Practical Detection of Metamorphic Computer Viruses

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

By

Sharmidha Govindaraj

December 2008

Approved by: Department of Computer Science
College of Science
San Jose State University
San Jose, CA

Dr. Mark Stamp

Dr. Robert Chun

Mr. Manikandan Veerachamy, CISCO systems

Acknowledgements

I would like to sincerely thank my advisor Dr. Mark Stamp for his expert guidance, thoughtful insights and bountiful patience without which this research is impossible. I would also like to thank him for introducing me to this interesting world of information security. I would also like to thank my committee members Dr. Robert Chun and Mr. Manikandan for taking time to read through my thesis and providing valuable comments.

Also, I'm grateful to my loving husband Vetri for his endless love and constant encouragement which provided me vast energy throughout my graduate studies. Special thanks to my adorable little one Dharshan for adapting to my tight schedules and sacrificing his play-time with me.

Abstract

Metamorphic virus employs code obfuscation techniques to mutate itself. It absconds from signature-based detection system by modifying internal structure without compromising original functionality. However, it has been proved that machine learning technique like Hidden Markov model (HMM) can detect such viruses with high probability. HMM is a state machine where each state observes the input data with appropriate observation probability. HMM learns statistical properties of “virus features” rather than “signatures” and relies on such statistics to detect same family virus. Each HMM is trained with variants of same family viruses that are generated by same metamorphic engine so that HMM can detect similar viruses with high probability when encountered later on.

Previous HMM-based detection techniques have relied on opcode sequences which are obtained by disassembling the binary (executable) code. Such an approach is impractical, since the disassembly process is slow, and this process must be applied to each file when scanning for viruses. In this paper, we develop a practical HMM-based metamorphic virus detector. We efficiently parses a Windows PE file and generate an approximate opcode sequence which is then used for scoring against the HMM. The results show that our method produce opcode sequences effectively, eliminate time-consuming disassembling phase, reduce training time of HMM by 70% and produce clear separation of scores between family virus and non-members.

Table of Contents

List of Tables and Figures

Tables

Figures

Figures in Appendices

1. Introduction

In today's electronically connected digital world, data is stored in the connected storages and shared globally. Modern technology has changed the way we learn, work, play, and live but it does not offer luxury of high availability and accessibility without endangering the security and privacy of information. No matter how secure data is stored and accessed, information still get stolen. Everyday and every second, somebody in the world has his/her identity and money seized. Even worse, information which is worth lots of time, energy, and resources is completely wiped out by malicious programs causing huge loss. As we all understand, the modern digital world poses multifaceted vulnerabilities, a major concern is to protect data from being corrupted or destroyed by malicious codes.

Malicious code is "any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function" (Jordan, 2002). Malicious code can be classified as virus, worm, Trojan, backdoors, and so on. Although all malicious codes are commonly called virus, each of the above term mean a type of attack the malicious code perform. For our purpose, we refer the commonly used term 'virus' to address all malicious codes in discussion for this project. Computer viruses have been consistently evolving. Each new generation of viruses poses new challenges for antivirus developers. Fortunately, antivirus developers do rise to the challenges and devises a method to protect data from viruses as they show up.

Our research focuses on a specific type of virus called metamorphic virus that uses obfuscating techniques to mutate itself. We will discuss further about detecting metamorphic viruses and enhancing one of the detection technique called Hidden Markov Models (HMM).

The organization of this report is as follows; Section 2 covers viruses and their types with an emphasis on metamorphic virus (with examples); Section 3 covers available detection techniques with an emphasis on HMM; Section 4 covers our research on detecting metamorphic viruses more practically and efficiently using HMM; Section 5 covers the training and testing of HMM; and finally, Section 6 covers the discussion of results.

Figure 1 shows number of new malicious threats every year.

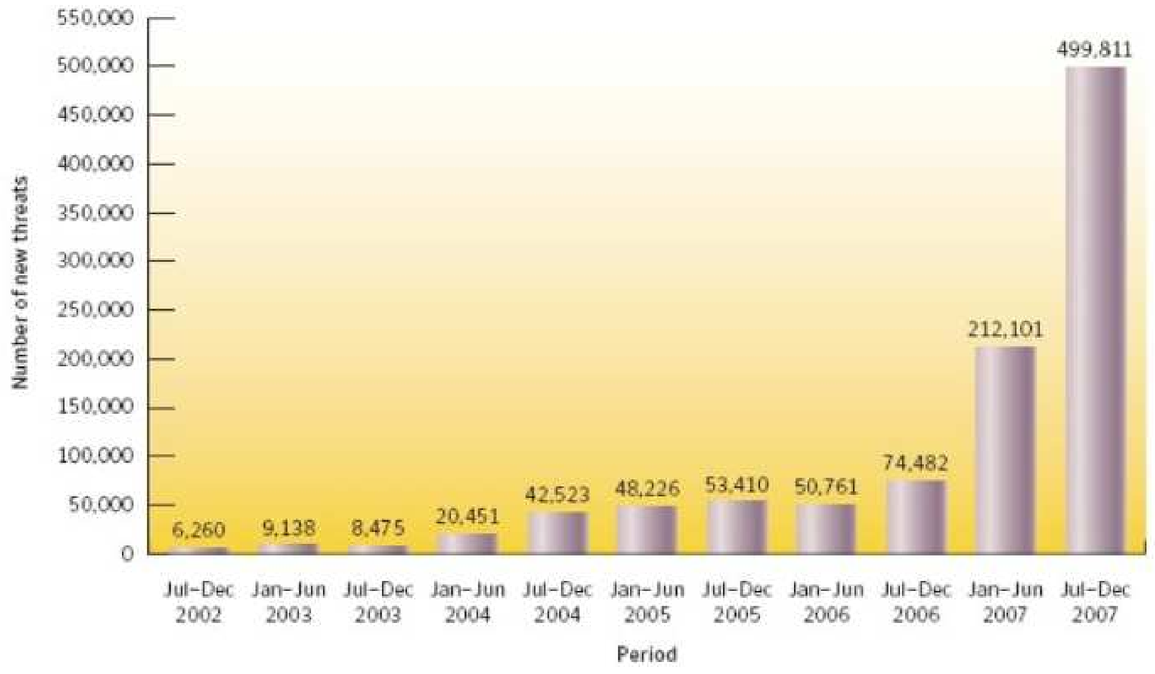


Figure 1. New Malicious Code threats
Source: Turner et al., 2008

2. Viruses and their types

Viruses are malicious programs that have threatened the world of computers for about thirty years; and will be more challenging than ever before. As the modern viruses present new challenges, the antivirus community is constantly putting efforts to understand, learn, and develop new antivirus kits to detect and remove viruses. There are many types of viruses with different risk levels we have discovered.

Some of well known viruses are

- Boot sector virus
- Polymorphic virus
- Macro virus
- Metamorphic virus

The following section explains in-depth details of metamorphic virus followed by brief introduction to other viruses.

2.1 Metamorphic Virus

Metamorphic viruses are viruses that mutate itself with the use of metamorphic engine that come along with virus code. These viruses are a new generation of viruses that escape signature detection techniques, as the shape of virus body is changed every time when it infects. To explain in short, metamorphic viruses mutate its body and change the internal structure preserving the functionality of virus.

Such metamorphism is employed by obfuscating the code using different techniques. Five of the techniques are listed below (Mohammed, 2003).

- Simple Substitution
- Instruction Reordering
- Dead code Insertion
- Register usage exchange
- Reordering subroutines

Figure 2 shows different shapes of virus body in each mutation (Szor, 2002).

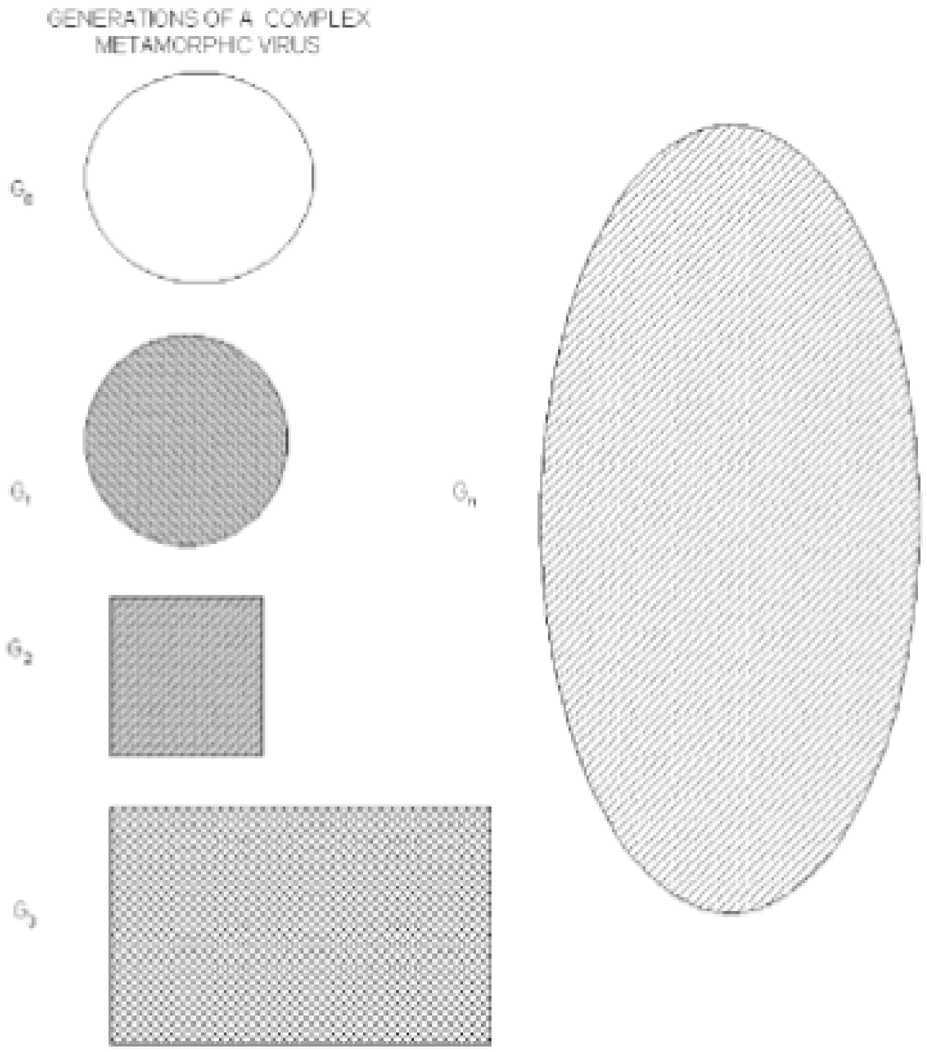
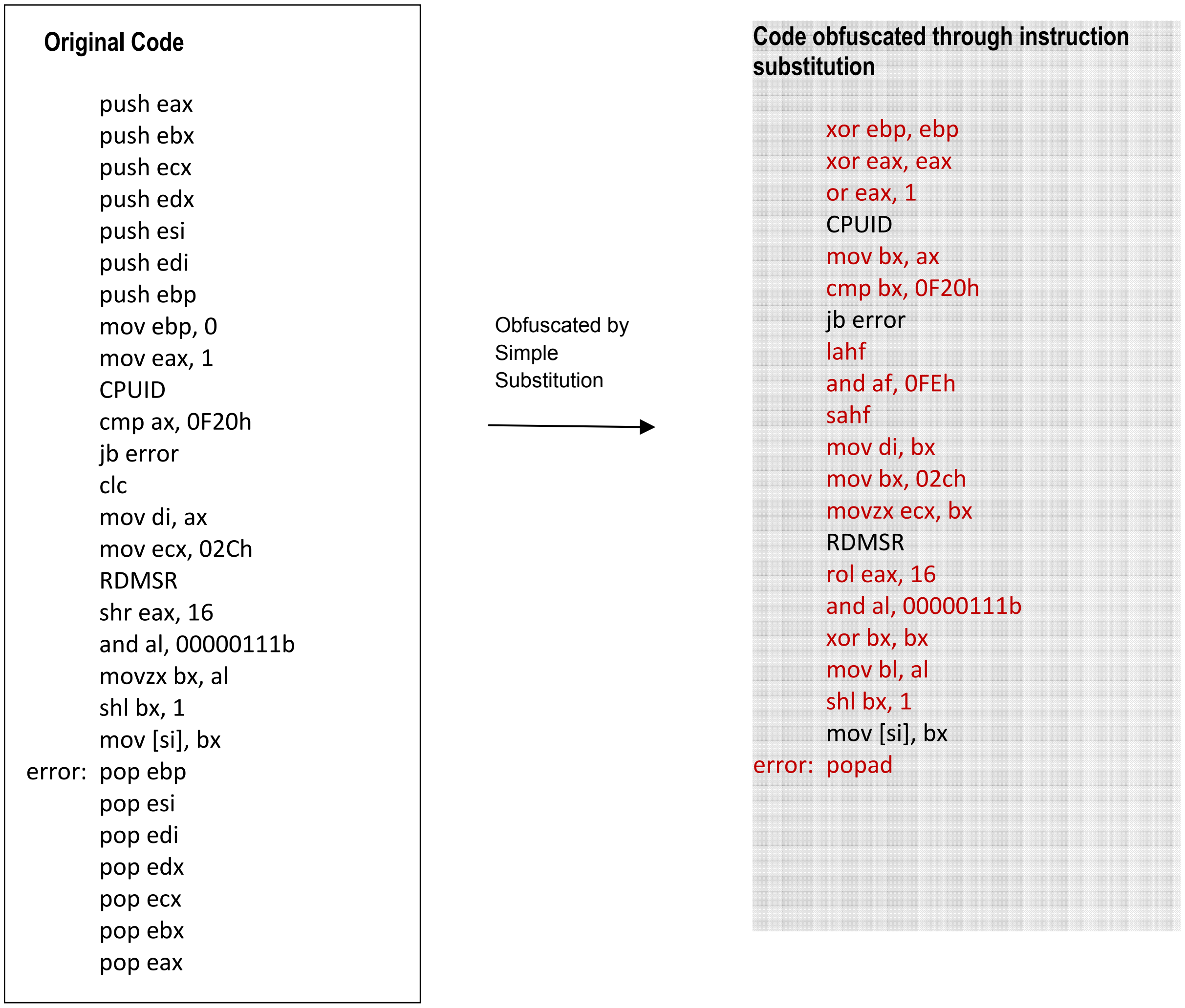


Figure 2. Different forms of a metamorphic virus
Source: Szor, 2001

2.1.1 Simple Substitution

This technique allows for the substitution of an instruction or a block of code with an equivalent instruction or a block of code. To accomplish this technique, the metamorphic generator must maintain a dictionary of instructions and their equivalents. An example showing how substitution is done is illustrated in figure 3 below.



Original Code

```
        push eax
        push ebx
        push ecx
        push edx
        push esi
        push edi
        push ebp
        mov ebp, 0
        mov eax, 1
        CPUID
        cmp ax, 0F20h
        jb error
        clc
        mov di, ax
        mov ecx, 02Ch
        RDMSR
        shr eax, 16
        and al, 00000111b
        movzx bx, al
        shl bx, 1
        mov [si], bx
error:  pop ebp
        pop esi
        pop edi
        pop edx
        pop ecx
        pop ebx
        pop eax
```

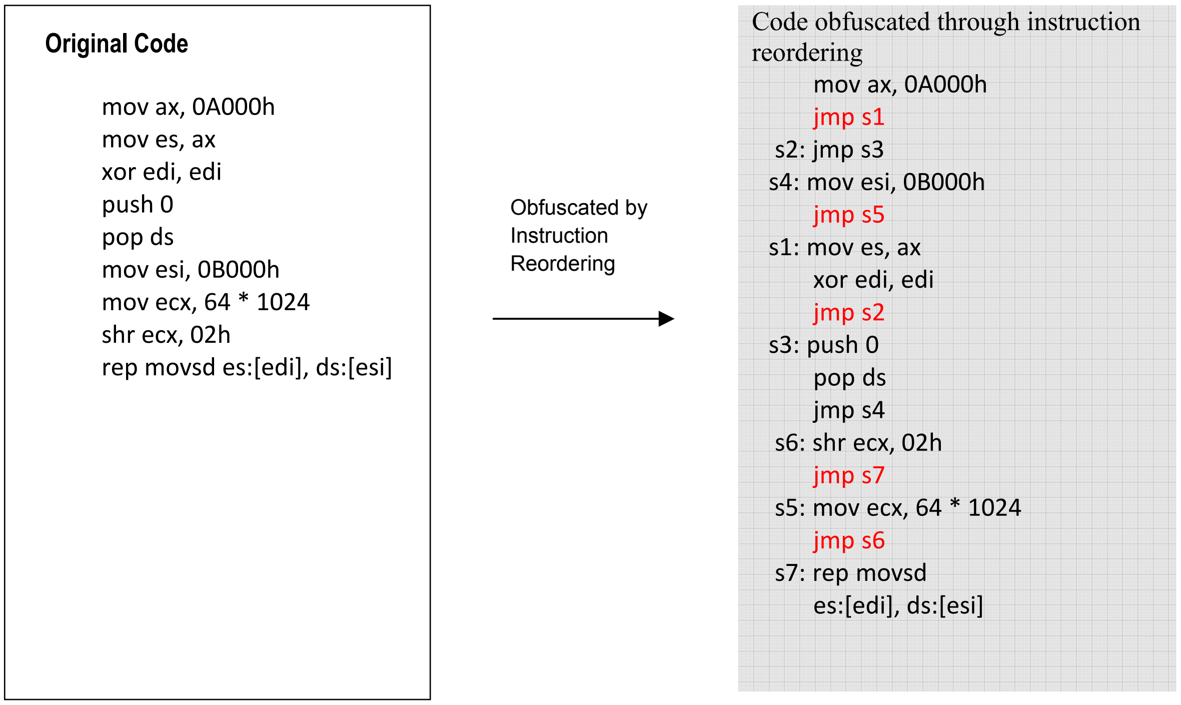
Code obfuscated through instruction substitution

```
        xor ebp, ebp
        xor eax, eax
        or eax, 1
        CPUID
        mov bx, ax
        cmp bx, 0F20h
        jb error
        lahf
        and af, 0FEh
        sahf
        mov di, bx
        mov bx, 02ch
        movzx ecx, bx
        RDMSR
        rol eax, 16
        and al, 00000111b
        xor bx, bx
        mov bl, al
        shl bx, 1
        mov [si], bx
error:  popad
```

Figure 3. Simple Substitution
Source: Author's Research

2.1.2 Instruction Reordering

Reordering instructions and inserting unconditional branches or jumps using GOTO statements is one way of metamorphism employed in virus body. Reordering can also be done by reordering the independent instructions in the same way compilers do. Figure 4 below illustrates an example of instruction reordering.



