to evaluate the ability of the developed framework to measure the similarity between real-life malware variants, we tested it against the malware mutant set used in [90]. Finally, in the third experiment, we used our framework to analyze and study the similarity between two complex botnet trojans, namely Zeus and Citadel.

All our experiments were performed on a virtual machine running 32-bit Windows XP with 4GB of RAM. IDA Pro 6.1 was utilized during the automatic disassembly process used in our framework. We used PEiD [70] to check whether the malware samples were packed. If so, several tools, such as UPX [49] and Immunity debugger [31] were used to unpack the analyzed malware samples.

It should be noted that the disassembly process of our framework relies on IDA Pro to do automatic function identification and this may not always work correctly. The incomplete or incorrect disassembly of malware samples may have an effect on the number of recovered functions and their associated assembly instructions.

4.1 Metamorphic Viruses Detection

The metamorphic virus generates copies of itself using code obfuscation techniques. Traditional signature detecting methods cannot easily detect obfuscated virus variants. In this test, we apply our framework to the metamorphic virus detection problem. The metamorphic viruses used in this experiment are created by several malware generation tools and some of these viruses have been active in the wild on real computer systems.

To detect metamorphic viruses, we first need to choose a similarity-scoring threshold which we obtained as follows. First, we evaluate the range of similarity scores for the metamorphic virus families by calculating the similarity scores for all pairs of the virus variants. Then we evaluate the range of similarity scores for the metamorphic viruses versus a set of representative benign files by calculating the similarity scores for all pairs. Finally, we determine the threshold by analyzing the obtained experimental results. If the two distributions of similarity scores in the two ranges are disjoint, namely the lowest score in the first range is higher than the highest score in the second range, any threshold between these two extremes can be used for ideal detection without false positives or false negatives. In addition, we also performed similarity analysis between benign files and between viruses from different metamorphic families.

4.1.1 Dataset

The families of metamorphic viruses used for testing were produced using the Next Generation Virus Creation Kits (NGVCK), Second Generation Virus Generator (G2), and Virus Creation Lab for Win32 (VCL32) as following:

- **Next Generation Virus Creation Kits** 20 virus variants generated by NGVCK.

These viruses can infect Win32 PEs and Win32 DLLs. An obfuscation technique is used in this creation kit to generate variants that are different in structure and assembly code. The viruses of this family are shown to be the most metamorphic among the viruses generated and tested in [88].

- **Second Generation Virus Generator** 20 virus variants generated by G2 that was written by the underground group Phalcon-Skism. These viruses are COM/EXE infectors and body-polymorphic which makes them look different while keeping the same functionality.

- **Virus Creation Lab for Win32** 20 virus variants generated by VCL32 that was created by the group "29A". These viruses specifically infect Windows files and are claimed to be metamorphic. The assembly codes of each virus are different because of the generator options. Changing the configuration of this generator can add or delete functions from virus, which is a challenge for our similarity detection that is based on similarity between functions.

To obtain the binaries corresponding to the above virus samples, we assembled their *.asm* files using Borland Turbo Assembler, TASM 5.0. We randomly selected 10 Cygwin utilities files as representatives of benign files throughout this experiment. Some of these Cygwin files have also served as representative of benign programs in previous works including [10, 64, 88].

4.1.2 Analysis Results

First, we applied the similarity analysis on the family of NGVCK viruses, which was reported as the most metamorphic family among the ones tested in [88]. Compared to the other two metamorphic families, the viruses of the NGVCK family have a far lower degree of similarity. Figure 25 (a) shows the similarity results where the red points correspond to the similarity scores between virus comparison pairs. The similarity scores between virus-benign pairs are represented by the blue points. Unlike G2 and VCL32, NGVCK viruses also differ significantly in size which is a result of inserting random number of junk instructions in the virus body. We also notice that the maximum similarity score between NGVCK viruses, which is 84.47%, is less than any maximum scores for the other two virus families. The maximum similarity score between the NGVCK virus family and benign files is 37.71%. Thus we can take this minimum score from the virus pair comparisons and set it as the threshold without any false positive detection.

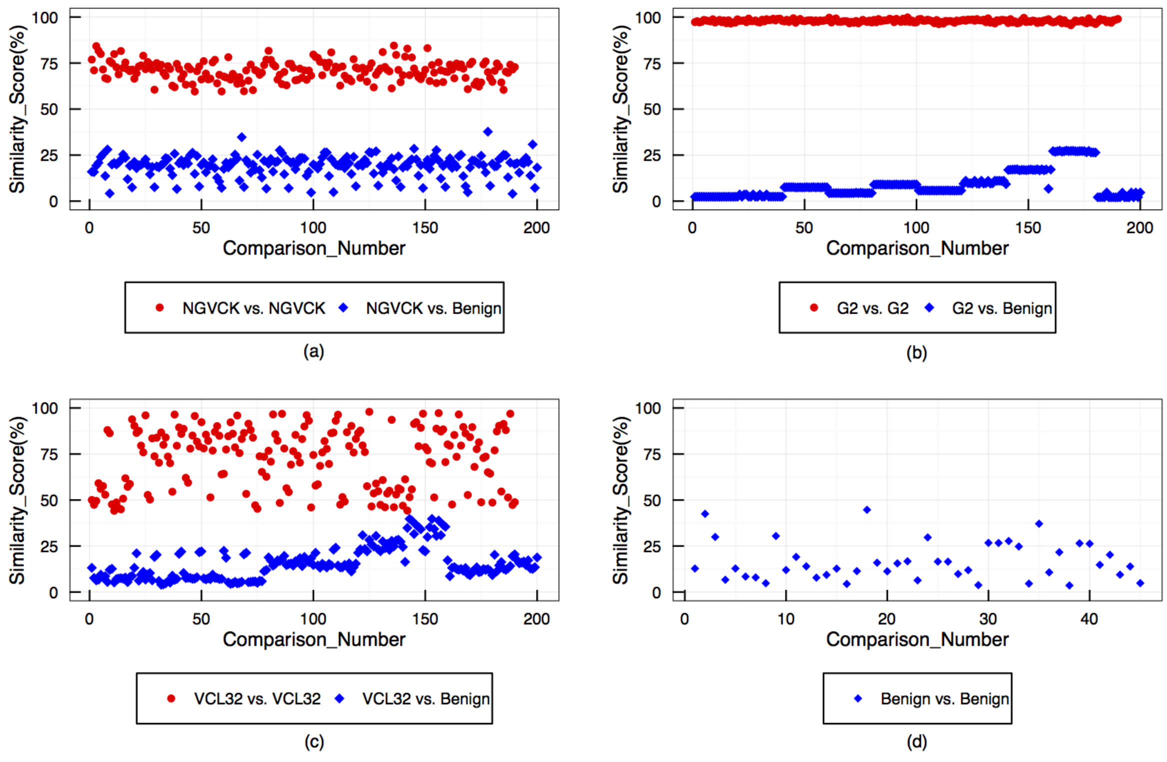


Figure 25: Similarity scores between metamorphic viruses and benign programs

The results in [88] showed that, among the consider generators, the G2 family has the highest average similarity score between its variants. The experimental results using this virus family is shown in Figure 25 (b). The G2 viruses are almost identical to one another and the similarity scores are all above 95%. We can see that the separation between virus comparison pairs and virus-benign comparison pairs is largest among these three metamorphic families. The G2 viruses can be detected with no false positives or false negatives using our framework.

The last tested virus family is VCL32. As shown in Figure 25, while some of the similarity scores of virus comparison pairs within this family are higher than 90%, the minimum score is about 44.17%. However, if we take this score as the detection threshold, all the virus variants can still be detected with false positive rate of 0%.

In addition to the above experiments, we also measured similarity between the representative benign files.The maximum and average score of benign comparison pair is 44.69% and 16.56%, respectively. We assumed that the benign files belong to different families based on their functionalities. Since they are Cygwin utilities files, it is possible that they have common functions. However, only three pairs of files have similarity higher than 30% and there are identical functions detected in each of these pairs such as the functions related to DLL (Dynamic-link) library and CRT (C run-time) library.

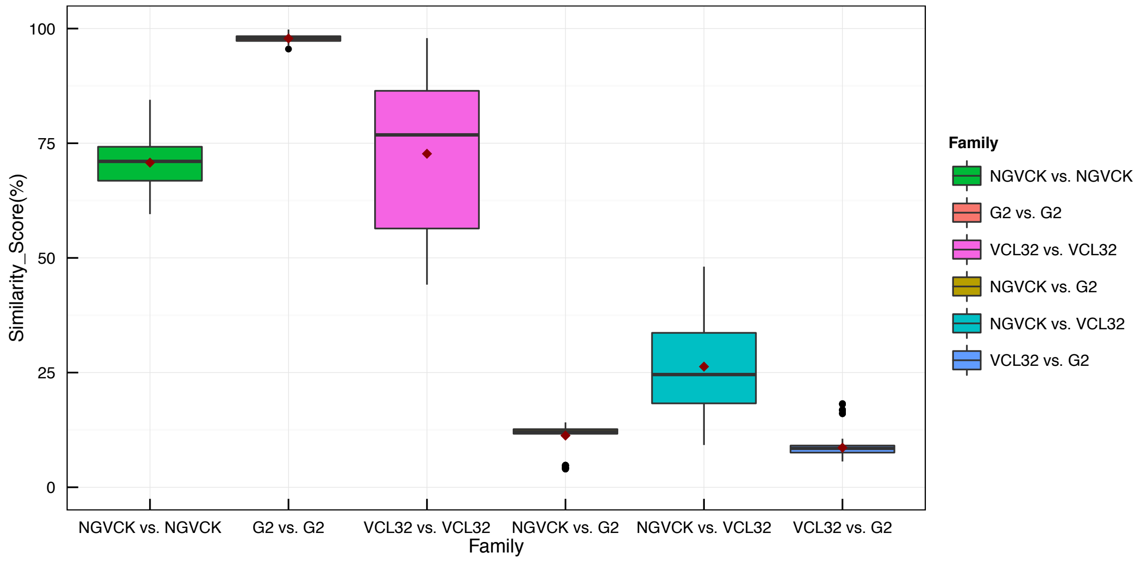


Figure 26: Box plot of similarity scores between virus families

After the experiment of detecting metamorphic virus, we apply our framework to distinguish the different metamorphic virus families by calculating similarity score between viruses of different families and setting a threshold. The distribution of similarity scores between virus families is shown in Figure 26, where the red point represents the average similarity score for each virus family comparison and the horizontal black line indicate the median score. The color box goes from the 25th percentile to the 75th percentile of the similarity scores in each virus family comparison (this is known as the *inter-quartile range* (IQR) [77]). The vertical black lines start from the edge of the color box and extend to the furthest score point that is within 1.5 times the IQR. If there are any score points that are past the ends of the vertical black lines, they are considered outliers and displayed with dots. The minimum, maximum and average similarity scores in Figure 26 are summarized in Table 12. We can see that the average similarity scores between viruses of the same family are significantly higher than the ones between viruses of different families. If we set the minimum sore from the VCL32 virus pair comparisons as the threshold, 2% of NGVCK-VCL32 virus pair scores become higher than the threshold. The reason for this is that very few comparisons between functions of NGVCK and VCL32 viruses have similarity score between 50% and 60%, which contributes the similarity score between viruses in these two families.

%	NGVCK vs. NGVCK	G2 vs. G2	VCL32 vs. VCL32	NGVCK vs. G2	NGVCK vs. VCL32	VCL32 vs. G2
Min	59.55	95.54	44.17	3.97	9.71	5.62
Max	84.47	99.79	97.92	14.14	48.10	18.23
Average	70.76	97.85	72.82	11.27	26.30	8.62

Table 12: Similarity scores between families of metamorphic viruses

Based on the results reported in [88], the NGVCK viruses use different types of code obfuscation techniques. We selected three of the NGVCK viruses for in-depth analysis and the relationship between them is shown in Figure 27. Our framework detected 6 functions shared by these three virus samples which is depicted by the circle in the middle of diagram. Between any two viruses, there are also function pair comparisons with similarity score higher than 80%. Two function pair comparisons between *sub_402880* and *sub_-4457E2*, and between *sub_403080* and *sub_4454C0* are illustrated in Figure 28 and Figure 29, respectively. Here, the function names such as *sub_402880* are automatically generated by IDA Pro during the disassembly process. The analyst can rename these functions with other strings. The similarity scores between these two pairs are 84.42% and 79.13%, respectively.

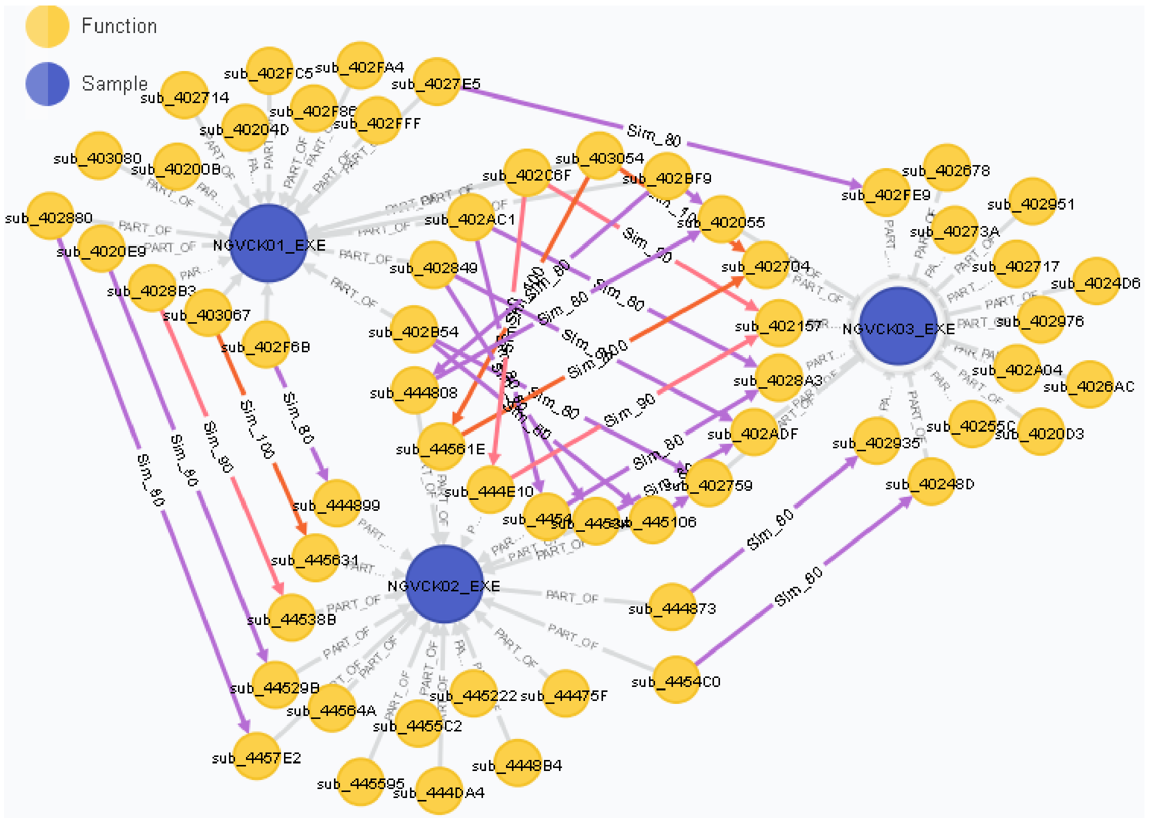
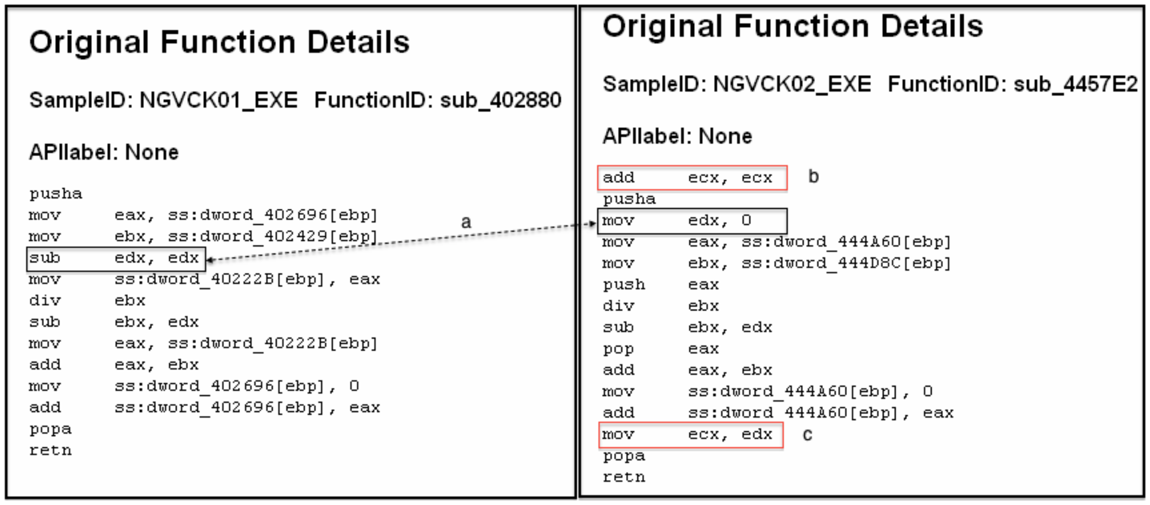