Original Code

```
mov ax, 0A000h
mov es, ax
xor edi, edi
push 0
pop ds
mov esi, 0B000h
mov ecx, 64 * 1024
shr ecx, 02h
rep movsd es:[edi], ds:[esi]
```

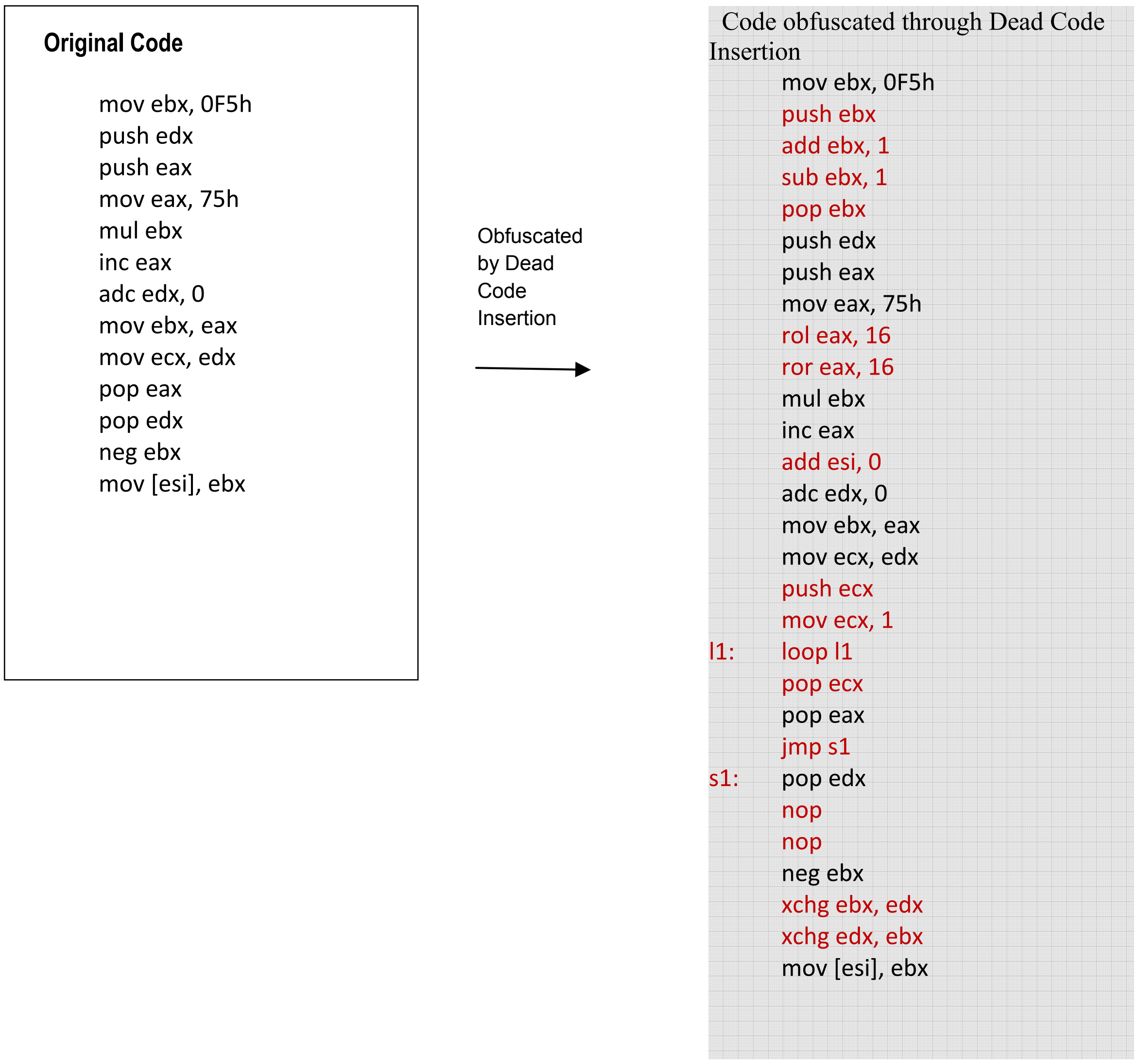
Code obfuscated through instruction reordering

```
        mov ax, 0A000h
        jmp s1
    s2: jmp s3
    s4: mov esi, 0B000h
        jmp s5
    s1: mov es, ax
        xor edi, edi
        jmp s2
    s3: push 0
        pop ds
        jmp s4
    s6: shr ecx, 02h
        jmp s7
    s5: mov ecx, 64 * 1024
        jmp s6
    s7: rep movsd
        es:[edi], ds:[esi]
```

Figure 4. Instruction reordering
Source: Author's Research

2.1.3 Dead Code Insertion

This technique inserts do-nothing or garbage instructions like NOP inside the virus body without altering original functionality. This is one of the easiest techniques to obfuscate the code section and the easiest to detect as the actual virus code is not rearranged. Dead code insertion is illustrated in Figure 5.



Original Code

```
mov ebx, 0F5h
push edx
push eax
mov eax, 75h
mul ebx
inc eax
adc edx, 0
mov ebx, eax
mov ecx, edx
pop eax
pop edx
neg ebx
mov [esi], ebx
```

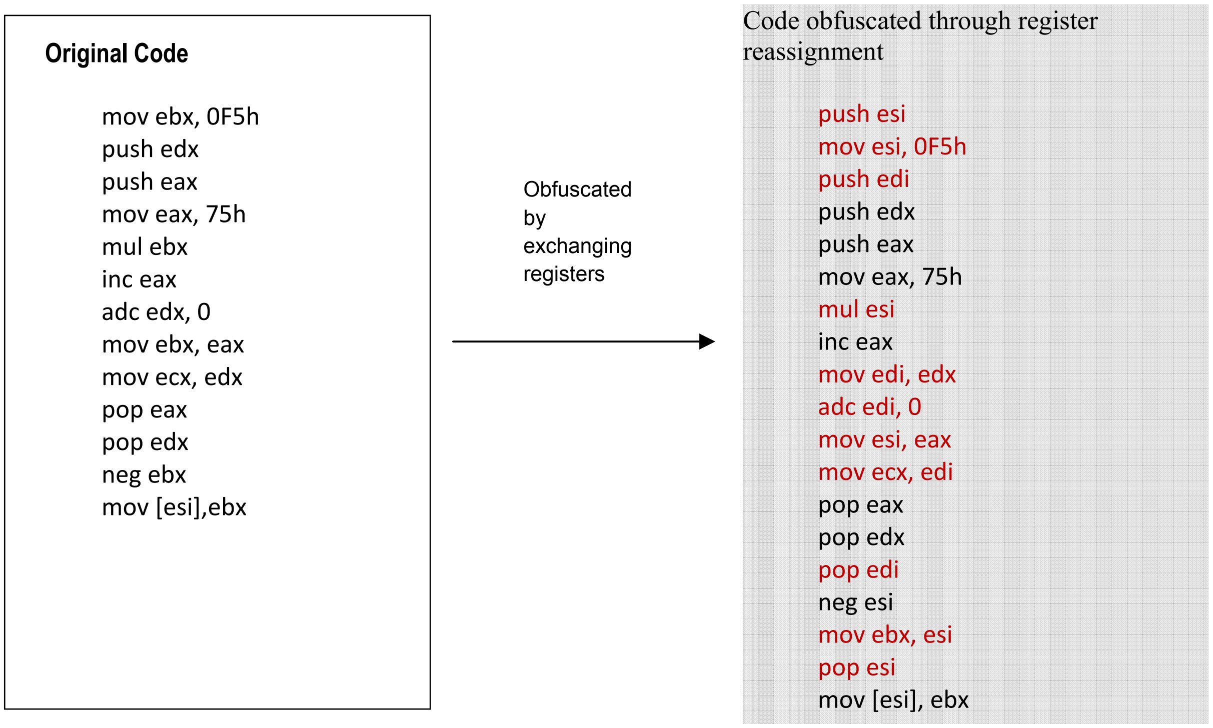
Code obfuscated through Dead Code Insertion

```
        mov ebx, 0F5h
        push ebx
        add ebx, 1
        sub ebx, 1
        pop ebx
        push edx
        push eax
        mov eax, 75h
        rol eax, 16
        ror eax, 16
        mul ebx
        inc eax
        add esi, 0
        adc edx, 0
        mov ebx, eax
        mov ecx, edx
        push ecx
        mov ecx, 1
l1:     loop l1
        pop ecx
        pop eax
        jmp s1
s1:     pop edx
        nop
        nop
        neg ebx
        xchg ebx, edx
        xchg edx, ebx
        mov [esi], ebx
```

Figure 5. Dead code insertion
Source: Author's Research

2.1.4 Register Usage exchange

Register Usage Exchange is a technique which involves changing usage of registers in the code without modifying the flow of code. This technique often requires adding more instructions for resetting or restoring the state of the registers. It seems to be more complex compared to other techniques as it requires knowledge of processor registers and supported instruction sets along with the ability to parse the binary code section and identify the register usage. Figure 6 below illustrates register usage exchange.



Original Code

```
mov ebx, 0F5h
push edx
push eax
mov eax, 75h
mul ebx
inc eax
adc edx, 0
mov ebx, eax
mov ecx, edx
pop eax
pop edx
neg ebx
mov [esi],ebx
```

Code obfuscated through register reassignment

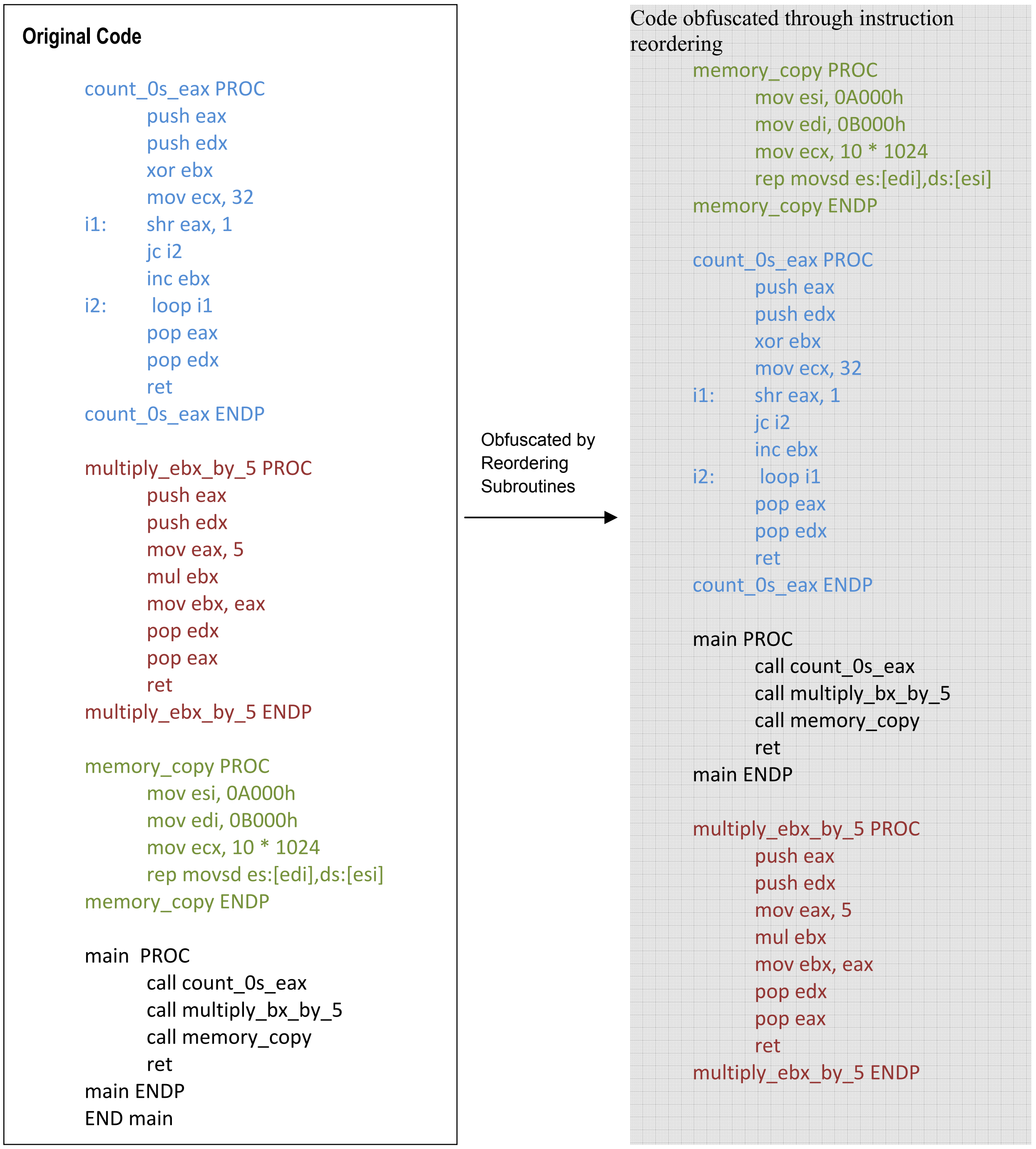
```
push esi
mov esi, 0F5h
push edi
push edx
push eax
mov eax, 75h
mul esi
inc eax
mov edi, edx
adc edi, 0
mov esi, eax
mov ecx, edi
pop eax
pop edx
pop edi
neg esi
mov ebx, esi
pop esi
mov [esi], ebx
```

Note: EBX replaced with ESI and EDX replaced with EDI

Figure 6. Register usage exchange
Source: Author's Research

2.1.5 Reordering Subroutines

Obviously, using this technique, metamorphic engine reorders subroutines and thus changes the structure of the code. The above technique is another simple technique to obfuscate the shape of the virus. Reordering Subroutines is illustrated in figure 7 below.



Original Code

```
count_0s_eax PROC
        push eax
        push edx
        xor ebx
        mov ecx, 32
i1:     shr eax, 1
        jc i2
        inc ebx
i2:      loop i1
        pop eax
        pop edx
        ret
count_0s_eax ENDP

multiply_ebx_by_5 PROC
        push eax
        push edx
        mov eax, 5
        mul ebx
        mov ebx, eax
        pop edx
        pop eax
        ret
multiply_ebx_by_5 ENDP

memory_copy PROC
        mov esi, 0A000h
        mov edi, 0B000h
        mov ecx, 10 * 1024
        rep movsd es:[edi],ds:[esi]
memory_copy ENDP

main  PROC
        call count_0s_eax
        call multiply_bx_by_5
        call memory_copy
        ret
main ENDP
END main
```

Code obfuscated through instruction reordering

```
memory_copy PROC
        mov esi, 0A000h
        mov edi, 0B000h
        mov ecx, 10 * 1024
        rep movsd es:[edi],ds:[esi]
memory_copy ENDP

count_0s_eax PROC
        push eax
        push edx
        xor ebx
        mov ecx, 32
i1:     shr eax, 1
        jc i2
        inc ebx
i2:      loop i1
        pop eax
        pop edx
        ret
count_0s_eax ENDP

main PROC
        call count_0s_eax
        call multiply_bx_by_5
        call memory_copy
        ret
main ENDP

multiply_ebx_by_5 PROC
        push eax
        push edx
        mov eax, 5
        mul ebx
        mov ebx, eax
        pop edx
        pop eax
        ret
multiply_ebx_by_5 ENDP
```

Figure 7. Reordering subroutines
Source: Author’s Research

2.2 Other viruses

One of the oldest and popular viruses from the late 1980s is boot sector virus. It replaces Master Boot Record (MBR) or boot sector in the hard drive with its own code. The boot sector is a drive sector where the Operating System (OS) boot loader lives. The Basic Input/Output System (BIOS) transfers control to the boot sector at the end of Power-On Self-Test (POST) to hand off control to the OS while booting. Infecting the boot sector enables the boot virus to gain the ability to take over the control whenever the system boots, stay hidden in memory during runtime, and perform its malicious activities.

One of the other popular and challenging viruses is polymorphic virus. It uses encryption to get away from antivirus software that only uses simple signature detection technique to detect viruses. Each polymorphic virus incorporates a decryptor at the top of an execution flow so that the virus can decrypt the encrypted part of the code at first and hand off the control to decrypted virus. As a polymorphic virus usually embeds the decryptor at the beginning of the code section, it enables anti-virus scanners to look for decryptor byte patterns at the beginning of a code section and detect the virus easily.

A Macro virus is a type of virus that mainly infects documents that are normally not executable. It is written in a macro language that is supported by word processors and email applications; this provides mechanism to embed macro programs within documents and execute it whenever the document is opened. Modern Antivirus software has the capability to detect such macro viruses.

3. Metamorphic virus and their detection techniques

As metamorphic viruses employ complicated techniques, many different methods have been developed to detect metamorphic viruses. Each detection method has its own pros and cons. Some of the detection techniques described in Symantec's white paper (Szor, 2001) are highlighted below.

Geometric Detection technique relies on “shape heuristic”; this allows to find whether a file is infected, or not, by learning the file structure of the virus and looking for learnt structures in the infected files. Often, this technique is prone to false positives as it simply learns the layout of the virus and does not learn about the virus at the instruction level.