Figure 27: Visualization of realtionship between three NGVCK viruses

Both function *sub_402880* and function *sub_4457E2* are used for file alignment during the virus infection process. The instructions of *sub_4457E2* with red rectangle are junk instructions that are not related to the function's outcome and their only purpose is to make



Original Function Details

SampleID: NGVCK01_EXE FunctionID: sub_402880

APIlabel: None

```
pusha
mov     eax, ss:dword_402696[ebp]
mov     ebx, ss:dword_402429[ebp]
sub     edx, edx
mov     ss:dword_40222B[ebp], eax
div     ebx
sub     ebx, edx
mov     eax, ss:dword_40222B[ebp]
add     eax, ebx
mov     ss:dword_402696[ebp], 0
add     ss:dword_402696[ebp], eax
popa
retn
```

Original Function Details

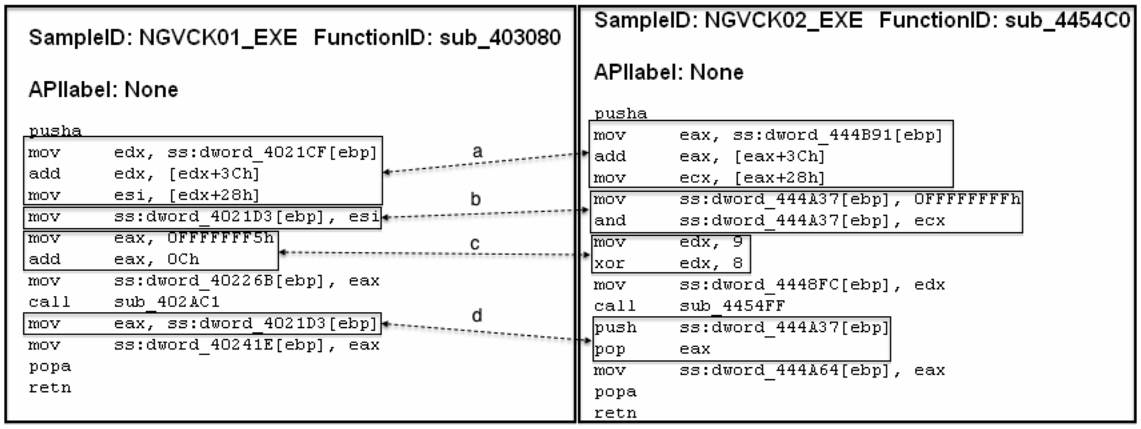
SampleID: NGVCK02_EXE FunctionID: sub_4457E2

APIlabel: None

```
add     ecx, ecx
pusha
mov     edx, 0
mov     eax, ss:dword_444A60[ebp]
mov     ebx, ss:dword_444D8C[ebp]
push    eax
div     ebx
sub     ebx, edx
pop     eax
add     eax, ebx
mov     ss:dword_444A60[ebp], 0
add     ss:dword_444A60[ebp], eax
mov     ecx, edx
popa
retn
```

Figure 28: Comparison between *sub_402880* and *sub_4457E2*

the virus variants look different. The function *sub_4457E2* replaces the instruction of *sub_-402880* "sub edx, edx" with the instruction "mov edx, 0"; both instructions result in clearing the register edx. Both *sub_403080* and *sub_4454C0* are used to get entry point in memory during the infection procedure. We can see that the first three instruction regions, a, b, and c, apply the technique of register swapping to make operands different among functions. The instruction regions in comparison c are both equivalent to the instruction "mov eax/edx, 1". In the last comparison region , d, the "mov register, memory data" is equivalent to "push memory data" followed by "pop register".



SampleID: NGVCK01_EXE FunctionID: sub_403080

APIlabel: None

```
pusha
mov     edx, ss:dword_4021CF[ebp]
add     edx, [edx+3Ch]
mov     esi, [edx+28h]
mov     ss:dword_4021D3[ebp], esi
mov     eax, 0FFFFFFF5h
add     eax, 0Ch
mov     ss:dword_40226B[ebp], eax
call    sub_402AC1
mov     eax, ss:dword_4021D3[ebp]
mov     ss:dword_40241E[ebp], eax
popa
retn
```

SampleID: NGVCK02_EXE FunctionID: sub_4454C0

APIlabel: None

```
pusha
mov     eax, ss:dword_444B91[ebp]
add     eax, [eax+3Ch]
mov     ecx, [eax+28h]
mov     ss:dword_444A37[ebp], 0FFFFFFFFh
and     ss:dword_444A37[ebp], ecx
mov     edx, 9
xor     edx, 8
mov     ss:dword_4448FC[ebp], edx
call    sub_4454FF
push    ss:dword_444A37[ebp]
pop     eax
mov     ss:dword_444A64[ebp], eax
popa
retn
```

Figure 29: Comparison between *sub_403080* and *sub_4454C0*

4.2 Measuring Similarity between Real-life Malware Variants

The majority of malware floating around are variants of already known malware. Malware writers reuse their old malicious code in the new malware variants, which is easier compared to trying to developing new effective, mostly bug-free malware from scratch. Different from the experiment reported in the previous section, the malware variants considered in this experiment are collected from the Internet rather than generated by us. In particular, in this section, we evaluate our framework using five families of malware variants that were downloaded from VX Heaven [6] and compare similarity results with the previous work reported in [90]. The experiment consists of computing similarity between malware variants and distinguishing the different malware families.

4.2.1 Dataset

In order to validate the performance of the proposed technique on a large variety of malware, we use different types of malware for testing including viruses, worms, backdoors, trojan horses and rootkits. The information related to each malware family can be briefly summarized as follows:

- **Virus.Sality** We use 6 malware variants that belong to Sality [85], which is a family of polymorphic viruses that spread by infecting Windows executable files. A computer infected with Sality becomes a bot that has capabilities of relaying spam, infecting web servers, and stealing sensitive information via peer-to-peer networks. Sality variants evolve with new malicious functionalities such as rootkit and Trojan components, which makes Sality one of the most effective and resilient malware to date [27]. The number of functions in variants of Sality disassembled by IDA Pro ranges from 157 to 220 functions.

- **Virus.Champ** 10 viruses are selected from Champ family [5], which is a dangerous family of non-memory resident parasitic polymorphic Win32 viruses which spread by infecting executable files. The number of disassembled functions in the tested variants did not change a lot and ranged from 45 to 47.

- **Email-Worm.Klez** The Klez family [79] used for testing includes 9 variants. Klez is an Internet worm which is capable of spoofing email addresses and launching automatically when a victim previews an e-mail containing it. Klez spawned a significant number of variants with increasingly clever self-distribution mechanisms. For this family, the number of functions, identified by IDA Pro ranges from 174 to 239.

- **Backdoor.DarkMoon** 8 variants of DarkMoon [1] are used for evaluating our method. DarkMoon is a popular remote access Trojans (RAT) used in targeted attacks such as distributed denial of service (DDoS) attacks. These DarkMoon variants open backdoor on different ports on the compromised computers and steal personal information via keylogging. Among these five malware families, DarkMoon variants have the largest number of statically disassembled functions, which ranges from 301 to 399 functions.

- **Rootkit.KernelBot** [4] 8 variants of this rootkit. These programs are used to hide Windows registry entries and processes with the purpose of camouflaging malicious programs running on compromised computers. The number of functions recovered by IDA Pro ranges from 13 to 19. This small number of recovered functions might not be the actual number of functions within this rootkit but rather the ones that IDA Pro was able to recover because the other ones were obfuscated with techniques that IDA Pro was not able to handle.

4.2.2 Analysis Results

We first compute similarity between variants belonging to Sality family to test our method. In Table 13, the similarity matrix shows the similarity scores between each one of the 15 pairs of these variants. Throughout the results presented in this section, the values (100%) in the main diagonal denote the cases when the variants are compared with themselves. The experimental results using our method are shown below the main diagonal, and the similarity scores above the main diagonal are results reported in [90]. Compared to these previous results, the maximum score in our proposed method, between variants Sality$_a$ and Sality$_c$ is 98.89% which is lower than 99.32% reported in [90]. However, our method improves the minimal score between Sality$_c$ and Sality$_f$, from 53.75% to 67.24%. The average similarity score in our method is 80.85%, which is higher than 76.44% reported in [90] which shows that our method performs better for this malware family.

%	Sality$_a$	Sality$_c$	Sality$_d$	Sality$_e$	Sality$_f$	Sality$_g$
Sality$_a$	**100**	99.32	99.31	66.46	60.61	81.73
Sality$_c$	98.89	**100**	98.87	65.96	53.75	87.79
Sality$_d$	96.57	96.34	**100**	68.02	60.48	83.55
Sality$_e$	68.78	68.6	70.2	**100**	89.70	66.46
Sality$_f$	67.36	67.24	68.53	96.88	**100**	64.62
Sality$_g$	88.37	88.34	86.95	75.46	74.17	**100**

Table 13: Similarity scores between Sality variants

Table 14 shows the similarity scores between variants of the Klez family. The Symbol "N/A" in the table indicates that the corresponding variant pair is not compared in [90] and hence its corresponding score is absent. In the common 16 variant comparisons, most similarity scores are higher than 90%, both in the previous work and in ours. The minimal and average score computed by our method are 87.85% and 94.59% which are higher than 79.73% and 93.40% computed in the previous work. We also applied our method to compare between other variants which are not covered in previous work. The similarity scores between Klez variants (*a-d*) and Klez variants (*f-j*) are lower than the scores of the common comparisons and the minimal score is 68.86%. This is probably because variants from Klez$_f$ and Klez$_j$ have more new functions updated by the malware author as compared to previous versions. Based on the gap between similarity scores, this family can be split into two branches, one containing Klez variants *a*, *b*, *c*, *d* and a second one with *f*, *g*, *h*, *i*, *j*, which is similar to the conclusion reported in [17].

%	Klez$_a$	Klez$_b$	Klez$_c$	Klez$_d$	Klez$_f$	Klez$_g$	Klez$_h$	Klez$_i$	Klez$_j$
Klez$_a$	**100**	94.40	99.35	79.73	N/A	N/A	N/A	N/A	N/A
Klez$_b$	93.32	**100**	94.84	90.75	N/A	N/A	N/A	N/A	N/A
Klez$_c$	96.18	93.65	**100**	81.45	N/A	N/A	N/A	N/A	N/A
Klez$_d$	87.85	90.93	88.13	**100**	N/A	N/A	N/A	N/A	N/A
Klez$_f$	70.04	70.03	69.75	72.8	**100**	98.98	93.04	93.09	99.10
Klez$_g$	69.61	69.61	69.33	72.4	97.51	**100**	92.92	92.97	98.98
Klez$_h$	68.86	68.86	68.57	71.18	95.54	95.25	**100**	99.06	92.81
Klez$_i$	68.87	68.87	68.58	71.19	95.55	95.27	97.82	**100**	92.87
Klez$_j$	70.04	70.03	69.75	72.8	97.79	97.51	95.54	95.55	**100**

Table 14: Similarity scores between Klez variants

The similarity scores between the variants of DarkMoon family is given in the matrix shown in Table 15 where "DM" represents DarkMoon. As mentioned in the above experiments, we also compare the similarity results computed by our proposed method with the previous results in [90]. The minimal similarity score, between DM$_{ab}$ and DM$_{at}$, is improved by our method from 55.52% to 70.16%. Half of the scores of comparison pairs are lower than the corresponding results in [90]. However, the average similarity score of the DarkMoon family is 81.75%, which is higher than 80.75% reported in [90].

Table 16 shows the results corresponding to the mutations of KernelBot rootkit family where KernelBot is donated by "KB". According to this matrix, all the similarity scores calculated by our method are higher than 73.87%. For the previous results [90], 4 scores of variant comparison pair are lower than 50%. Overall, the average similarity score calculated by our method is 81.88% which is far better than 68.53% reported in [90].

%	DM_{ab}	DM_{ad}	DM_{ae}	DM_{ah}	DM_{ai}	DM_{ar}	DM_{at}	DM_{aw}
DM_{ab}	**100**	75.09	78.25	70.34	69.41	75.08	55.52	71.57
DM_{ad}	84	**100**	92.77	92.14	91.42	86.99	74.92	78.01
DM_{ae}	83.69	91.76	**100**	90.86	88.04	92.08	65.57	83.26
DM_{ah}	81.51	90.34	91.85	**100**	95.46	91.36	76.41	76.81
DM_{ai}	79.01	87	88.51	89.44	**100**	94.38	76.54	78.25
DM_{ar}	80.09	87.17	89.42	89.62	93.49	**100**	73.04	82.24
DM_{at}	70.16	72.42	72.79	72.38	75.19	75.2	**100**	85.10
DM_{aw}	73.17	74.45	76.42	75.01	78.21	79.12	87.65	**100**

Table 15: Similarity scores between DarkMoon variants

%	KB_{a}	KB_{ah}	KB_{ak}	KB_{al}	KB_{as}	KB_{at}	KB_{aw}	KB_{az}
KB_{a}	**100**	95.00	65.88	73.91	64.16	63.77	95.89	89.86
KB_{ah}	89.5	**100**	62.92	70.83	48.85	60.27	90.91	84.93
KB_{ak}	84.66	77.79	**100**	91.09	48.42	58.97	63.41	71.79
KB_{al}	84.9	80.81	97.09	**100**	45.10	54.12	69.66	65.88
KB_{as}	81.15	75.92	78.66	78.6	**100**	78.48	48.19	53.16
KB_{at}	84.19	74.88	76.44	76.19	81.21	**100**	60.61	67.74
KB_{aw}	98.61	87.89	80.47	83.24	79.6	82.66	**100**	87.88
KB_{az}	88.07	78.62	77.21	74.67	73.87	77.09	88.57	**100**

Table 16: Similarity scores between KernelBot variants

The last tested malware family is Champ and the similarity results are shown in Table 17, where "C" means Champ. We can see that all scores of Champ variant comparisons except the comparisons containing C_{7001} with the method in [90] are higher than the scores evaluated by our method. However, the minimal score obtained using our approach, 67.27%, is still higher than 62.58% which is the smallest score computed by the previous method.