Code emulation is employed by creating a virtual machine which emulates the underlying hardware including processor, memory, and peripherals and runs an operating system. This technique detects viruses by running suspicious files on its guest virtual machine and looks for any malicious activities and patterns. The above technique has the ability to detect complicated viruses but it needs considerable system resources to create a virtual machine.

The last and most successful technique is the Machine learning technique. This technique uses the concept of data mining, neural networks, and HMM to learn the structure of the virus at the instruction level. Though, data mining techniques produce more false positives, neural networks and HMM have a very low rate of false positives.

As our research is focused on using HMM for metamorphic virus detection, HMM will be discussed in detail in the following section.

3.1 Hidden Markov Models

The Hidden Markov Model is a state machine with a finite set of states, each of which is associated with a probability distribution for certain observation symbols. This model is called “Hidden” Markov Model because the external observer can only see the outcome or the observation, and the state remains hidden. Transition between states is associated with transition probability and an outcome, or observation is associated with observation probability. HMMs are statistical learning techniques by which we can train the model for particular observation sequence (opcode sequence from a program). After training a HMM with a set of opcode sequence, the model gains the ability to detect similar opcode sequence in a given input.

The notations used in HMM are listed below.

T = the length of the observation sequence

N = the number of states in the model

M = the number of observation symbols

Q = the states of the Markov process {q0, q1, . . . , qN−1

V=set of possible observations {0, 1, . . . ,M −1}

A = the state transition probabilities matrix

B = the observation probability matrix

π = the initial state distribution matrix

O = (O0, O1, . . . ,OT−1) = observation sequence.

λ= (A, B, π) is a HMM model

Figure 8 below shows the Hidden Markov Model state transition where X is a hidden state and O is observation sequence which an observer can see.

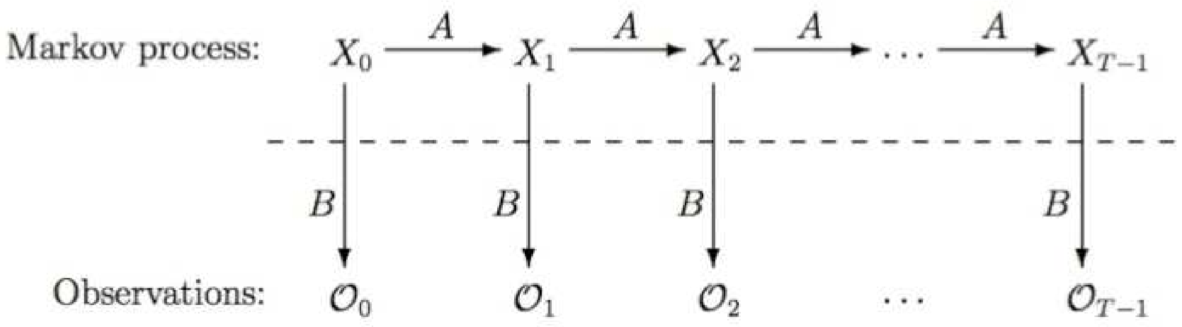


Figure 8. Hidden Markov Model
Source: Stamp, M., 2004

4. Metamorphic detection with HMM

Initially, HMM is trained with variants of same family viruses (viruses generated with same virus generation kit) during which HMM create a model for each family viruses. Once training is completed, HMM use that model to detect whether a given file belongs to particular family, or not. Before the training phase, a number of steps should be carried out. Let us examine the steps involved in Wong and Stamp's (2006) work; first, different viruses are generated using virus generators; second, the generated viruses are assembled using TASM 5.0 to create executables; and finally, the executables are disassembled back into assembly code using IDA Pro. The above steps are illustrated in Figure 9 below.

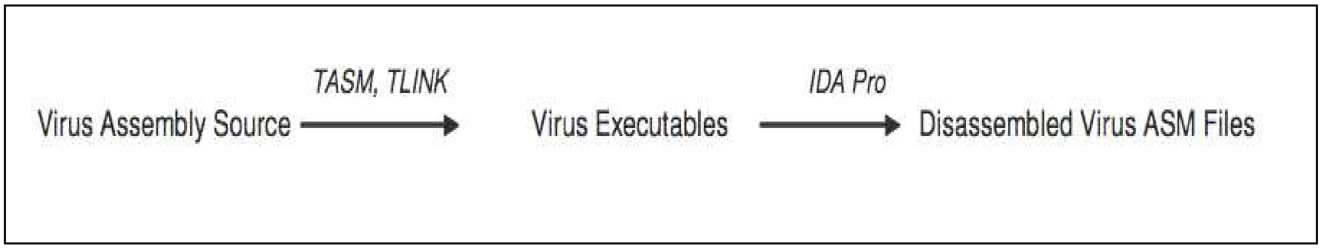


Figure 9. Preprocessing of virus files
Source: Wong and Stamp (2006)

Once disassembled, they extracted assembly opcode sequences from disassembled ASM files and concatenated all the opcode sequences to form a single long sequence. Finally, HMM was trained with the single concatenated sequence. Large collections of metamorphic viruses generated by virus generator kits are grouped into different data sets with each data set containing viruses generated by same Virus generation kit. Five fold cross validation is applied to a data set and further subdivided into five subsets: four being training sets and one being a test set; each time, a different train set and test set is used. Training set viruses are used for HMM training and test set viruses are used to test, or evaluate the performance of HMM in finding the same family virus.

5. Efficient preprocessing of metamorphic virus executables

As explained in section 4, Wong and Stamp used IDA pro, a disassembler, to disassemble the executables before extracting the opcode sequence for the training set. This disassembling step is time-consuming, inefficient, and impractical when it involves large numbers of virus files. An alternative method is to extract the opcode sequences directly from executables and use the resultant sequence to train HMM.

Extracting opcode sequences programmatically from binary executables with no manual work involved is very complicated, as the binary file is raw and, in most cases, data is embedded within the code section. This research is focused on simplifying and completely removing the manual work involved in the process of creating opcode sequence and improving the efficiency of overall preprocessing. In our method, we followed three consecutive steps to preprocess a virus file. The steps involved in the method of preprocessing under discussion as are follows;

1. Extracting Code section: An executable may include a number of sections such as code, data, and stack. As virus codes mostly lives only in code section, we need to extract the code section from the executable file discarding other sections. Though there are a lot of executable formats currently in use, we have taken only Portable Executable (PE) format and DOS executable format into consideration as these formats are most-used popular formats.
2. Create opcode sequence: Analyze each virus file individually and determine Most Frequently Occurred (MFO) mnemonics. Find out all possible opcodes for MFO mnemonics and create a lookup table of MFO opcodes. The opcode sequence is created directly from the executable files by scanning byte by byte and checking if it falls into MFO opcodes by looking into the MFO opcode table.
3. Concatenate opcode sequence: Finally, opcode sequences are divided into data set and train set. All opcode sequences of data set are concatenated to

form a single observation sequence. This observation sequence is used as train set for HMM.

5.1 Extraction of code segments from Virus executables

5.1.1 PE executable format

PE executable (PE) format is a "file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems" (Wikipedia). I have focused on PE executables as it is the most used and most vulnerable format being the standard of windows OS. Before dealing with extraction of code segment from PE executables, it is essential to discuss bits and pieces of PE file format to have a good idea of PE executable. The subsequent sections describe PE format in detail and how to extract code section from PE format compliant file. The format of a PE file is shown figure 10 (Page 16).

MS DOS header

A PE file always starts with a MS DOS header that can be identified by a two-byte signature represented in ASCII as "MZ" or in hex as "0x5A4D". Though MS-DOS header is comprised of many fields, e_magic and e_lfanew are the fields we are interested in. e_magic field contains the signature of MS DOS header and e_lfanew contains Relative Virtual Address (RVA) to PE header. It also includes a checksum file that can be used to check the integrity of the header.

A MSDOS stub program is included in windows 32 and 64 bit format to display a message "This program cannot be run in DOS mode" when PE executables are run under MSDOS environment.

This header was embedded in PE executables to provide backward compatibility when the industry was transitioning from DOS operating system to Windows operating system.

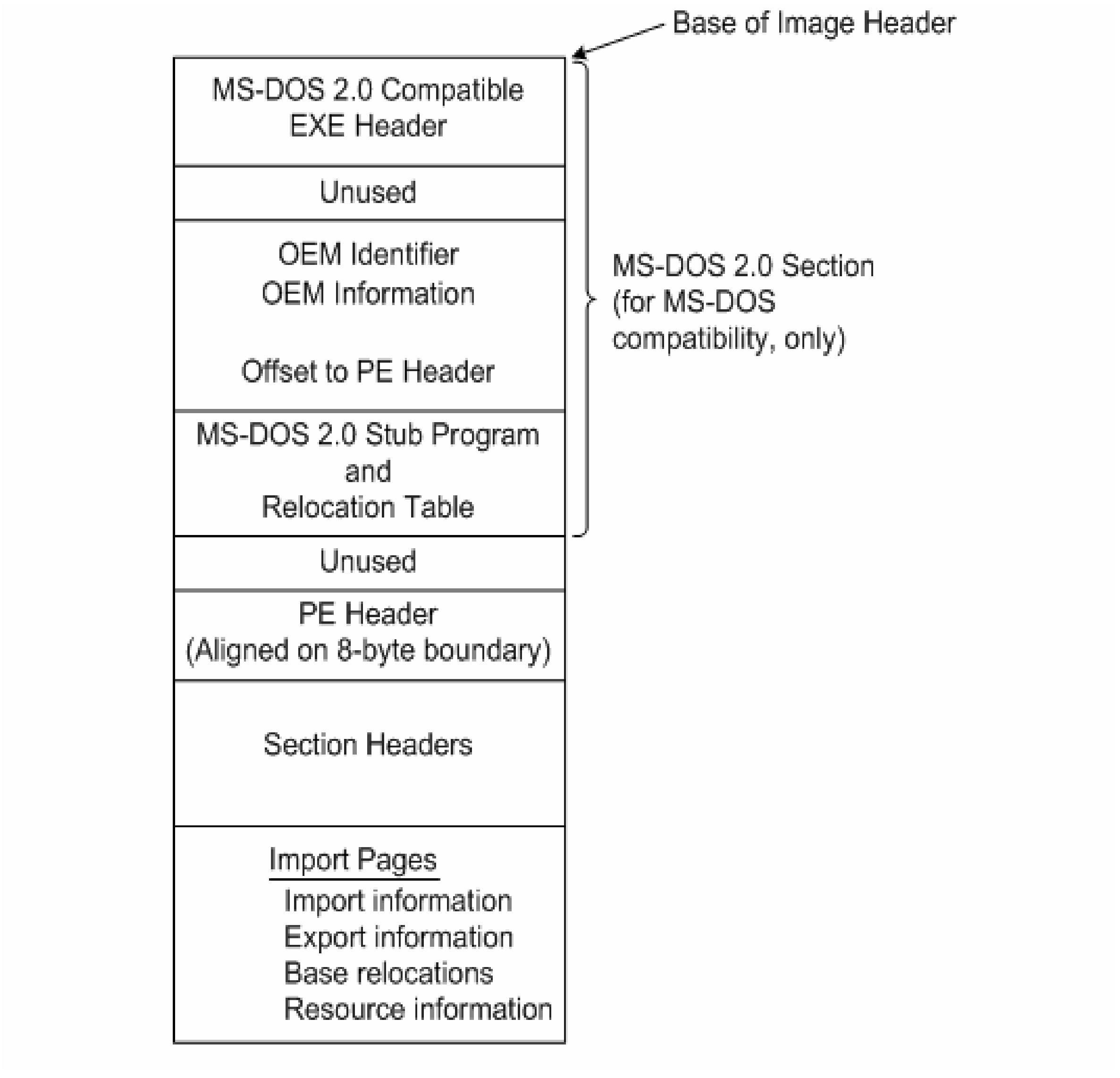
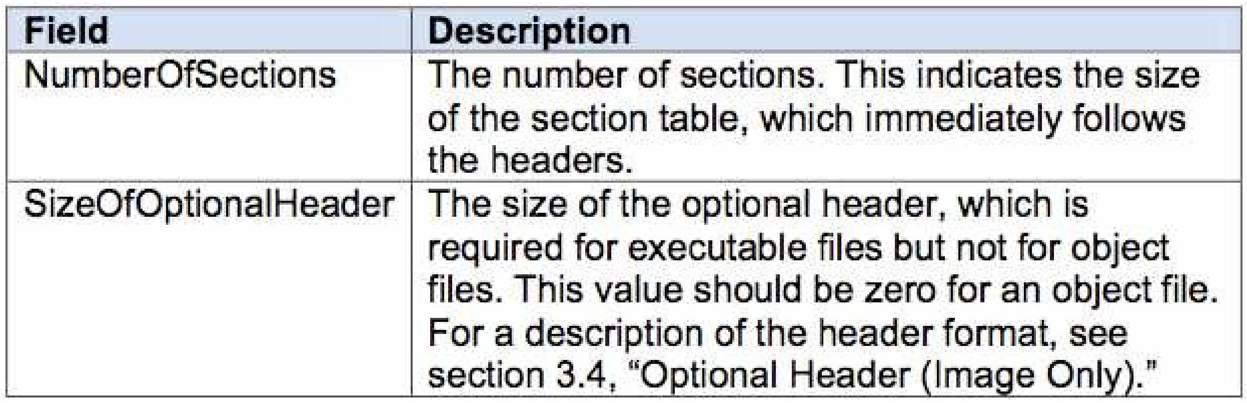


Figure 10. PE executable format Layout
Source: Microsoft PE specification, 2008

PE header

The MS-DOS header is followed by PE header that contains a PE signature File header and Optional Header. The PE signature is used to identify the PE header in a PE file which is represented by a 4-byte value in ASCII as “PE” or in hex as “0x00004550” Among the many fields file header contains, we are interested in two important fields. Those fields and their usages are explained in Table 1.

Table 1. PE file Header



Field	Description
NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see section 3.4, “Optional Header (Image Only).”

Source: Microsoft PE Specification, 2008

Since optional header is not required for our purpose, the field SizeOfOptionalHeader is used to skip the optional header.

Section Header

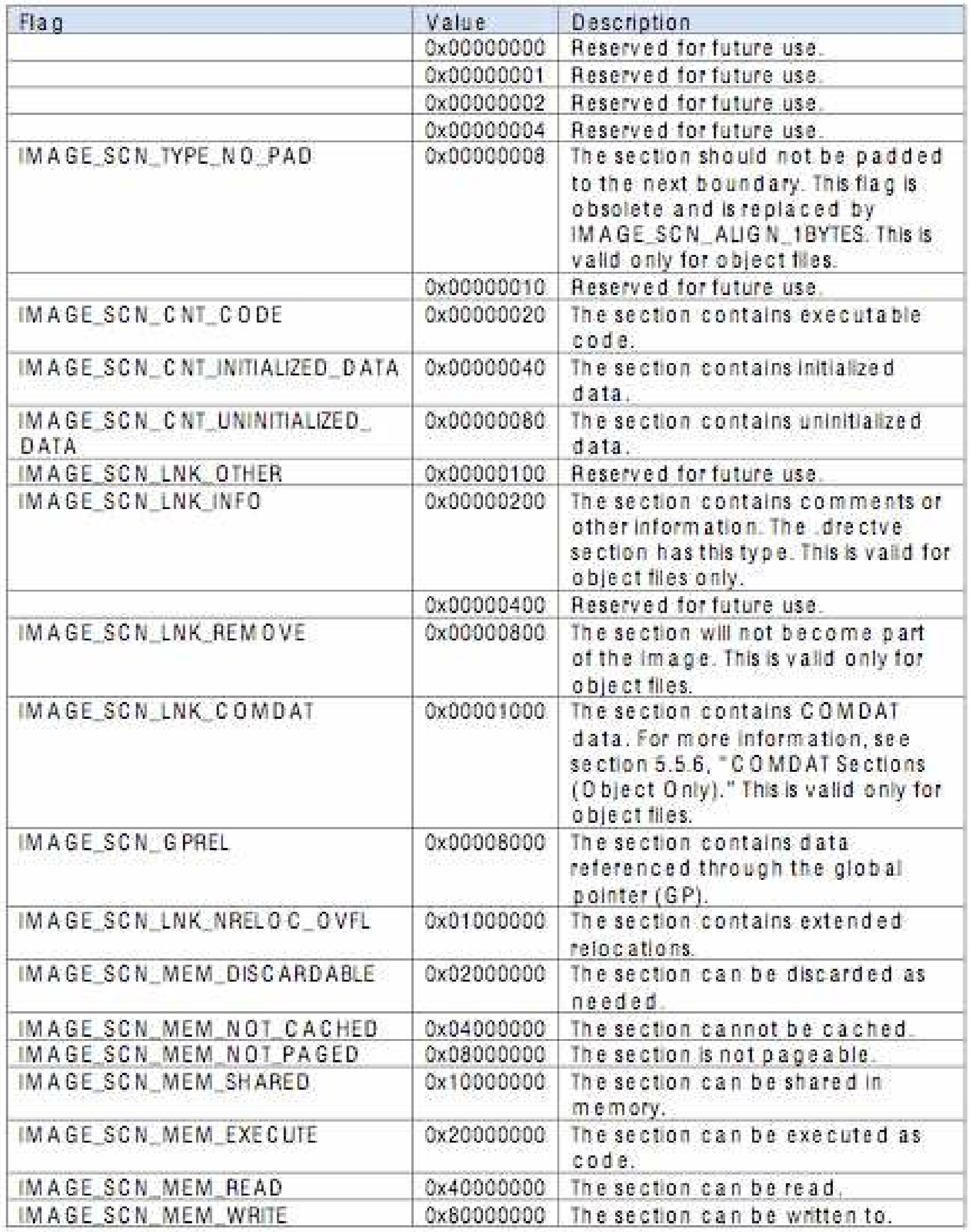
Followed by the optional header is a section header that contains information about different sections of the file. Table 2 shows all the fields in section header. Section header is an array of structures where there is a structure for each section containing all the fields as shown in table 2 (Page 18). The name field and characteristics field are required to find the code section. The pointertorawdata and sizeofrawdata fields are used to locate and extract the code section. Table 3 (Page 20) shows section header characteristic flags.