%	C	C_{5430}	C_{5447}	C_{5464}	C_{5477}	C_{5521}	C_{5536}	C_{5714}	C_{5722}	C_{7001}
C	**100**	92.40	92.40	97.67	95.29	95.91	94.55	100	97.65	62.58
C_{5430}	91.28	**100**	100	94.74	97.04	96.47	91.46	92.40	93.49	66.67
C_{5447}	91.28	97.09	**100**	94.74	97.04	96.47	91.46	92.40	93.49	66.67
C_{5464}	90.97	94.02	94.02	**100**	97.65	98.25	93.33	97.67	96.47	65.03
C_{5477}	90.86	95.82	95.82	93.66	**100**	99.41	93.25	95.29	95.24	67.08
C_{5521}	90.86	95.82	95.82	93.66	95.45	**100**	92.68	95.91	94.67	66.67
C_{5536}	91.09	89.96	89.96	89.71	89.62	89.62	**100**	93.33	96.93	62.82
C_{5714}	94.46	91.2	91.2	91.3	90.85	90.85	91.74	**100**	97.65	62.58
C_{5722}	93.34	90.68	90.68	90.79	90.34	90.34	92.83	93.96	**100**	64.60
C_{7001}	68.01	70.62	70.62	70	70.2	70.2	67.27	67.59	68.51	**100**

Table 17: Similarity scores between Champ variants

In order to evaluate the ability of the proposed method to distinguish between different malware families, 150 non-variant pairs are randomly selected from all variants of the five malware families. Figure 30 shows the distribution of similarity scores of all comparison pairs, where the *x*-axis represents the comparison pairs, and the *y*-axis represents the scores of each pair. We use red points to donate pairs belonging to the same malware family and blue points to donate pairs belonging to different families. The results show that all the similarity scores between variants belonging to different families are lower than 25%. The maximum score of non-variant pairs is 24.04%. In the comparisons between malware variants, the minimal score is 67.24%. Compared to the results in [90], there is no overlap between these two types and hence a threshold can be set to easily distinguish between these different malware families. The distribution of similarity scores between the five families is shown in Figure 31, where the red point represents the average similarity score for each family comparison. Obviously, the similarity scores between variants of the same family are far higher than the scores between variants belonging to different families.

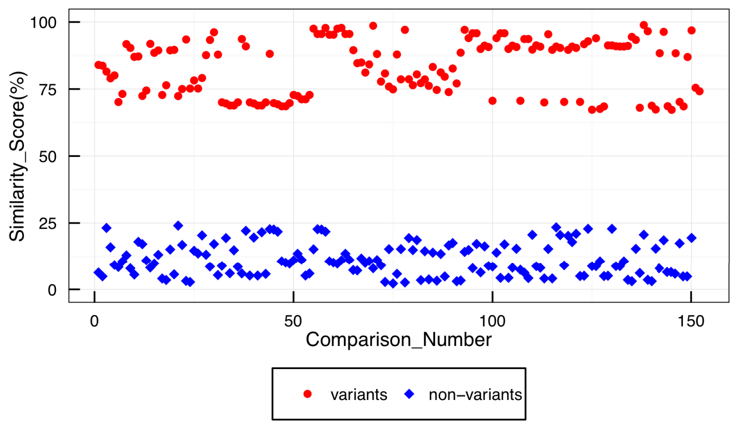


Figure 30: Pair-wise similarity among the same/different malware families

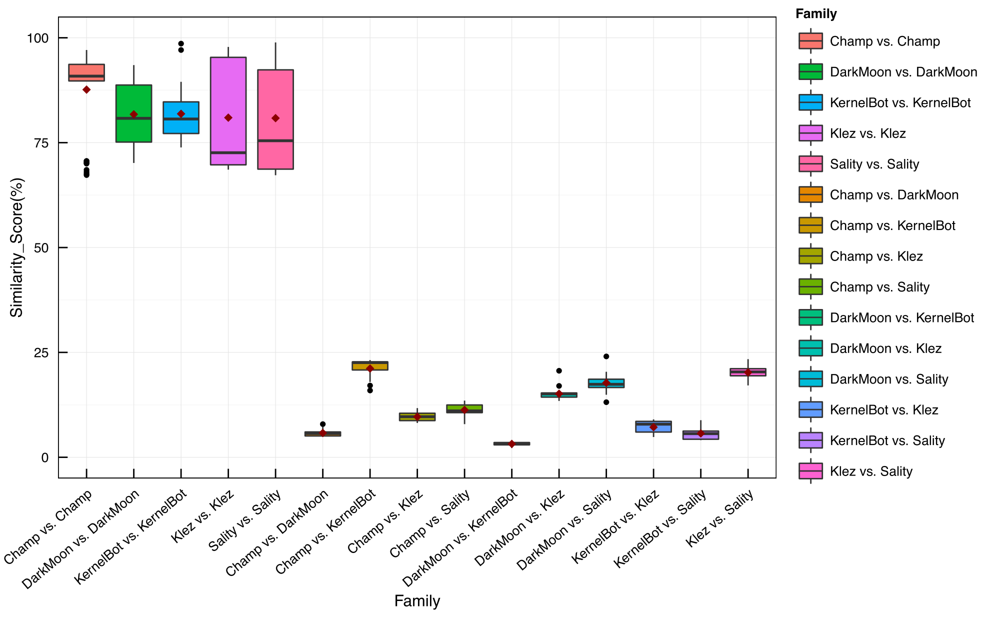


Figure 31: Box plot of similarity scores between variant families

4.3 Botnet Trojans Analysis

Zeus [87] and Citadel [46] are two notorious botnet trojans that infect computers and steal personal information. The Citadel Trojan is an offspring of the Zues Trojan [72], i.e., the Cidatel Trojan has reused some of the source code of Zeus. In this section, we apply similarity analysis on these two Trojans using our proposed framework.

4.3.1 Dataset

As one variant of the Zeus-family, Citadel shares some of its code with Zeus. The similarity between these two malware has been analyzed in previous works including [59, 72].

The main features of Zeus can be summarized as follows [37]:

- Steals data in HTTP forms, such as banking account information.
- Modifies the HTML pages of the target websites within the web browser.
- Redirects target web pages to ones controlled by the attacker.
- Uploads files from the victim's computer.
- Downloads and executes arbitrary programs.

The new features of Citadel can be summarized as follows [46]:

- Uses modified encryption for bot communications with C&C servers.
- Detects if it is running within a virtual machine or sandbox.
- Changes the behavior of domain names resolution on infected computers.
- Covers a much larger range of Windows functions to be hooked.
- Manipulates the infected computer as a bot to participate a distributed denial of service (DDoS) attack.