Table 2. PE Section Header Fields

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory.
16	4	SizeOfRawData	The size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header
24	4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.
28	4	PointerToLinenumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
32	2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
34	2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
36	4	Characteristics	The flags that describe the characteristics of the section.

Source: Microsoft PE Specification, 2008

Table 3. PE Section Header Characteristic Flags



Flag	Value	Description
	0x00000000	Reserved for future use.
	0x00000001	Reserved for future use.
	0x00000002	Reserved for future use.
	0x00000004	Reserved for future use.
IMAGE_SCN_TYPE_NO_PAD	0x00000008	The section should not be padded to the next boundary. This flag is obsolete and is replaced by IMAGE_SCN_ALIGN_1BYTES. This is valid only for object files.
	0x00000010	Reserved for future use.
IMAGE_SCN_CNT_CODE	0x00000020	The section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	The section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	The section contains uninitialized data.
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved for future use.
IMAGE_SCN_LNK_INFO	0x00000200	The section contains comments or other information. The .drectve section has this type. This is valid for object files only.
	0x00000400	Reserved for future use.
IMAGE_SCN_LNK_REMOVE	0x00000800	The section will not become part of the image. This is valid only for object files.
IMAGE_SCN_LNK_COMDAT	0x00001000	The section contains COMDAT data. For more information, see section 5.5.6, "COMDAT Sections (Object Only)." This is valid only for object files.
IMAGE_SCN_GPREL	0x00008000	The section contains data referenced through the global pointer (GP).
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	The section contains extended relocations.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	The section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	The section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	The section is not pageable.
IMAGE_SCN_MEM_SHARED	0x10000000	The section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE	0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	The section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	The section can be written to.

Source: Microsoft PE Specification, 2008

Figure 11 shows the layout of PE executable in more detail with signatures, partitioned file, and optional header and pointers from section header entry to appropriate sections.

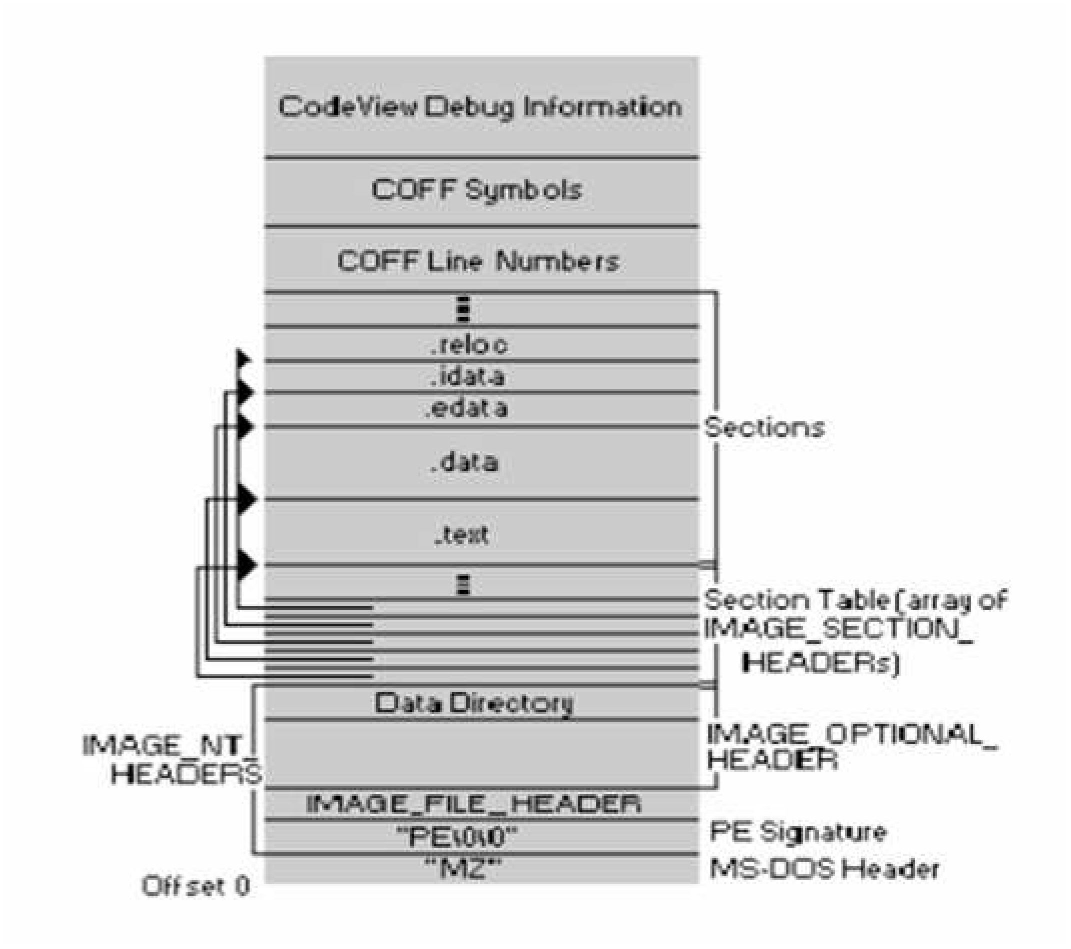


Figure 11 Detailed Layout of PE executable
Source: Patriek, 2002

5.1.2 PE Code Segment Extraction

This section explains how our program extracts code segment from PE executables with reference to actual codes. Figure 12 (page 22) demonstrates a high level execution flow of code segment extraction.

First, the DOS header is read and the e_magic field is checked for MS DOS signature "MZ" or "0x5A4D". If the signature is valid, then we have to jump to PE header using the address in e_lfanew field. Once the PE header is read from the file, the signature field is checked for PE signature “PE” or “0x00004550”. If the signature is valid, the file being processed is confirmed as a PE executable file. Once PE header is located and validated, the SizeOfOptionalHeader field is used to skip the optional header since optional header is not required for our purpose. Now we have reached the section header.

The section header is an array of structures where there is a structure for each section in the file. So, to find the section header for code section, we have to compare the name field to “.text” or characteristics field to “0x0000020”. As the name field is not standardized, it is not named always “.text” and so we are checking for characteristics field too. According to characteristics flags, “0x0000020” mean that the section contains executable code. So, once the code section header is located, the field called PointerToRawData is used to locate the code section, and the field called SizeOfRawData is used to extract the code section.

After completing the code segment extraction, the program is tested with different input exe files. All the tested files differ in size or number of code sections. Further testing is conducted by using HexEdit and PEdump utilities, a dumping utility for executables. The same exe files, which were used for testing our program is given as input to both of these utilities. The output of our program is binary compared with utility outputs. Comparison showed that our code worked flawlessly and extracted code segments exactly.

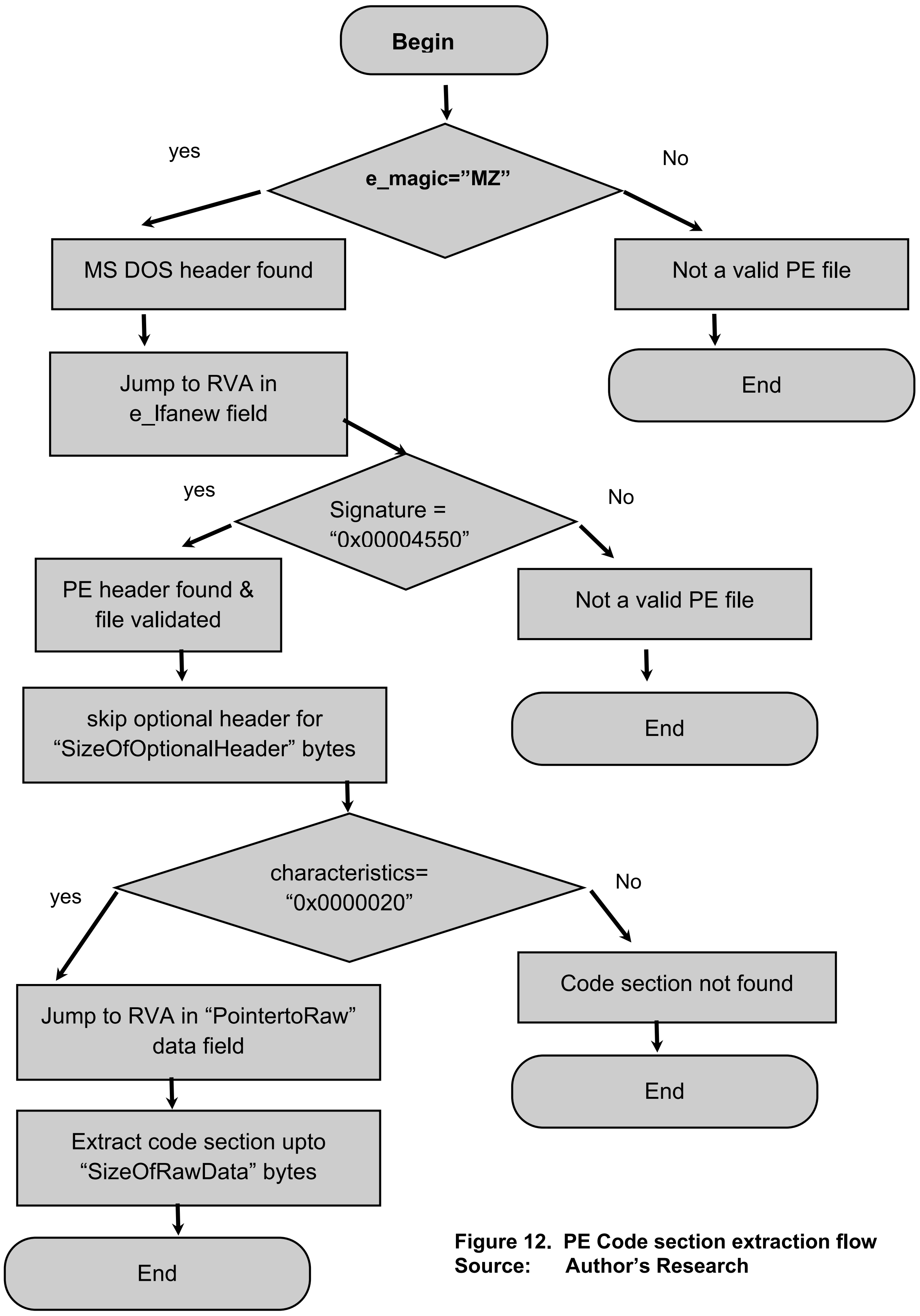

Begin
e_magic="MZ"
yes
No
MS DOS header found
Not a valid PE file
Jump to RVA in e_lfanew field
End
Signature = "0x00004550"
yes
No
PE header found & file validated
Not a valid PE file
skip optional header for "SizeOfOptionalHeader" bytes
End
characteristics= "0x0000020"
yes
No
Jump to RVA in "PointertoRaw" data field
Code section not found
Extract code section upto "SizeOfRawData" bytes
End
End

Figure 12. PE Code section extraction flow
Source: Author's Research

5.1.3 DOS executable Format

Although DOS executables seem to be outdated, many early viruses, like G2 and MPCGEN, are yet in the DOS executable form. MS DOS header in DOS executables is exactly the same as in PE executables. For our purpose, we are required to read following fields: e_magic, e_cblp, e_cp and e_ip. e_magic field contains the signature represented in ASCII as “MZ” or in hex as “0x54AD”. This field is validated to check whether the given file is a valid DOS file. If DOS executable signature is found, e_cp, e_ip and e_cblp field are read from the header. e_ip field specify the offset where code segment starts. e_cp field specify number of pages in the file where each page is 512 bytes long. e_cblp field specify number of bytes used in the last page of code segment.

Once we get the values of all the above fields, size of the code segment is calculated as follows,

$$\text{Size of code segment} = e\_cp*512 - (512 - e\_cblp)$$

Once size of code segment is calculated, extract the code section starting from the offset pointed by e_ip.

5.2 Preprocessing of Code Segment and Opcode extraction

As discussed earlier, Wong and Stamp used IDApro, a disassembler, to create disassembled ASM files and extract assembly opcode sequences from executable files. One of the goals for this project is to eliminate time-consuming and inefficient disassembling process.

With the code section of virus executables in hand, we started researching for methods which doesn’t go through disassembling to extract assembly opcodes like MOV, ADD and so on. We found two obvious alternatives. First method is Most Frequently Occurred (MFO) opcode searching method which looks for the MFO opcodes in the binary executable and creates the opcode sequence of MFO opcodes. Second method is adding a part of disassembling code which disassembles on the fly with no manual intervention and extracting only the opcodes leaving behind operands. Of these two alternatives, we selected the former approach because latter involves disassembling and our major goal is to skip disassembling.

5.2.1 Intel x86 Instruction Set

A brief introduction to Intel x86 instruction set is required to understand low level details of assembly instruction. Figure 13 shows Intel instruction format.

Each instruction consists of instruction prefixes, instruction opcode bytes, MOD value, address displacement value and an immediate data. The format of an Intel x86 processor architecture based instruction is shown in the figure 13 below. The assembly language commands corresponding to opcodes are called mnemonics. For example, the assembly language command ADD is a mnemonic corresponding to the opcode 0x80.

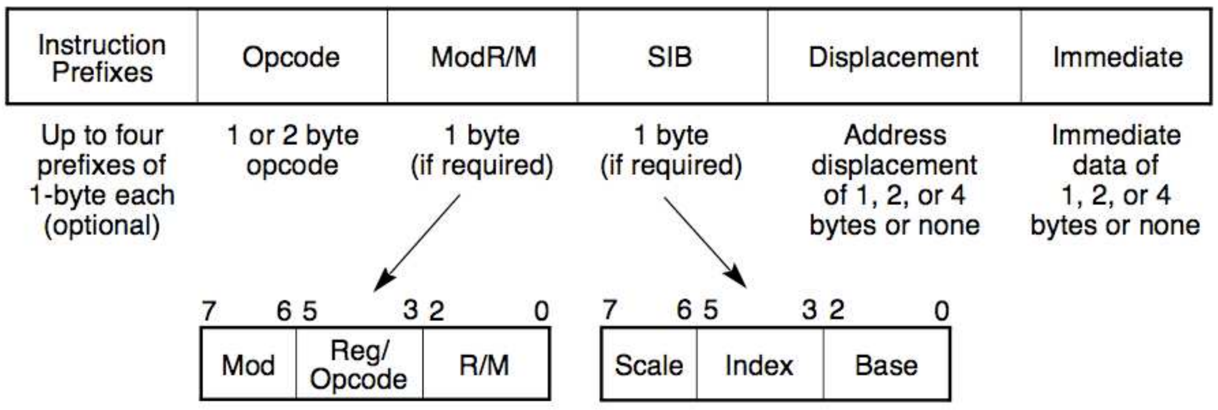


Figure 13. Intel instruction format
Source: Intel Programmer's Manual

The purpose of different fields of an instruction set is described below.

1. Instruction prefixes are used as modifiers to the main command. Prefixes can be used to repeat string operations, to provide segment overrides, and to change operand and address sizes.
2. An opcode is a one or more bytes long binary representation of assembly language mnemonic. While assembling, the assembler translates mnemonics to corresponding codes.
3. Mod field allows specifying which of the general purpose registers or addressing modes are used in an instruction.
4. Displacement field is used to provide a displacement value to an address referred in an instruction. For example, an ADD instruction with a reference to an address displaced by an offset 4056 can be represented as “ADD ax, [bp+di] + 4056”. The displacement can be 1, 2, or 4 bytes long.
5. An immediate operand is a constant, used as an operand in an instruction, which can be a 1, 2, or 4 bytes value. In an instruction, “ADD ax, 10”, immediate operand is 10.

Some of the basic properties of such instruction are as follows:

- The length of an assembled instruction varies based on number of fields and size of each field used in an instruction.
- A single mnemonic may be translated into different opcodes based on the type of operands used.
- The Mod field varies based on operand used.
- An operand can be a register, immediate, direct or indirect memory reference with or without displacement.
- Some fields are optional.

There are three types of registers: 8-bit, 16-bit and 32-bit registers represented as r8, r16 and r32. Table 4 below shows registers available in each type.

Table 4. Registers and corresponding register encodings

Register Encodings	r8	r16	r32
0	AL	AX	EAX
1	CL	CX	ECX
2	DL	DX	EDX
3	BL	BX	EBX
4	AH	SP	ESP
5	CH	BP	EBP
6	DH	SI	ESI
7	BH	DI	EDI

Source: Intel's Programming Manual

5.2.2 Preprocessing of executable code segment

Since there are more than 100 instructions in Intel x86 instruction set, rather than working on all those instructions, it is inevitable to take only the Most Frequently Occurred (MFO) instructions into account for three important reasons:

1. It is time-consuming to collect binary opcodes covering the whole instruction set to form opcode table.
2. The opcode table should be as small as possible to achieve better efficiency.
3. Training HMM with small set of MFO instruction opcodes allows HMM to find patterns or features of virus effectively

As per Billar, only fourteen instructions in entire Intel instruction set are MFO instructions. Those instructions are ADD, AND, CALL, CMP, JMP, JNZ, JZ, LEA, MOV, PUSH, POP, RETN, TEST and XOR. After a careful analysis, we found that using MFO instructions enables HMM to learn some patterns in the virus code and detect viruses more effectively. Figure 14 and 15 below shows the percentage of occurrence of 14 MFO opcodes in normal and malicious files respectively.