4.3.2 Analysis Results

Using our framework, the similarity score between these two trojan is evaluated to 74.17%, which is almost same as the results reported in [72]. We also confirmed the relationship between the two bots by calculating their containment scores which are 91.52% for Zeus and 61.75% for Citadel. Compared to the previous results reported in [59] using binary clone analysis, the same conclusion was reached, namely Citadel copied most of its functions from Zeus and improved itself by adding new functions. 465 unique exact matching function pairs are detected by our similarity analysis.

Trojan	# of Function	# of Function with API Class	# of Unique API Class
Zeus 2.1.0.1	605	262	45
Citadel 1.3.5.1	845	395	47

Table 18: Statistics extracted from Zeus and Citadel

Table 18 shows the number of analyzed functions, number of functions with API classes, and the number of unique API Classes for these two Trojans. A function with API Class is one which imports APIs that are identified by our API class recognizer. The numbers of unique API classes identified in Zeus and Citadel are almost the same; Citadel has two additional API classes, *Directory* and *Pipe*.

Figure 32 shows the component categories of APIs identified in these two trojans. For simplicity, we only show the API class whose frequency in these two trojans is higher than 10. The bar graph stacks are proportional to the number of functions labeled with the same API class. As depicted in the figure, the API classes identified in these two trojans are very similar. The difference is that Citadel imports more API related to string and file operations. For *String* class, Citadel has 74 functions with this label but Zeus only has only 17. For *File* class, Citadel has 46 functions with this label while Zeus has 7. The *Others* consists of the API classes identified in less than 10 functions, such as *Authorization*, *Hashing*, and

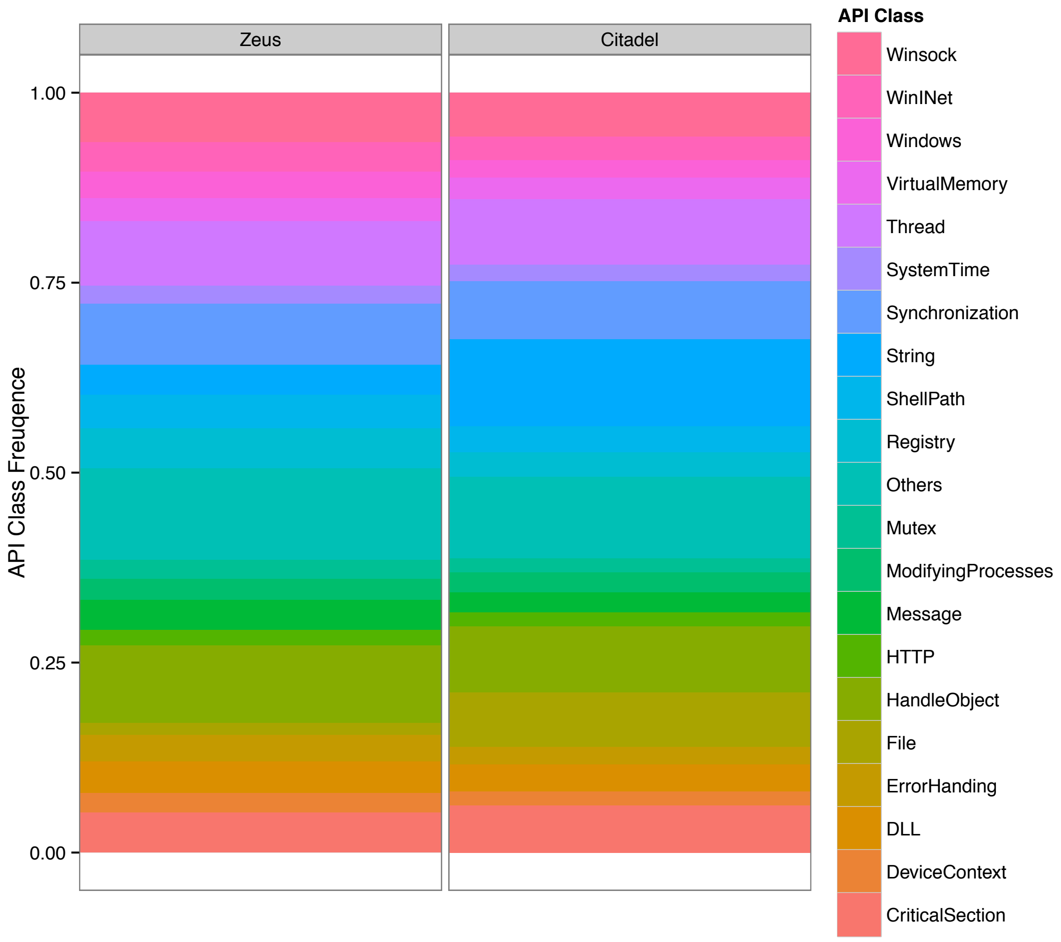


Figure 32: The API classes identified in Zeus and Citadel

Cryptography, and *SystemShutdown*.

Based on previous analysis of Zeus and Citadel [59, 89], we know that these two Trojans use cryptographic methods such RC4 to encrypt configuration files and stolen data. Compared to the encryption methods applied in Zeus, Citadel provides better security by using AES instead of RC4 to encrypt configuration files from the C&C server. In what follows, we focus on cryptographic functions in these two Trojans.

To identify the functions related to encryption algorithms and locate similar ones between Zeus and Citadel, we query the functions that have *CALL* relationship with API class *Hashing* and *Cryptography* and also have similarity relationship with any function of the

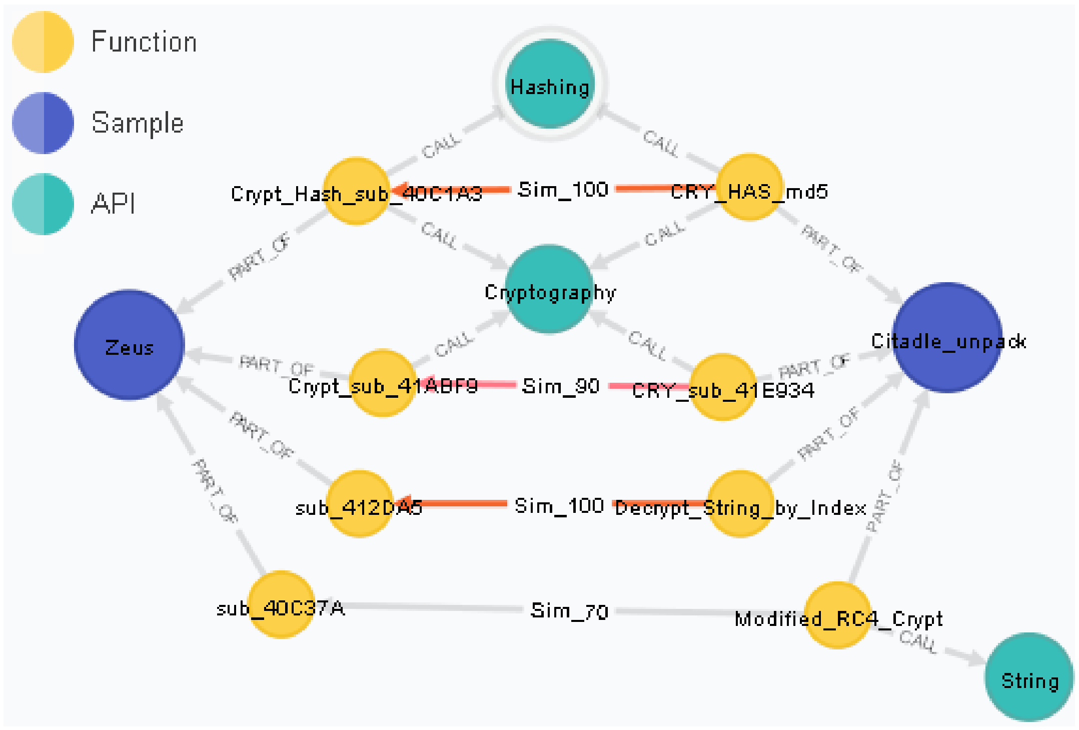


Figure 33: Visualization of cryptographic function pairs

other trojan from our graph database. Based on previous analysis [59], we also add functions that do not call any specific APIs but perform encryption or decryption operations as shown in Figure 33. The exact similarity scores and API information from Figure 33 are summarized in Table 19.