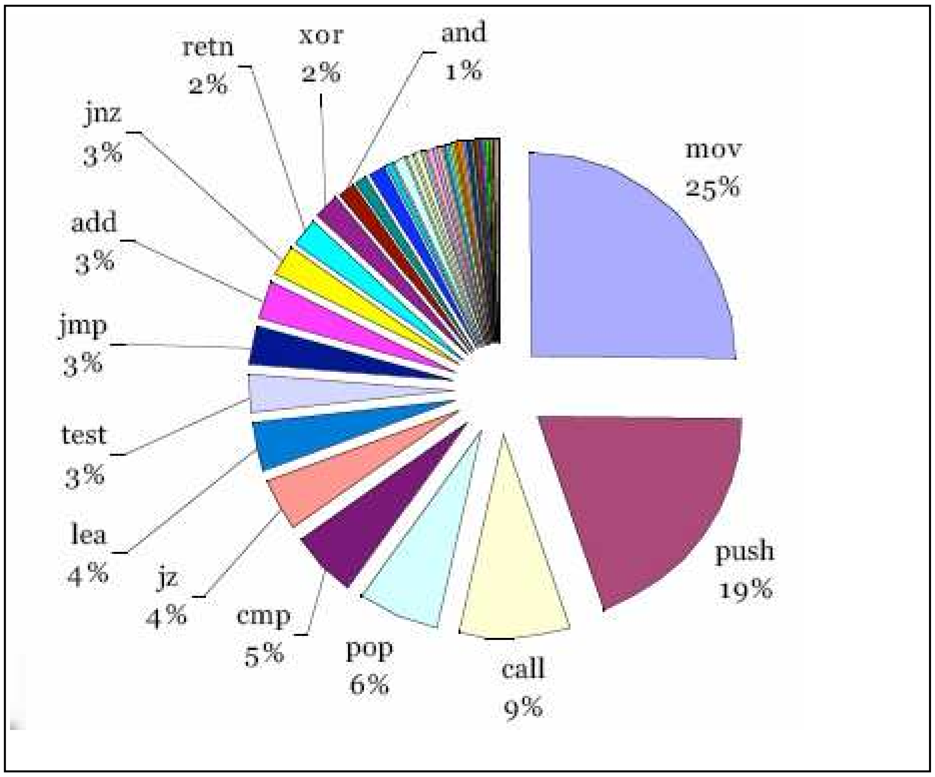


Figure 14. Frequency of Occurrence of 14 MFO opcodes in normal files (in percentage)
Source: Billar

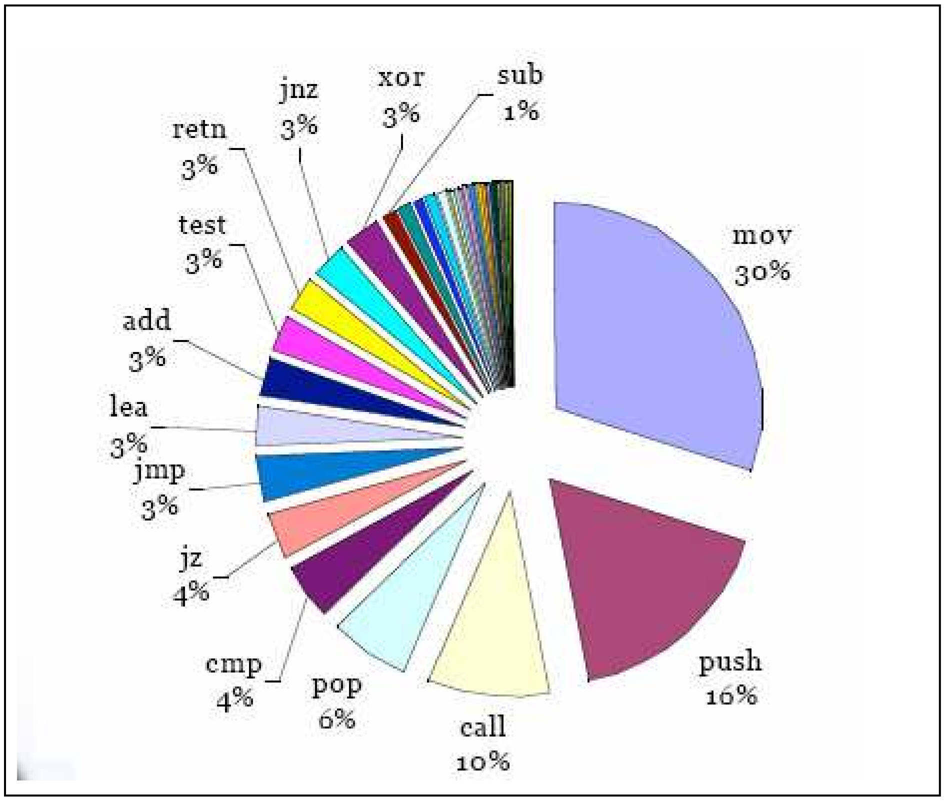


Figure 15. Frequency of Occurrence of 14 MFO opcodes in Malwares (in percentage)
Source: Billar

As demonstrated in figures 14 and 15, approximately 90% of total instructions used are 14 MFO instructions.

Billar et al. has also discussed the percentage of occurrence of 14 MFO opcodes in different categories of malwares like Viruses, Worms, Trojans and Bots. Table 5 (page 27) shows the frequency of occurrence in percentage.

As the key idea in our approach is to search for a binary instruction opcode in the code segment, there are possibilities for false predictions. For instance, when we search for a 1-byte binary opcode, it may potentially hit many operands with same byte value resulting in false positives. In this context, false positive occurs when an operand or a part of an irrelevant opcode is detected as an opcode in examination. For example, one of the opcodes for JMP is 0xEB and one of the opcodes for SUB is 0xEB83. When an operand 0xEB or the part of SUB opcode is detected as JMP, it is considered as a false positive prediction.

Table 5. Frequency of Occurrence of 14 MFO opcodes in different malwares

Opcode	Goodware	Bot	Trojan	Virus	Worm
MOV	25.3%	34.6%	30.5%	16.1%	22.2%
PUSH	19.5%	14.1%	15.4%	22.7%	20.7%
CALL	8.7%	11.0%	10.0%	9.1%	8.7%
POP	6.3%	6.8%	7.3%	7.0%	6.2%
CMP	5.1%	3.6%	3.6%	5.9%	5.0%
JZ	4.3%	3.3%	3.5%	4.4%	4.0%
LEA	3.9%	2.6%	2.7%	5.5%	4.2%
TEST	3.2%	2.6%	3.4%	3.1%	3.0%
JMP	3.0%	3.0%	3.4%	2.7%	4.5%
ADD	3.0%	2.5%	3.0%	3.5%	3.0%
JNZ	2.6%	2.2%	2.6%	3.2%	3.2%
RETN	2.2%	3.0%	3.2%	2.0%	2.3%
XOR	1.9%	3.2%	2.7%	2.1%	2.3%
AND	1.35%	0.5%	0.6%	1.5%	1.6%

Source: Billar

As discussed earlier in Intel x86 instruction set, the length of an opcode varies based on number of operands, types of registers and types of memory access used in an instruction. It may be 1, 2, or more bytes in length. As it will be time consuming to search for longer opcodes, after a careful analysis, it has been found that MFO instructions are mostly 1 or 2 bytes long. Further, we discovered that more the number of 2-byte opcodes used to identify MFO opcodes, better the accuracy. Due to the fact that the probability for an operand or data to have the same value as the two-byte opcode is less, we have tried to extend 1-byte opcodes to 2-byte opcodes. The 1-byte opcode that can not be converted into 2-byte opcode should be located based on some conditions rather than looking for it indiscriminately. We used a utility called Debug32 to find 2-byte alternatives for 1-byte opcode. Figure 16 below illustrates how 1-byte opcode is converted into 2-byte opcodes based on the type of registers used in the ADD instruction.

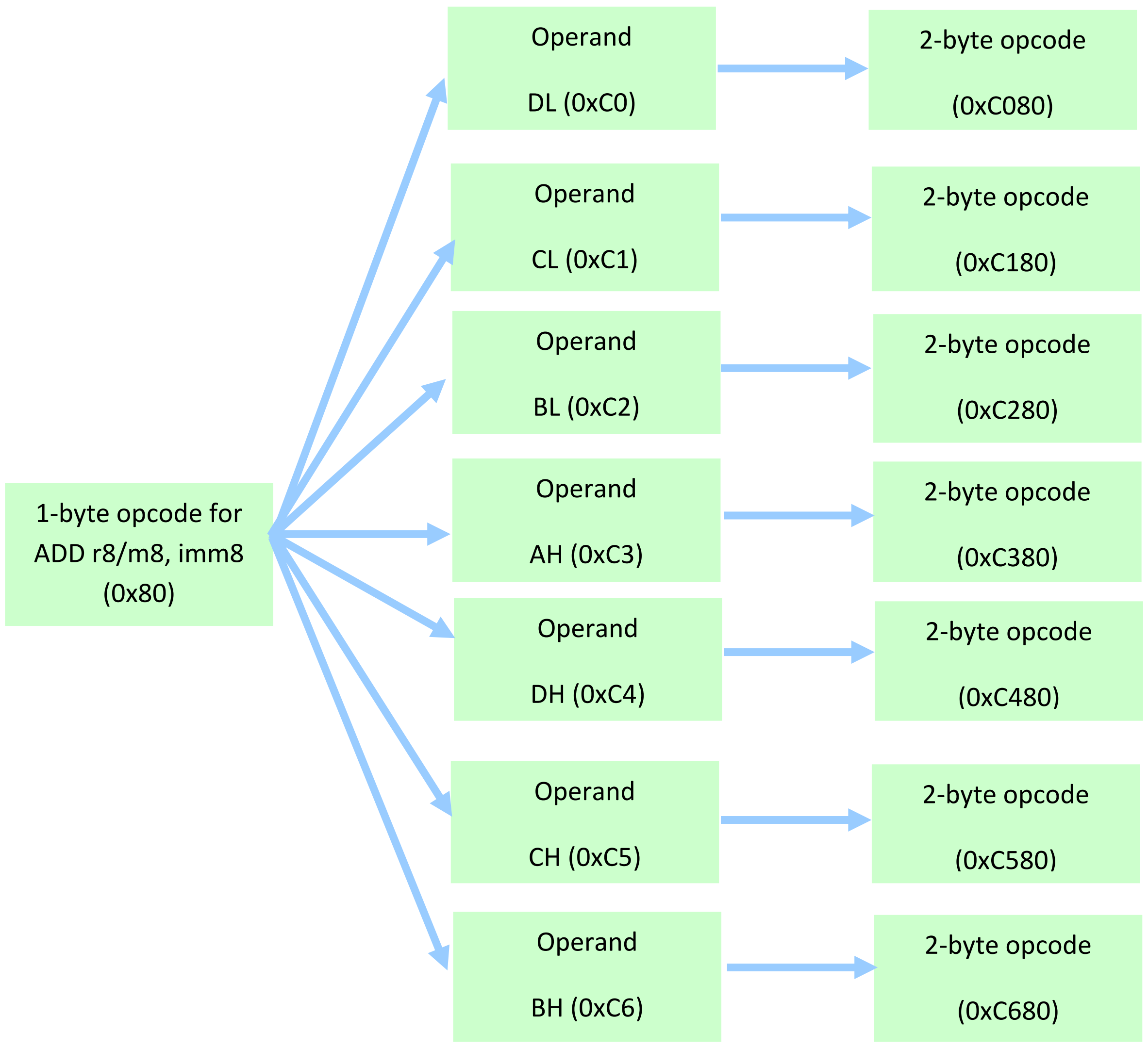


Figure 16. Convert 1-byte opcode to 2-byte opcode for ADD r8/m8, imm8
Source: Author's Research

As figure 16 illustrates, in the instruction “ADD r8/m8, imm8”, ADD refers to actual instruction or mnemonic, r8 refers to 8-bit register, m8 refers to 8-bit memory location and imm8 refers to 8-bit constant. The register references in this instruction can be substituted with any of the seven 8-bit registers (DL, CL, BL, AH, DH, CH, BH) to extend 1-byte opcode (0x80) into 2-byte opcodes.

After careful analysis of virus source files, we decided to collect all possible opcodes for register and direct memory addressing instructions and only MFO opcodes for indexed addressing instructions. Since including all the indexing instructions in the opcode table introduces all possible byte values from 0x00 to 0xFF in the second byte of the opcode, the probability of catching false positives is high. For example, binary opcode for instruction “ADD r8/m8, r8” is 0x02. In general, 1-byte opcode 0x02 can be extended to 2-byte based on type of register or memory addressing used. If we have to include all the indexing instructions for ADD, opcode table will require having all values from 0x0200 to 0x02FF. With the second byte position having a possibility of any value between 00 to FF, any operand or sub-opcode with value 0x02 will be detected as ADD regardless of the second byte.

Though effort has been made to change every 1-byte opcode to 2-byte, there are instructions whose opcodes cannot be extended. In most cases, the instructions with AL/AX/EAX as the first operand and imm8 as the second operand have 1-byte opcode. There is no way to extend these 1-byte opcodes to 2-byte.For example, binary opcode for instruction “ADD AL, imm8” is 0x04 which is an instruction referring the register AL directly. There are totally 60 such 1-byte opcodes for 14 MFO instructions of which 35 MFO opcodes are included in the collection. The 35 opcodes collected are the 1-byte opcodes of CMP, CALL, JMP, JNZ, JZ, POP and PUSH. The 1-byte opcodes for remaining instructions are neglected to avoid False Positives (FP).Finally, we maintained two sets of opcode list: 1-byte opcode list and two-byte opcode list.

A high level flow involved in detecting MFO opcodes are shown in Figure 15 below.

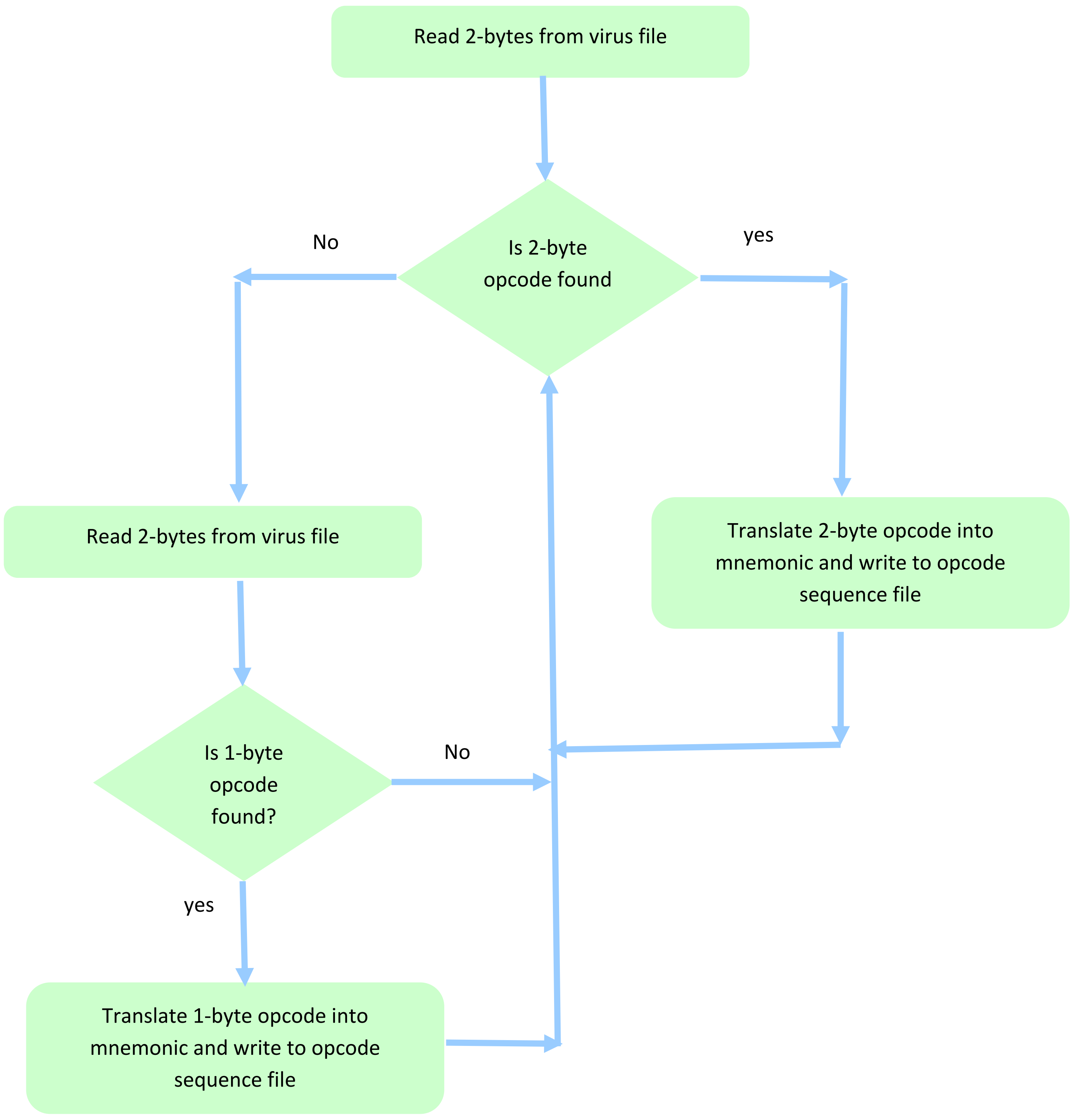

Read 2-bytes from virus file
No
Is 2-byte opcode found
yes
Read 2-bytes from virus file
Translate 2-byte opcode into mnemonic and write to opcode sequence file
Is 1-byte opcode found?
No
yes
Translate 1-byte opcode into mnemonic and write to opcode sequence file

Figure 17. Flow Diagram for MFO Opcode Detection
Source: Author's Research

Since most of the 1-byte instructions are PUSH and POP, we may end up catching False Positives (FP) for these instructions. So, we checked for certain conditions while detecting PUSH instructions based on the MFO pattern found in the virus assembly files. The pattern found for PUSH instruction is that PUSH is always followed by another PUSH or POP instruction. So, whenever 1-byte PUSH opcode is detected, the subsequent byte is checked for PUSH or POP. If the subsequent byte is detected as PUSH or POP, both of the bytes are added to observation sequence. Otherwise, both bytes are skipped.