Zeus	**Citadel**	**SimialrityScore**	**API Class**
Crypto_Hash_sub_40C1A3	CRY_HAS_MD5	100%(Exact Matching)	Cryptography, Hashing
sub_412DA5	Decrypt_String_by_Index	100%(Exact Matching)	None
Crypto_sub_41ABF9	CRY_sub_41E934	95.73%	Cryptography
sub_40C37A	Modified_RC4_Crypt	74.74%	String

Table 19: Similarity scores and API classes of selected cryptographic functions in Zues and Citadel

In Table 19, the functions *Crypto_Hash_sub_40C1A3* and *CRY_HAS_MD5* in the first function comparison pair performs MD5 hashing operation. Here, we rename the functions in IDA Pro to indicate the specific operation performed by them such as *HAS*, representing hashing operation. The imported APIs identified and classified into *Cryptography* and

Hashing API classes by the API class recognizer consist of *CryptAcquireContextW*, *CryptReleaseContext*, *CryptCreateHash*, *CryptHashData*, *CryptGetHashParam*, and *CryptDestroyHash*. We can see that Citadel completely copies its MD5 function implementation from Zeus. There are a few strings in Zeus and Citadel that are stored in an encrypted format. When the functions read these encrypted strings, they call *sub_412DA5* and *Decrypt_String_by_Index* to decrypt them and store them in a buffer on the stack. These two function are also identical among Zeus and Citadel. *Crypto_sub_41ABF9* and *sub_41E934* are related to *Cryptography* since they both have imported the API *CryptUnprotectData* that usually decrypts and does an integrity check of data. *sub_40C37A* performs RC4 encryption in Zeus while *Modified_RC4_Crypt* is a special crafted RC4 function with additional XOR operation for Citadel. Although the RC4 function of Citadel is modified a lot from the original RC4 function of Zeus, our framework detects that the similarity score between them is about 74.74% and only one function *sub_40C37A* from Zeus is similar to *Modified_RC4_Crypt* with a score higher than 70% .

Chapter 5

Conclusions and Future Work

In order to reduce unnecessary redundant analysis of similar malware code, in this thesis we presented a framework for automated malware similarity analysis. Our goal is to detect similarity between malware samples by matching duplicated, and possibly obfuscated, functions among them. We applied a function matching method that is capable of performing accurate similarity detection on compared functions. In addition, the API classes imported by the target analyzed functions are extracted and classified in order to provide more insight into the activities performed by the malware. With similarity identified on the function level, a novel similarity metric is used to conclude similarity between malware samples. The calculated similarity results are stored in a graph database for visualizing the relationship between analyzed malware samples and for querying specific functions that facilitate the analysts' job. The implemented GUI enhances the usability of the developed system and allows the analysts to identify and browse malicious code in similar functions.

In order to evaluate effectiveness of the proposed framework, three experiments were conducted, including detection of metamorphic viruses created by malware generation tools, identification of malware variants in the wild, and analysis of two botnet trojans. The obtained experimental results show that our framework is capable of detecting malware variants and effective in identifying similar functions shared by them.

It should be noted that our framework has several limitations. For example, it assumes that submitted malware samples are correctly unpacked. It also assumes that submitted malware samples can be reverse engineered into assembly code by IDA Pro. Furthermore, the relationship graph in the visualization module can handle limited number of nodes ($\leq 2^{35}$). In addition to addressing these limitations, we can further develop the prototype framework implemented in this thesis in the following three directions:

1. *Control Flow Graph* (CFG): Functions can be represented as CFGs that express the relationships between (basic) blocks of instructions. In addition to syntactic matching, structural matching on the function level can also be done using the information extracted from these CFGs.

2. *Function Profiling*: The process of function profiling is to determine probable malicious activities that can be performed by functions inside the malware code based on the assembly instructions of these functions. By analyzing the imported APIs, our API Class Recognizer can provide basic information about the function's operations. However, profiling functions without APIs or with unidentified APIs is still a challenging task that requires further investigation.

3. *Data Constants Collection*: Interactive dissemblers, such as IDA Pro that is used in our framework, allow us to access data constants in the disassembled malware. These extracted data constants, such as referenced strings, can be compared during similarity analysis at the function level. If the same referenced string appears in two functions, this information can be used to improve the performance and accuracy of the overall similarity detection process.

Bibliography

[1] Backdoor.Darkmoon. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2005-081910-3934-99.

[2] IDA Pro. [Online]. Available: http://www.hex-rays.com/products/ida.

[3] Neo4j. [Online]. Available: http://www.neo4j.org/.

[4] Rootkit.KernelBot. [Online]. Available: http://www.pandasecurity.com/homeusers/security-info/198629/Kernelbot.A.

[5] Virus.Win32.Champ. [Online]. Available: http://virusview.net/description/family/Virus/Win32.Champ.

[6] VX Heaven. [Online]. Available: http://vxheavens.com/.

[7] C. Annachhatre, T. H. Austin, and M. Stamp. Hidden Markov models for malware classification. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2013.

[8] M. W. Bailey, C. L. Coleman, and J. W. Davidson. Defense against the dark arts. In *ACM SIGCSE Bulletin*, volume 40, pages 315–319. ACM, 2008.

[9] D. Baysa, R. M. Low, and M. Stamp. Structural entropy and metamorphic malware. *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, 2013.

[10] D. Baysa, R. M. Low, and M. Stamp. Structural entropy and metamorphic malware. *Journal of Computer Virology and Hacking Techniques*, 9:179–192, 2013.

[11] V. Bénony. Hopper: The OS X and Linux Disassembler. [Online]. Available: http://www.hopperapp.com/.

[12] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[13] A. Sæbj∅rnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis(ISSTA '09)*, pages 117–128. ACM, 2009.

[14] J. M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[15] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences 1997*, pages 21–29. IEEE, 1997.

[16] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3):255–259, 1998.

[17] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus bulletin conference*, volume 11, 2004.

[18] S. Cesare and Y. Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom),2011*, pages 181–189. IEEE, 2011.

[19] P. Charland, B. C. Fung, and M. R. Farhadi. Clone search for malicious code correlation. In *ATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)*, 2012.

[20] S. Choi, H. Park, H. I. Lim, and T. Han. A static api birthmark for windows binary executables. *Journal of Systems and Software*, 82(5):862–873, 2009.

[21] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[22] C. Collberg and J. Nagra. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional*, 2010.

[23] Intel Corporation. IA-32 Intel Architectures Software Developer's Manual, volume 2. 2006.

[24] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, pages 33–48, 2011.

[25] P. Deshpande. Metamorphic detection using function call graph analysis. Master's thesis, San Jose State University, 2013.

[26] N. Elita, M. Gavrila, and C. Vertan. Experiments with string similarity measures in the ebmt framework. In *Proc. RANLP*, 2007.

[27] N. Falliere. Sality: Story of a peer-to-peer viral network. *Rapport technique, Symantec Corporation*, 2011.

[28] H. Flake. Structural comparison of executable objects. 2004.