In addition to PUSH and POP, we added conditions to detect 1-byte JMP. We noticed more FP for JMP because whenever our algorithm comes across 2-byte SUB, it is detected as JMP because both instructions are sharing a common opcode. In this case, let us consider the 1-byte opcodes 0xEB and 0xE9 for JMP and two-byte opcodes 0xEB83 and 0xE983 for SUB. As you notice here, both of the instructions are sharing the same opcode 0xEB and 0xE9. To avoid such false positives, whenever we encounter 0xEB and 0xE9, the consecutive byte is checked for 0x83. If the consecutive byte is detected as 0x83, both of the bytes are skipped. Otherwise, 1-byte 0xEB or 0xE9 is written as JMP in the observation sequence.

Using our algorithm, the generated opcode sequence for each virus file was 95% accurate with 5% being FP. It means that 20 out of 450 opcodes in the opcode sequence are FP.

5.3 Creating Opcode sequence

To create opcode sequences, an input set is formed with executables of virus. Input set is divided into three sets consisting of family viruses, non-family viruses and normal files. The virus generated by a same generator belongs to the same family and is referred as family virus. In contrast, virus generated by different generator belongs to different family and is referred as non-family virus. Family viruses are named as “NGVCKexes” consisting of 200 metamorphic virus variants generated by Next Generation Virus Creation Kit (NGVCK) generator. Non- Family viruses are named as

“OtherExes” consisting of 25 virus generated by Second Generation virus Generator (G2) and Mass Code Generator (MPCGEN). It includes

- 15 virus variants generated by Second Generation virus generator (G2) version 0.70a released in January 1993 representing non-family virus
- 10 virus variants generated by Mass Code Generator (MPCGEN) version 1.0 released in 1993 representing non family virus

The normal files are 40 random utility executables collected from Cygwin DLL (version 1.5.25).

Wong and Stamp collected 10 G2, 10 VCL32 and 5 MPCGEN as non-family virus. VCL32 generated files has some properties that doesn’t allow us to include it as input set for our program. VCL32 generated files have all the function definitions inside data sections and only function calls in code section. Due to the reason that code section is same in all VCL32 virus executables and our program extracts only the code section to extract opcode sequences, we have not considered VCL32 files.

Once an input set is created, it is given as input to “create_obs.exe” program where “Obs” stands for observation sequence or opcode sequence. The output of this program is the data set and the compare set. A data set of 200 individual files each consisting of corresponding opcode sequence is created and a compare set of 65 individual non-family viruses and normal files consisting of corresponding opcode sequence is created.

5.4 Training and Testing HMM

Training and testing followed the same methodology of (Wong, 2006). Five-fold cross validation is applied to the data set and divided into train set and test set. So, train set consists of 160 virus opcode sequence (four subsets each with 40 viruses) and test set (one subset) consists of 40 virus opcode sequence. Each time, a different test set is selected and other four subsets are used as train set. This process is repeated five times. The length of each train file in data set ranges from 395 to 445 with an average of 420. So, the typical length of concatenated 160 opcode sequence is in the range of 65,450 to 65,650 with an average of 65,550.

Once HMM is trained with the concatenated opcode sequence, a model is created for every train set. After training, the test set and compare set is scored with corresponding trained model. For each file in test set and compare set, Log Likelihood Per Opcode (LLPO) is calculated as its score. For further details about LLPO, refer (Wong, 2006). A threshold value is also calculated which is an average of minimum LLPO in data set and maximum LLPO in compare set. The files with scores above (greater than) the threshold are classified as virus and files with scores below (less than) the threshold are classified as non- virus or non member. Training and classifying is explained in figure 18. The steps followed in training and classifying are

1. Train HMM with train set consisting of 160 opcode sequence files
2. Score and calculate LLPO for files in test set and compare set
3. Determine threshold value to classify member virus and non-members
4. Continue step 1 until all test sets are scored

These steps are diagrammatically shown in figure 18 (page 36).

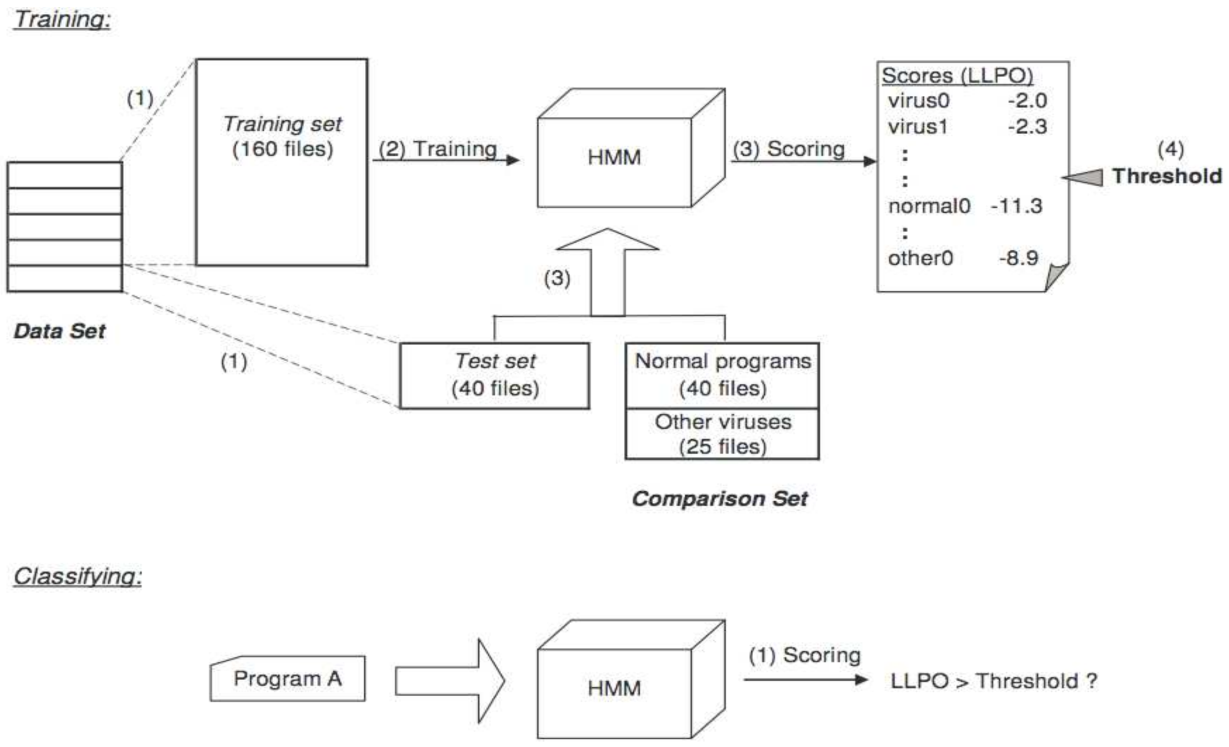


Figure 18. Training and classifying process
Source: Wong, 2006

6. Experiment Setup and Results

Section 6.1 describes the input data, platform setup and programming languages used in the experiment. Section 6.2 provides the results obtained using our method which eliminates disassembling and works on 14 MFO opcodes. Section 6.3 provides the results obtained using Wong's method which uses disassembling and works on all opcodes in Intel instruction set. In the final section, we compare results of our method with results of Wong's method to test the accuracy and efficiency of our method.

6.1 Experiment Setup

As discussed earlier, input set consists of three set of executables. First set consists of 200 NGVCK executables named as N0 to N199 (N stands for NGVCK), second set consists of 15 G2 executables named as G2T0 to G2T14 and 10 MPCGEN executables named as MPC0 to MPC9, and third set consists of 40 Cygwin executables named as CYG0 to CYG39.

Extracted code section from each virus executable is collected in ICS (Individual Code Section) Data Set and named as cs_n0 to cs_n199.

Data set consists of 200 NGVCK opcode sequence files named as OBSN0 to OBSN199 (OBS stands for observation sequence and N stands for NGVCK). Compare set consists of 40 Cygwin opcode sequence files named as OBSC0 to OBSC39 (C stands for Cygwin) and 25 non-family virus opcode sequence files named as OBSV0 to OBSV24 (V stands for other virus).

TrainFile consists of 10 files, 5 being “alphabet” file consisting of distinct opcodes in each train set and 5 being “in” (in stands for input) file consisting of concatenated 160 opcode sequence in test set. Each alphabet and input file is named 160_OBSN_E0 to 160_OBSN_E4. In the file name, 160 stands for number of opcode sequences being concatenated, OBS stands for observation sequence, N stands for NGVCK and E0 stands for excluded set 0 which is the test set.

With number of states N being different each time ranging from 2 to 6, let us see how models are named. There are 25 models created by HMM with 5 being created for each state N. If a model is named as 160_OBSN_N2_E0, then

- 160 is the number of files in train set
- OBSN stands for NGVCK observation sequence
- N2 stands for number of states as 2
- E0 stands for test set 0

Table 6 below shows the experiment platform and programming languages used.

Table 6. Experiment Setup

Platform	Windows XP
Virus Generators	NGVCK, G2 and MPCGEN
Programming Languages	C, Ruby
Assembler & Linker	TASM, TASM32, TLINK, TLINK32, MSVC 6.0, Ruby
Utilities	HexDump, Debug32

Source: Author's Research

6.2 Experiment Results I

With N ranging from 2 to 6, and test sets ranging from 0 to 4, 25 models were created with HMM.

Let us examine how the HMM separated family viruses from compare set files. All 25 models made a clear separation of scores between family viruses and compare set files. Each model scored a data set consisting of 40 family viruses and compare set consisting of 40 normal files and 25 non-family viruses. Table 7 shows LLPO scores of 40 family viruses and 40 normal files. The scores show that LLPO scores of family viruses are -1.9 or greater and LLPO scores of normal files are -2.1 or lower.

Table 7. LLPO scores of 40 family viruses and 40 normal files (compare set) using model 160_0BS_N2_E0.

NGVCK Family Viruses				Normal cygwin files			
Virus Name	LLPO	Virus Name	LLPO	File Name	LLPO	File Name	LLPO
OBSN0	-1.91341	OBSN20	-1.85286	OBSV0	-2.15787	OBSV20	-2.52410
OBSN1	-1.91630	OBSN21	-1.85252	OBSV1	-2.10833	OBSV21	-2.58423
OBSN2	-1.94792	OBSN22	-1.87886	OBSV2	-2.48227	OBSV22	-2.42321
OBSN3	-1.78941	OBSN23	-1.94889	OBSV3	-2.49157	OBSV23	-2.44344
OBSN4	-1.81915	OBSN24	-1.91749	OBSV4	-2.39297	OBSV24	-2.51328
OBSN5	-1.88139	OBSN25	-1.84351	OBSV5	-2.53091	OBSV25	-2.63752
OBSN6	-1.89580	OBSN26	-1.82954	OBSV6	-2.75892	OBSV26	-2.21347
OBSN7	-1.85012	OBSN27	-1.87690	OBSV7	-2.75575	OBSV27	-2.46925
OBSN8	-1.86159	OBSN28	-1.85007	OBSV8	-2.48225	OBSV28	-2.54372
OBSN9	-1.91538	OBSN29	-1.89606	OBSV9	-2.46713	OBSV29	-2.46418
OBSN10	-1.83419	OBSN30	-1.93708	OBSV10	-2.48225	OBSV30	-2.50300
OBSN11	-1.78523	OBSN31	-1.87644	OBSV11	-2.46713	OBSV31	-2.85430
OBSN12	-1.88537	OBSN32	-1.80577	OBSV12	-2.37040	OBSV32	-2.47473
OBSN13	-1.82211	OBSN33	-1.84254	OBSV13	-2.71943	OBSV34	-2.24818
OBSN14	-1.90262	OBSN34	-1.86094	OBSV14	-2.71957	OBSV34	-2.49244
OBSN15	-1.91341	OBSN35	-1.92944	OBSV15	-2.49580	OBSV35	-2.49583
OBSN16	-1.87386	OBSN36	-1.90475	OBSV16	-2.51546	OBSV36	-2.69585
OBSN17	-1.81544	OBSN37	-1.82279	OBSV17	-2.39297	OBSV37	-2.49893
OBSN18	-1.91167	OBSN38	-1.86641	OBSV18	-2.71439	OBSV38	-2.53286
OBSN19	-1.90808	OBSN39	-1.89339	OBSV19	-2.44965	OBSV39	-2.56675

Source: Author

Figure 19 below shows the scores of test set 1 and scores of compare set files for model with three states; i.e., N=3 . There is a clear distinction of scores between family and non-family viruses. Two of the normal files have scores closer to family virus scores but doesn't interleave the family virus scores.

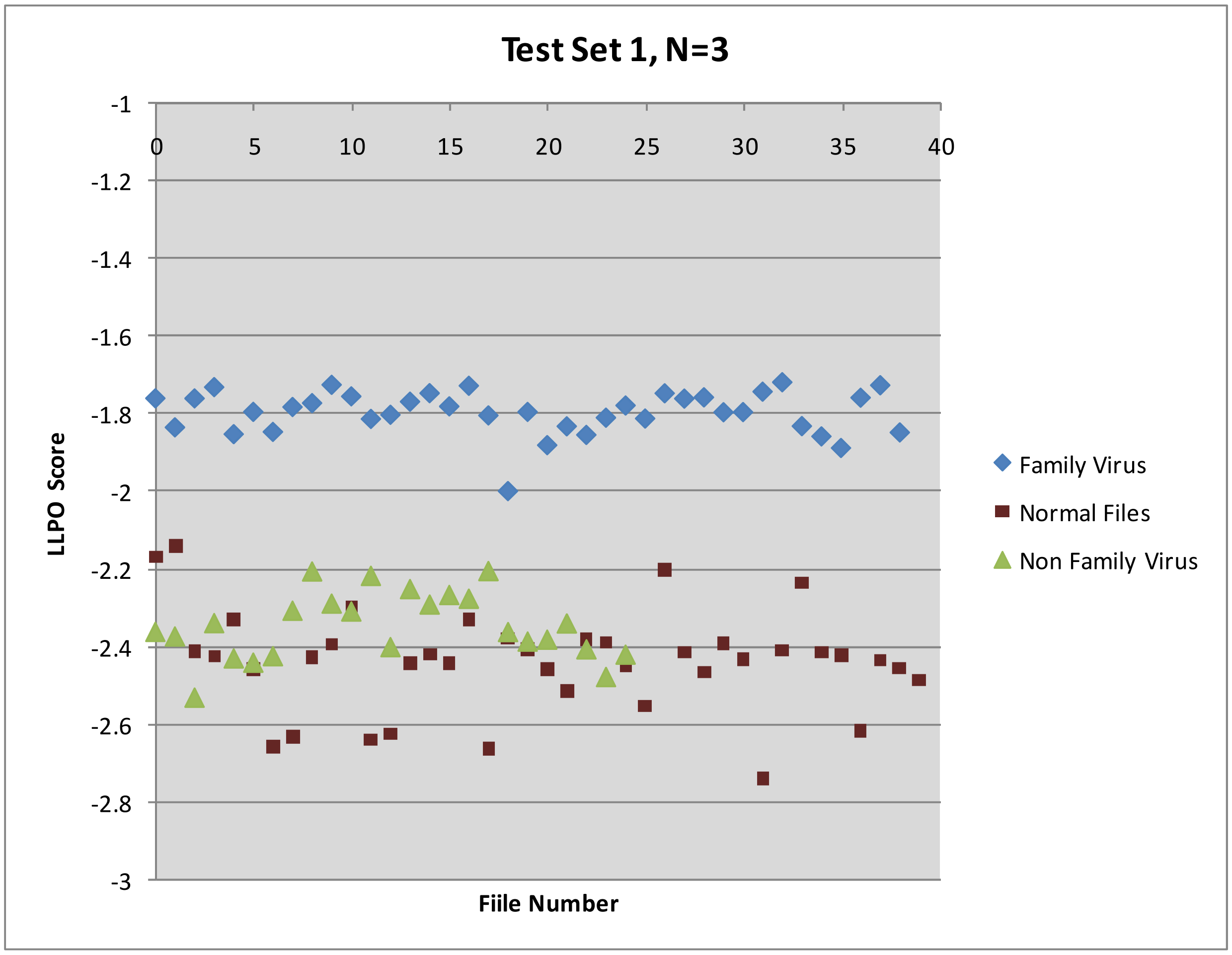


Figure 19. Log Likelihood per Opcode (LLPO) of family viruses, non-family viruses and normal files
Source: Author's Research

The score results shown in the above diagram is the typical range of scores we obtained for all models. Refer Appendix B to view the graphs for all states. The overall results show that HMM is able to separate the family viruses from normal files and non-family viruses regardless of number of states.

To classify a file as family virus or non-member, we need to determine a cutoff or threshold value. The files which are scored greater than threshold are considered as family viruses and those which are scored lower than threshold is considered as non-members. Threshold is calculated as the average of minimum score of family virus and maximum score of non member files.

$$\text{Threshold} = (\text{MinDataLog} + \text{MaxCompareLog})/2$$

where

MinDataLog is the minimum score of family virus

MaxCompareLog is the maximum score of non member files

If score of a family virus is lower than threshold, it results in False Negative (FN) prediction because a family virus is classified as non-member file. In other hand, if score of a non-member file is greater than threshold, it results in False Positive (FP) prediction because a non-member file is classified as family virus.

Table 8 shows the minimum score of NGVCK family viruses, maximum score of non-member files and corresponding threshold assigned by each model. There are 25 different scores corresponding to 25 models. Two greatest and lowest thresholds are marked bold in Table 8.

Table 8. Minimum score of NGVCK family viruses, maximum score of non-member files and threshold assigned by model

Test Set		Min score of family viruses	Max score of non member files	Threshold
Test Set 0	N = 2	-1.9488	-2.1083	-2.0286
	N = 3	-1.8745	-2.1342	-2.0044
	N = 4	-1.8633	-2.0813	-1.9723
	N = 5	-1.8230	-2.0417	**-1.9323**
	N = 6	-1.7994	-2.0841	**-1.9448**
Test Set 1	N = 2	-1.9252	-2.1490	**-2.0957**
	N = 3	-1.8896	-2.1400	**-2.0710**
	N = 4	-1.9810	-2.1048	-2.0429
	N = 5	-1.9438	-2.1413	-2.0426
	N = 6	-1.9645	-2.1667	-2.0510
Test Set 2	N = 2	-1.9381	-2.1456	-2.0438
	N = 3	-1.8905	-2.1396	-2.0151
	N = 4	-1.8632	-2.1055	-1.9843
	N = 5	-1.8381	-2.1418	-1.9900
	N = 6	-1.8158	-2.1345	-1.9752
Test Set 3	N = 2	-1.9289	-2.1429	-2.0359
	N = 3	-1.8661	-2.1337	-1.9999
	N = 4	-1.8496	-2.0998	-1.9747
	N = 5	-1.8311	-2.1361	-1.9836
	N = 6	-1.8158	-2.1411	-1.9785
Test Set 4	N = 2	-2.0463	-2.1441	-2.0952
	N = 3	-1.9836	-2.1357	-2.0596
	N = 4	-1.9500	-2.1000	-2.0250
	N = 5	-1.9185	-2.1362	-2.0274
	N = 6	-1.9368	-2.1457	-2.0413

Source: Author's Research

A single threshold should be determined from the 25 thresholds assigned by the model. The determined threshold will act as a cutoff point for all the model scores. If the determined threshold is too small, FP rate will be increased. If the determined threshold is too large, FN rate will be increased. The final threshold which is greater than all non-member files and lower than all family viruses will avoid more FP and FN. We experimented with four different threshold values. The corresponding false prediction rate can be viewed in Table 9 (page 50). The thresholds used for the experiment are -1.93, -1.94, -2.07, -2.09. When the threshold is as large as -1.93, there are 15 FN. So, only 25 of 40 family viruses are classified as family viruses and remaining 15 is classified as normal file. Of the four thresholds used, only -2.09 and -2.07 results in detection rate greater than 95%. -2.07 is considered as final threshold because the number of false prediction is as low as 2 when compared to 4 for -2.09. The above false prediction is FP resulting in classification of 2 non-member files as family viruses. Since there are no FN when threshold is set to -2.07, detection rate is determined as 1.0000 where

Detection Rate = TP / #FV

where

TP - True Positives which means number of family viruses classified as family viruses

#FV - Total number of family viruses

In the above case where threshold is -2.07, all 40 family viruses are classified as family viruses. So the detection rate is 1.000.

Table 9. Thresholds and False Predictions

Test Set		-1.93			-1.94			-2.07			-2.09		
		FP	FN	Detect Rate	FP	FN	Detect Rate	FP	FN	Detect Rate	FP	FN	Detect Rate
N = 2	Test Set 0	0	3	0.925	0	2	0.95	0	0	**1.000**	0	0	1.000
	Test Set 1	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 2	0	0	1.000	0	0	1.000	0	0	**1.000**	1	0	1.000
	Test Set 3	0	0	1.000	0	0	1.000	2	0	**1.000**	2	0	1.000
	Test Set 4	0	0	1.000	0	0	1.000	0	0	**1.000**	1	0	1.000
N = 3	Test Set 0	0	1	0.975	0	1	0.975	0	0	**1.000**	0	0	1.000
	Test Set 1	0	1	0.975	0	1	0.975	0	0	**1.000**	0	0	1.000
	Test Set 2	0	1	0.975	0	1	0.975	0	0	**1.000**	0	0	1.000
	Test Set 3	0	1	0.975	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 4	0	1	0.975	0	1	0.975	0	0	**1.000**	0	0	1.000
N = 4	Test Set 0	0	1	0.975	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 1	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 2	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 3	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 4	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
N = 5	Test Set 0	0	2	0.95	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 1	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 2	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 3	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 4	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
N = 6	Test Set 0	0	3	0.925	0	1	0.975	0	0	**1.000**	0	0	1.000
	Test Set 1	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 2	0	1	0.975	0	1	0.975	0	0	**1.000**	0	0	1.000
	Test Set 3	0	0	1.000	0	0	1.000	0	0	**1.000**	0	0	1.000
	Test Set 4	0	1	0.975	0	0	1.000	0	0	**1.000**	0	0	1.000

Source: Author's Research

Now, let us examine the training time of HMM to train each model. By default, HMM is trained for 800 iterations. The running time of each iteration depends on number of states N and length of observation sequence T. In our experiment, value of N ranges from 2 to 6 and average observation sequence length is 65,450. The training time of HMM ranges from 31 seconds for N =2 to 18 minutes for N = 6. Figure 20 below shows the training time taken in seconds to create models with N ranging from 2 to 6.

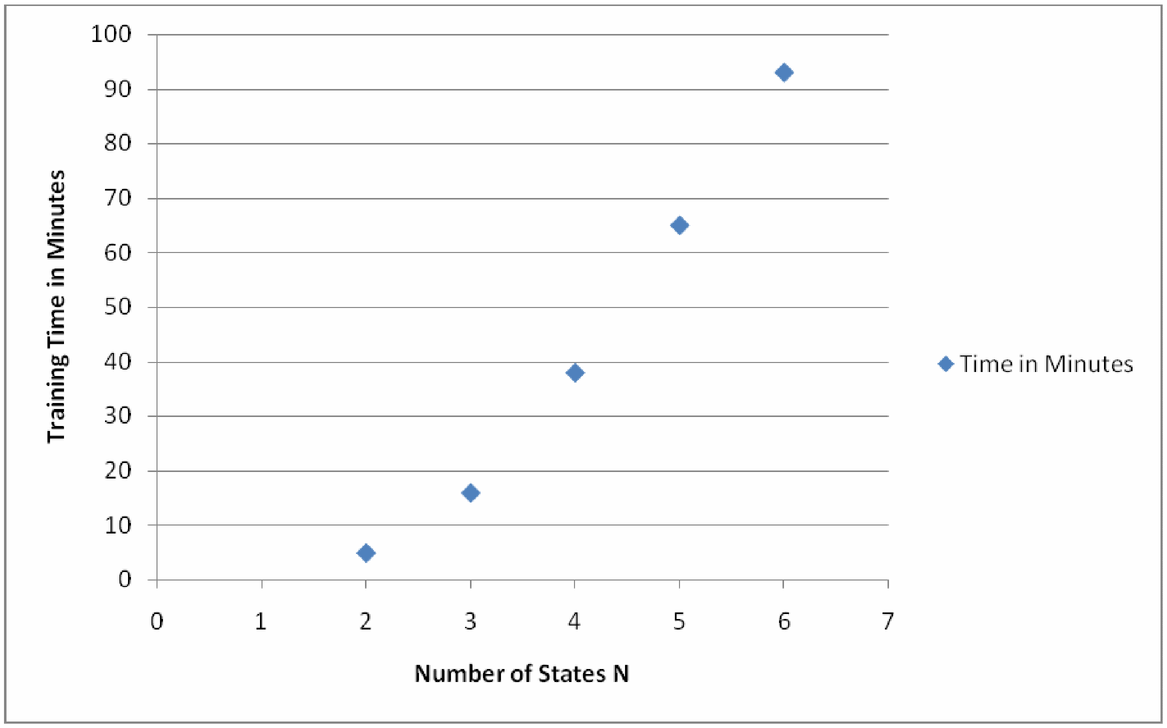


Figure 20. Training time of 25 models for 800 iterations
Source: Author's Research

Eventually, the trained model creates A, B and Pi matrices where A matrix is the state transition probability, B matrix is the observation probability and Pi is the initial state distribution. To examine the features of a virus, HMM observes the observation sequence and plot the values in the B matrix. So, after a model is trained, HMM assigns probability of occurrence of each opcode in particular state which can be viewed in B matrix. Table 10 (page 52) shows transpose of B matrix for 2 states and test set 2.

Table 10. Transposed B matrix for N = 2 and Test set 2

Opcode	State 0	State 1
Call	0.04139156206336	0.10697308319445
add	0.03274170541118	0.19756946116809
cmp	0.00712020350956	0.0329235076862
jz	0	0.0329235076852
lea	0.03349106402365	0.02797364747966
mov	0.7834975094335	0.19927252251573
xor	0	0.02828160382626
jnz	0	0.04193156798704
jmp	0	0.05445961728529
pop	0	0.03281600972899
push	0.06910259796872	0.11012487167077
retn	0.03265535759005	0.06104544906463
and	0	0.01339285867145
test	0	0.0169689229576

Source: Author's Research

In table 10 above, any state with zero value means that the corresponding opcode doesn't belong to that state. For example, opcode jz has zero value in state 0 and non-zero value in state 1 which implies that jz occurs only in state 1.

In figure 21, the above table is plotted. The graph shows that opcode MOV occurs mostly in state 0. Opcodes XOR, POP, AND, TEST, JZ, JNZ and JMP occur only in state 1 and have zero probability in state 0. Rest of the opcodes occurs in both states.

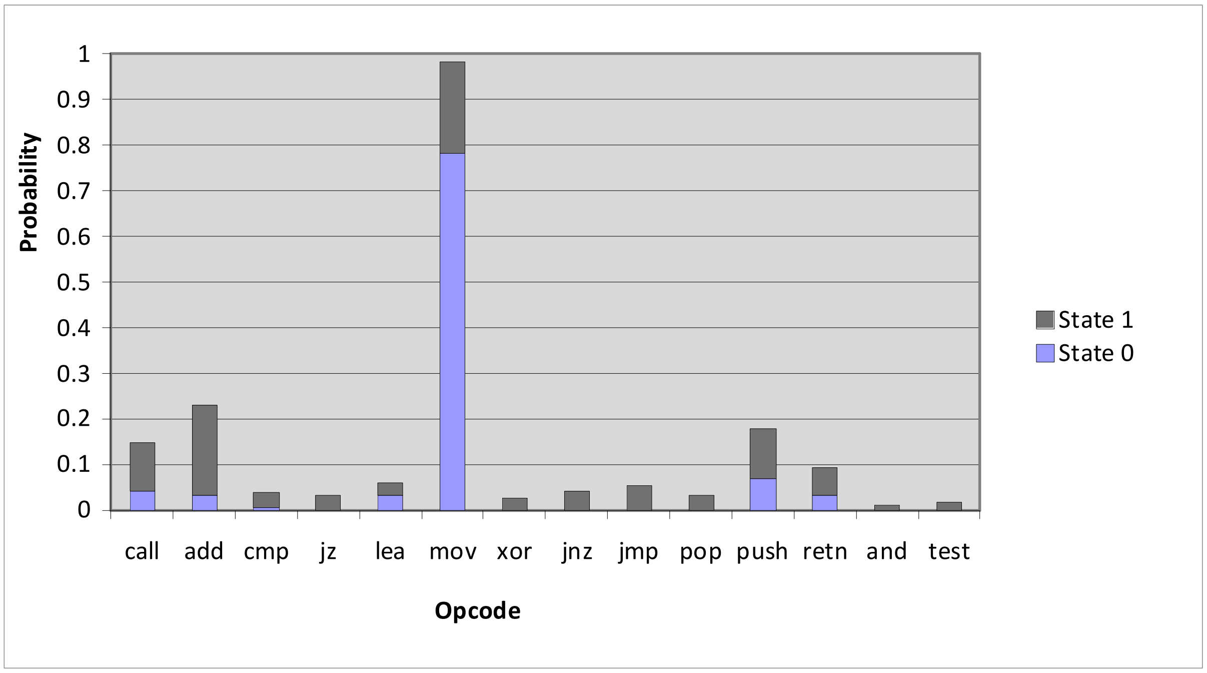


Figure 21. Probability distribution of observation symbols in each state for N = 2 and test set 2
Source: Author's Research

6.3 Experiment Results II from Wong's method

As discussed earlier, Wong's method require disassembled executables as input. First, all input executables should be disassembled. Using IdaPro, we disassembled the same set of input files (200 NGVCK, 40 Cygwin, 15 G2 and 10 MPCGEN executable files) used in our method and created respective asm files. We used the generated asm files as input to the HMM. The typical observation sequence length of concatenated opcode sequence ranges from 91,830 to 92,430 with an average of 92,130.

With N ranging from 2 to 6, and test sets ranging from 0 to 4, 25 models were created with HMM. Let us examine how the HMM separated family viruses from compare set files. All 25 models made a clear separation of scores between family viruses and compare set files. Each model scored a data set consisting of 40 family viruses and compare set consisting of 40 normal files and 25 non-family viruses.

Figure 22 shows the scores of test set 1 and scores of compare set files for model with three states; i.e., N=3 . There is no clear distinction and some interleaving of scores between family viruses and normal files. About three scores of normal files are interleaving with scores of family viruses.

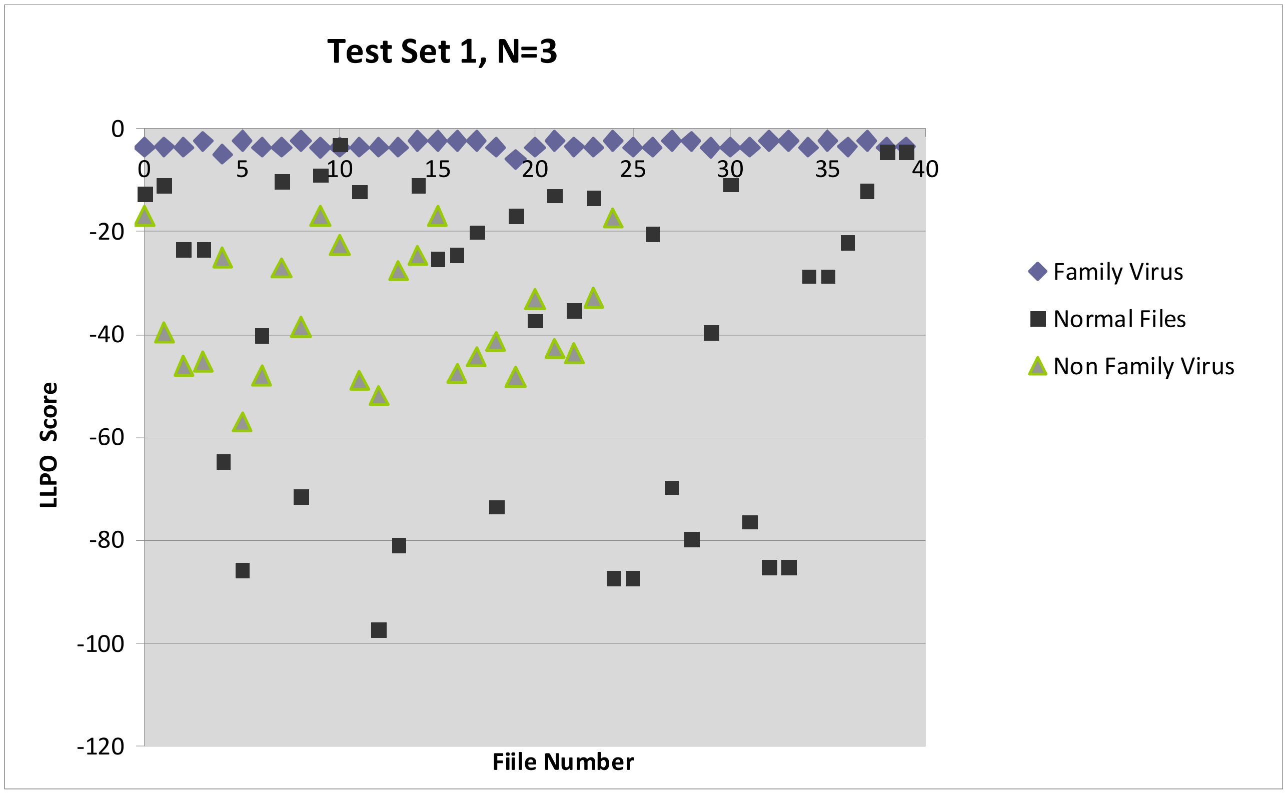


Figure 22. Log Likelihood per Opcode (LLPO) of family viruses, non-family viruses and normal files
Source: Author's Research

HMM is not able to determine a well defined threshold for any of the models, since the maximum score of compare set is lesser than the minimum score of data set. For example, for the model with N=3 and test set 1, the minimum score of data set is -5.9 and the maximum score of compare set is -3.0. Since, -5.9 is lesser than -3.0, it is not able to find threshold. Also, due to the fact that all the models have interleaving scores, HMM doesn't find well defined threshold. So, after analyzing all the scores and keeping the detection rate greater than 95%, we determine -5.4 as the threshold. With -5.4 as threshold, there are 39 FP predictions and 7 FN predictions. Table 11 shows the FP and FN for each model.

Table 11. False Predictions for threshold = -5.4

Model		-5.4		
		FP	FN	Detection Rate
N = 2	Test Set 0	1	0	1.000
	Test Set 1	3	1	0.975
	Test Set 2	3	0	1.000
	Test Set 3	3	0	1.000
	Test Set 4	3	0	1.000
N = 3	Test Set 0	0	0	1.000
	Test Set 1	3	1	0.975
	Test Set 2	3	1	0.975
	Test Set 3	3	0	1.000
	Test Set 4	3	0	1.000
N = 4	Test Set 0	1	0	1.000
	Test Set 1	3	1	0.975
	Test Set 2	3	0	1.000
	Test Set 3	3	0	1.000
	Test Set 4	3	0	1.000
N = 5	Test Set 0	1	0	1.000
	Test Set 1	3	1	0.975
	Test Set 2	3	1	0.975
	Test Set 3	3	0	1.000
	Test Set 4	3	0	1.000
N = 6	Test Set 0	1	0	1.000
	Test Set 1	3	1	0.975
	Test Set 2	1	0	1.000
	Test Set 3	1	0	1.000
	Test Set 4	1	0	1.000

Source: Author's Research

The diagrammatic representation of table 11 can be viewed in figure 23.