[29] K. S. Han, I. K. Kim, and E. G. Im. Detection methods for malware variant using api call related graphs. In *Proceedings of the International Conference on IT Convergence and Security 2011*, pages 607–611. Springer, 2012.

[30] A. Hanel. Hooked on Mnemonics Worked for Me: Automated Generic Function Naming in IDA. [Online]. Available: http://hooked-on-mnemonics.blogspot.ca/2012/06/automated-generic-function-naming-in.html.

[31] I. Immunity. Immunity debugger. [Online]. Available: http://www.immunityinc.com/products-immdbg.shtml.

[32] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[33] R. K. Jidigam. Metamorphic detection using singular value decomposition. Master's thesis, San Jose State University, 2013.

[34] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security*, pages 637–652, 2013.

[35] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[36] K. Kendall and C. McMillan. Practical malware analysis. In *Black Hat Conference, USA*, 2007.

[37] L. Kessem. Citadel Outgrowing its Zeus Origins. [Online]. Available: https://blogs.rsa.com/citadel-outgrowing-its-zeus-origins/.

[38] J. Kim and B. R. Moon. New malware detection system using metric-based method and hybrid genetic algorithm. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 1527–1528. ACM, 2012.

[39] J. Kornblum. Fuzzy Hashing. [Online]. Available: http://dfrws.org/2006/proceedings/12-Kornblum-pres.pdf.

[40] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[41] T. Lancaster and F. Culwin. A comparison of source code plagiarism detection engines. *Computer Science Education*, 14(2):101–112, 2004.

[42] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 2005 IEEE Workshop on Information Assurance and Seucrity*, pages 64–71. IEEE, 2005.

[43] X. Li, P. K. Loh, and F. Tan. Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 149–154. IEEE, 2011.

[44] M. M. Masud, L. Khan, and B. Thuraisingham. A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers*, 10(1):33–45, 2008.

[45] S. McGhee. Pairwise alignment of metamorphic computer viruses. Master's thesis, San Jose State University, 2007.

[46] J. Milletary. Citadel Trojan Malware Analysis. [Online]. Available: http://botnetlegalnotice.com/citadel/files/Patel_Decl_Ex20.pdf.

[47] P. Mishra. Taxonomy of uniqueness transformations. *The Faculty of the Department of Computer Science*, page 110, 2003.

[48] C. Nachenberg. Computer virus-coevolution. *Communications of the ACM*, 50(1):46–51, 1997.

[49] M. Oberhumer, L. Molnár, and J. F. Reiser. UPX: Ultimate Packer for eXecutables. [Online]. Available: http://upx.sourceforge.net/.

[50] C. Oprisa, G. Cabau, and A. Colesa. From plagiarism to malware detection. In *Proceedings of 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 227–234. IEEE, 2013.

[51] PandaLabs. Annual Report Pandalabs 2013 Summary. [Online]. Available: http://press.pandasecurity.com/wp-content/uploads/2010/05/PandaLabs-Annual-Report_2013.pdf.

[52] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[53] M. Patel. Similarity tests for metamorphic virus detection. Master's thesis, San Jose State University, 2011.

[54] P. Porter. Top Maliciously Used APIs. [Online]. Available: https://www.bnxnet.com/top-maliciously-used-apis/.

[55] P. Porter. The Windows Api for Hackers and Reverse Engineers. [Online]. Available: https://www.bnxnet.com/windows-api-for-hackers/.

[56] S. Priyadarshi. Metamorphic detection via emulation. Master's thesis, San Jose State University, 2011.

[57] B. B. Rad and M. Masrom. Metamorphic virus variants classification using opcode frequency histogram. *arXiv preprint arXiv:1104.3228*, 2011.

[58] B. B. Rad, M. Masrom, and S. Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.

[59] A. Rahimian, R. Ziarati, S. Preda, and M. Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.

[60] J. W. Ratcliff and D. E. Metzener. Pattern-matching-the gestalt approach. *Dr Dobbs Journal*, 13(7):46, 1988.

[61] D. Raygoza. Automated Malware Similarity Analysis. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-09/RAYGOZA/BHUSA09-Raygoza-MalwareSimAnalysis-PAPER.pdf.

[62] B. B. Red, M. Masrom, and S. Ibrahim. Evolution of computer virus concealment and anti-virus techniques: A short survey. *arXiv preprint arXiv:1104.1070*, 2011.

[63] T. Dullienand R. Rolles. Graph-based comparison of executable objects (English version). *SSTIC*, 5:1–3, 2005.

[64] N. Runwal, R. M. Low, and M. Stamp. Opcode graph similarity and metamorphic detection. *Journal of Computer Virology and Hacking Techniques*, 8:37–52, 2012.

[65] N. Runwal, R. M. Low, and M. Stamp. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8(1-2):37–52, 2012.

[66] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of 2001 IEEE Symposium on Security and Privacy (S&P 2001)*, pages 38–49. IEEE, 2001.

[67] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala. Malware detection using assembly and api call sequences. *Journal in computer virology*, 7(2):107–119, 2011.

[68] G. Shanmugam, R. M. Low, and M. Stamp. Simple substitution distance and metamorphic detection. *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, 2013.

[69] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones (IWSC)*, 2009.

[70] X. Snaker, Q. Jibz, and P. BOB. PEiD. [Online]. Available: http://www.aldeid.com/wiki/PEiD.

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

[72] AhnLab ASEC Analysis Team. Malware Analysis: Citadel. [Online]. Available: http://seifreed.es/docs/Citadel%20Trojan%20Report_eng.pdf.

[73] A. H. Toderici and M. Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013.

[74] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proc. BlackHat DC Conf*, 2007.

[75] Wikipedia. Bray-Curtis dissimilarity. [Online]. Available: http://en.wikipedia.org/wiki/Bray%E2%80%93Curtis_dissimilarity.

[76] Wikipedia. Huffman coding. [Online]. Available: http://en.wikipedia.org/wiki/Huffman_coding.

[77] Wikipedia. Interquartile range. [Online]. Available: http://en.wikipedia.org/wiki/Interquartile_range.

[78] Wikipedia. Jaccard index. [Online]. Available: http://en.wikipedia.org/wiki/Jaccard_index.

[79] Wikipedia. Klez. [Online]. Available: http://en.wikipedia.org/wiki/Klez.

[80] Wikipedia. Lempel-Ziv-Welch. [Online]. Available: http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch.

[81] Wikipedia. Levenshtein distance. [Online]. Available: http://en.wikipedia.org/wiki/Levenshtein_distance.

[82] Wikipedia. Longest common subsequence. [Online]. Available: http://en.wikipedia.org/wiki/Longest_common_subsequence_problem.

[83] Wikipedia. Manhattan distance. [Online]. Available: http://en.wikipedia.org/wiki/Manhattan_distance.

[84] Wikipedia. Pearson correlation. [Online]. Available: http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

[85] Wikipedia. Sality. [Online]. Available: http://en.wikipedia.org/wiki/Sality.

[86] Wikipedia. W-shingling. [Online]. Available: http://en.wikipedia.org/wiki/W-shingling.

[87] Wikipedia. Zeus (Trojan horse). [Online]. Available: http://en.wikipedia.org/wiki/Zeus_(Trojan_horse).

[88] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal of Computer Virology and Hacking Techniques*, 2:211–229, 2006.

[89] J. Wyke. What is Zeus? [Online]. Available: http://www.sophos.com/medialibrary/PDFs/technical%20papers/Sophos%20what%20is%20zeus%20tp.pdf.

[90] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren, and N. Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47, 2013.

[91] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.

[92] O. Yuschuk. OllyDbg: 32-bit assembler level analysing debugger. [Online]. Available: http://www.ollydbg.de/.