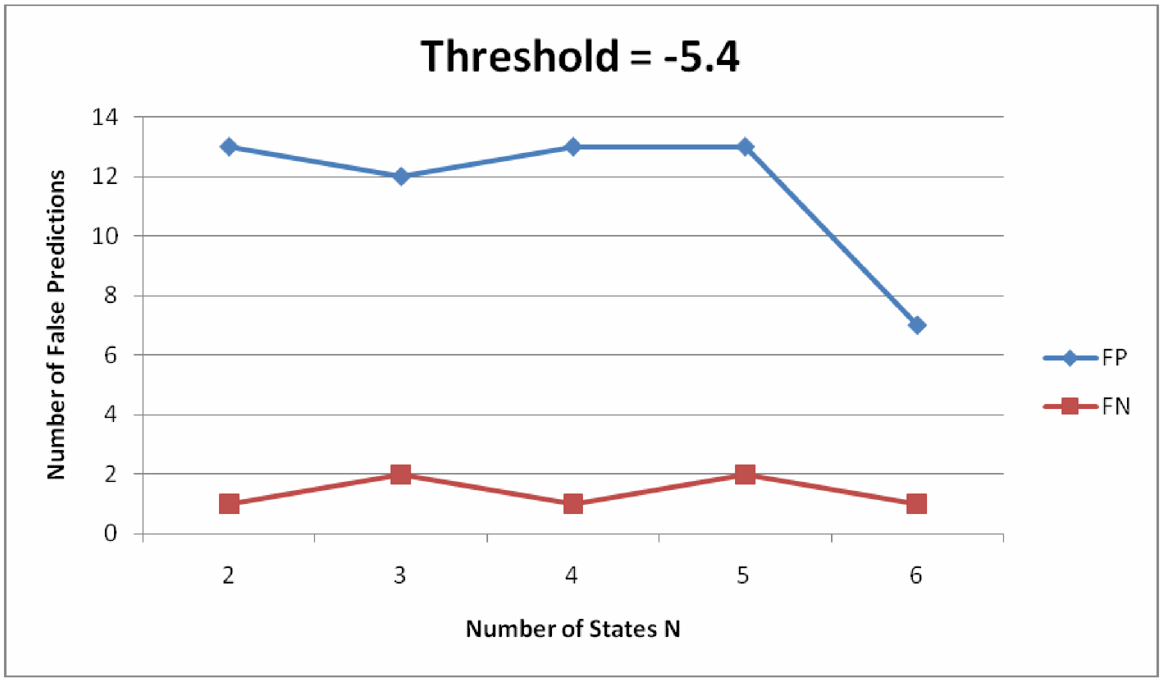


Figure 23. Number of false predictions at each state N
Source: Author's Research

Now, let us examine the training time of HMM to train each model. In default, HMM is trained iteratively for 800 iterations. The run time of each iteration depends on number of states N and length of observation sequence T. In our experiment, value of N ranges from 2 to 6 and average observation sequence length is 92,130. The training time of HMM ranges from 5 mins for N =2 to 48 minutes For N = 6. Figure 23 shows the training time taken in seconds to create models with N ranging from 2 to 6.

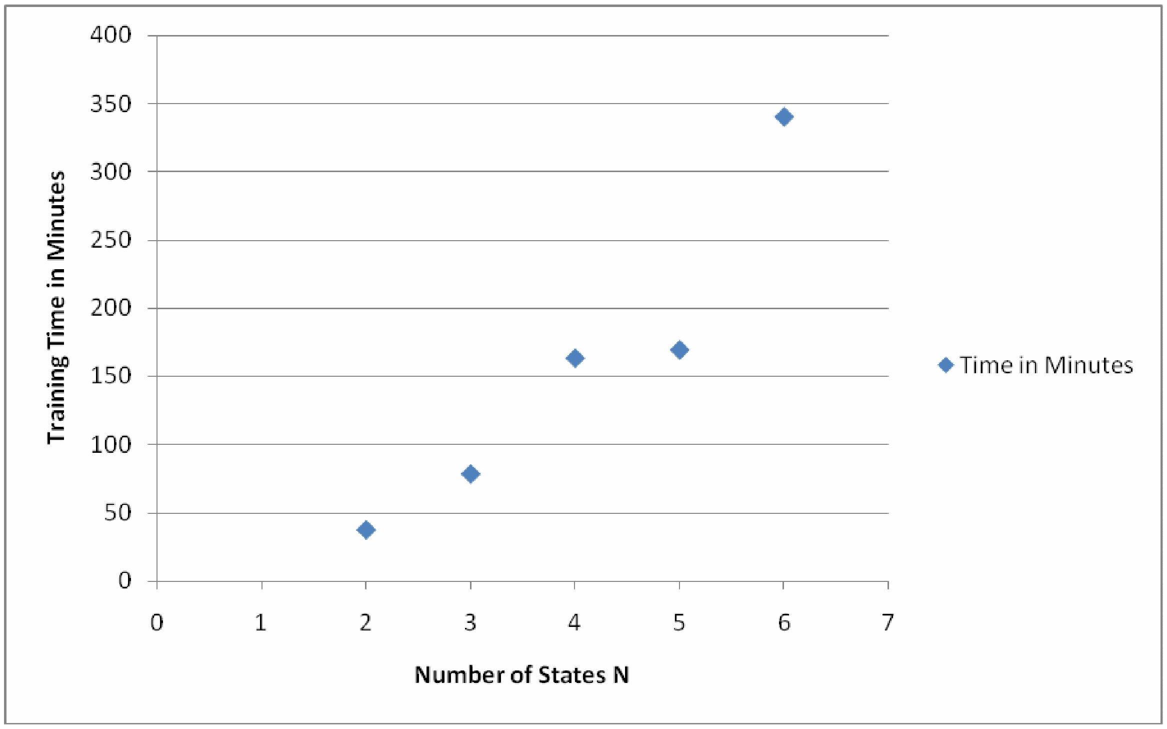


Figure 24. Training time of 25 models for 800 iterations
Source: Author's Research

6.4 Comparison of our method with Wong's Method

To determine the efficiency and accuracy of our method, our results are compared with Wong's method. The observation sequence length and training time are compared in figures 24 and 25 (page 58) respectively. The comparison shows that our method produces smaller opcode sequence since we extracted only 14 MFO opcodes which eventually results in lesser training time. Using our method, the training time is reduced by 60%. So, our method shows significant improvement in efficiency.

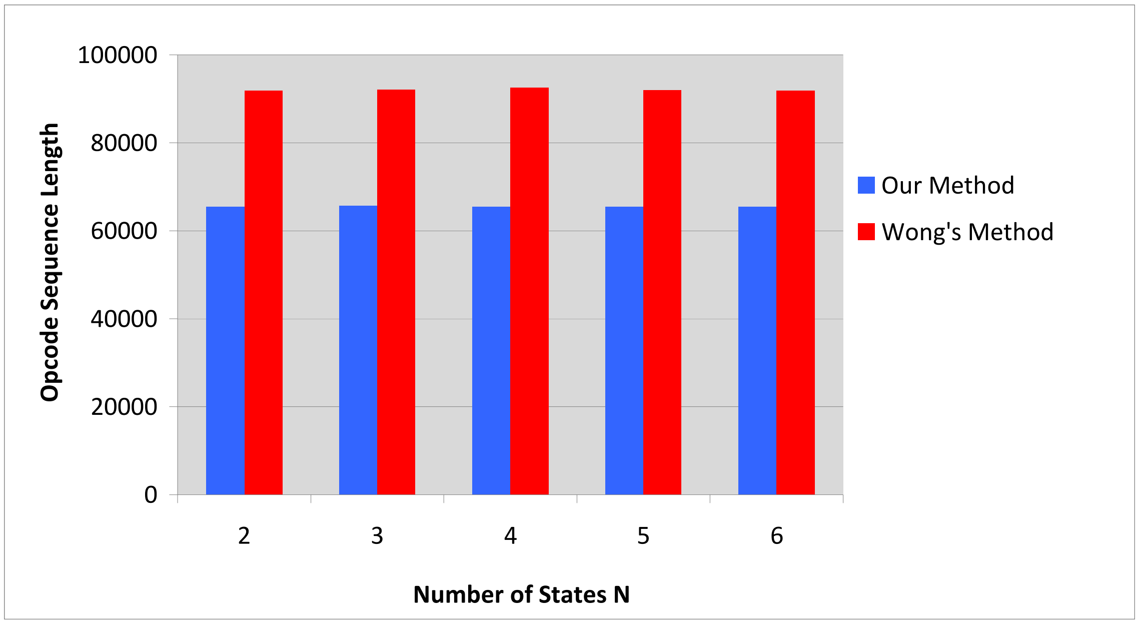


Figure 25. Comparison of opcode sequence length T in both methods
Source: Author's Research

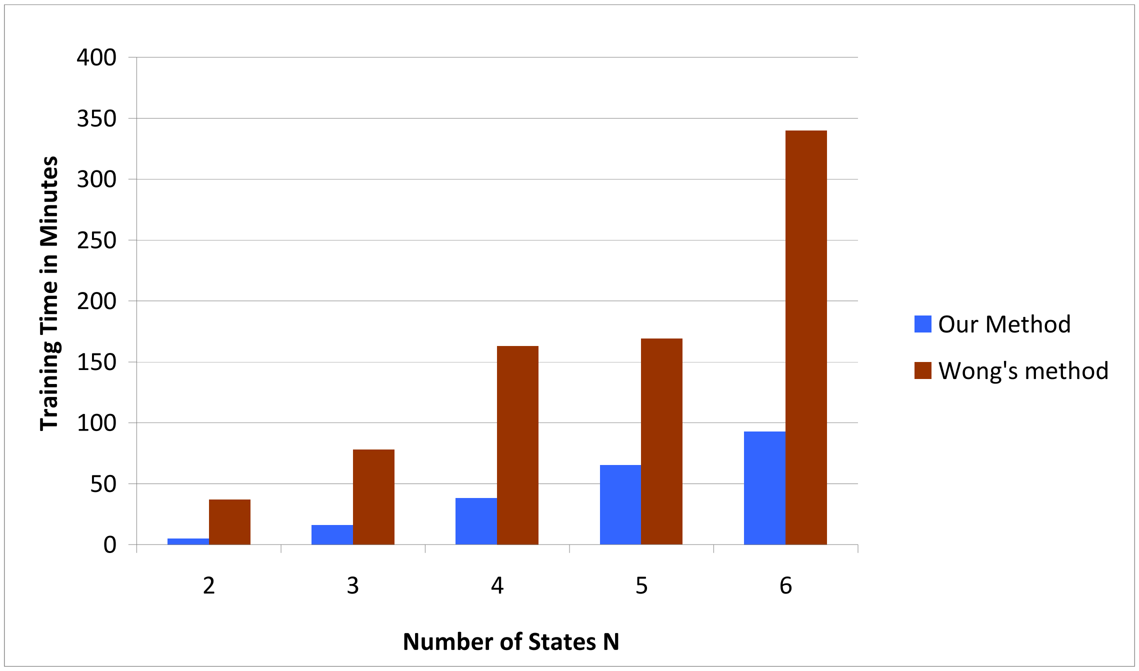


Figure 26. Comparison of HMM training time in both methods
Source: Author's Research

The total HMM training time is on average 4.5 hours for our method and 14.5 hours for Wong's method. Also, our program detects opcodes in the executables in less than 5 minutes in comparison to IDApro disassembling which takes on average 1.5 hours for same set of files. For the entire experiment, our method took only 4.5 hours compared to 16 hours for Wong's method. In summary, the overall performance is improved by 70% with our method when compared to Wong's method.

In addition to performance, there is a clear distinction of scores between family viruses and non-members in our method. With threshold set at -2.07, there are only 2 FP predictions and no FN predictions resulting in 100% detection rate. In Wong's method, there is some interleaving of scores between family viruses and normal files. With threshold set at -5.4, there are 39 FP predictions and 7 FN predictions resulting in 97% detection rate. This shows that accuracy is significantly improved in our method when compared to Wong's method. Figure 27 shows the number of false predictions in our method and Wong's method.

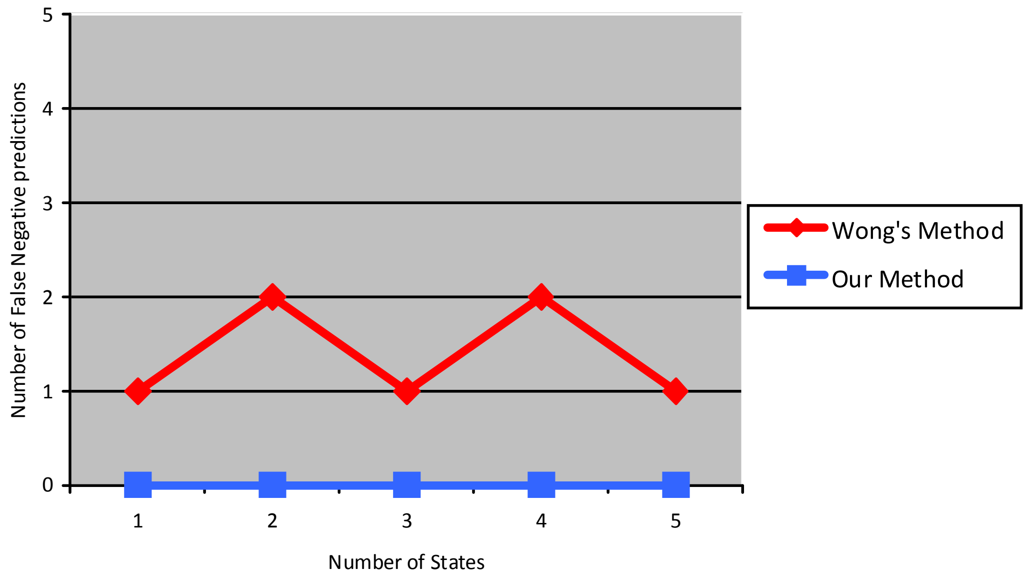


Figure 27. Comparison of False Negative Prediction
Source: Author's research

7. Conclusion

Our method extracts code section from the virus binary files, detects MFO instruction opcodes, forms opcode sequence, trains HMM, and scores test files. After careful analysis of the virus files, 14 MFO instructions were identified (Billar) and corresponding opcodes are collected to produce opcode table. The produced opcode table was used in the process of forming opcode sequence. As the table is precise and concise, it helps to improve overall efficiency significantly.

Our method achieved the primary goal of this work. It completely eliminated the manual process involved in the disassembling phase, reduced the total running time by 70%, and significantly improved overall efficiency.

8. Future Work

We extracted only the code segment from the executables. It can be expanded to include data segment which will be challenging as it includes data in addition to the function codes we are interested. Also, our opcode table consists of fewer number of 1-byte opcodes that are searched indiscriminately resulting in ~3% false positives. It can be further improved by analyzing the virus assembly files and determining conditions to identify 1-byte opcodes.

Appendix A: Bibliography

Billar, D. Statistical Structures: Fingerprinting Malware for Classification and Analysis
http://cs.wellesley.edu/~dbilar/papers/Bilar_OpcodeDistribution_ICGeS07.pdf

Jordan, M. (2002). Anti-Virus Research - Dealing with Metamorphism, Virus Bulletin
http://ca.com/us/securityadvisor/documents/collateral.aspx?cid=48051

Kolter, J.Z., & Maloof,M.A.(2004) Learning to Detect Malicious Executables in the Wild, In ACM Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 470–478

Matt Pietrek (2002) An In-Depth Look into the Win32 Portable Executable File Format
Part 1: http://msdn2.microsoft.com/en-us/magazine/cc301805.aspx
Part 2: http://msdn2.microsoft.com/en-us/magazine/cc301808.aspx

Microsoft Portable Executable and Common Object File Format Specification, Revision 8.1, February 2008

Mohammed, M. (2003) Zeroing in on metamorphic computer viruses, masters thesis, University of Louisiana at Lafayette
www.cacs.louisiana.edu/~arun/papers/moin-mohammed-thesis-dec2003.pdf

Stamp, M. (2004) A revealing introduction to hidden Markov models
www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf

Stamp, M. (2006). Information Security: Principles and Practice, Wiley-Interscience

Szor, P. (2005). The Art of Computer Virus Research and Defense, Addison-Wesley

Szor, P., & Ferrie, P. (2001). Hunting for metamorphic, Symantec Security Response
enterprisesecurity.symantec.com/PDF/metamorphic.pdf

Perriot, Ször, P., & Ferrie, P. (2002)Striking Similarites: Win32/Simile and Metamorphic

Turner, D. et al.,(2008). Symantec Global Internet Security Threat Report Trends for July–December 07, Volume XII

Wong, W. (2006). Analysis and detection of metamorphic computer viruses, masters thesis, Department of Computer Science, San Jose State University www.cs.sjsu.edu/faculty/stamp/students/Report.pdf

Appendix B: Converged HMM Matrices

Table B- 1. Coverged HMM Matrices for N = 2 and Test Set 0

	N=2, M=14, T=65538		
I		1.00000000000000	0.00000000000000
A		0.97529089931559	0.02470910068450
		0.07294422965863	0.92705577034146
B	call	0.11056864271285	0.01032365432858
	and	0.01145967815425	0.00000000000000
	add	0.18800057580368	0.00039631434332
	mov	0.21328078353347	0.95689813171848
	cmp	0.03301040979904	0.00000000000000
	jz	0.06610252853323	0.00000000000000
	lea	0.02880279179210	0.03238189960963
	retn	0.06832909701597	0.00000000000000
	jnz	0.03691201145227	0.00000000000000
	jmp	0.04826955343759	0.00000000000000
	push	0.12836473711460	0.00000000000000
	pop	0.02806701922270	0.00000000000000
	xor	0.02414499033569	0.00000000000000
	test	0.01468718109252	0.00000000000000

Table B- 2. Coverged HMM Matrices for N = 2 and Test Set 1

	N=2, M=14, T=65637		
I		0.00000000000000	1.00000000000000
A		0.99023469357353	0.00976530642649
		0.00517079384893	0.99482920615105
B	call	0.04138172849122	0.10732144082279
	add	0.03279131576133	0.19815336386213
	cmp	0.00713632659984	0.03265250124626
	jz	0.00000000000000	0.07525014019944
	lea	0.03353125642709	0.02783380289300
	mov	0.78330672606554	0.20080176000483
	xor	0.00000000000000	0.02768944552793
	jnz	0.00000000000000	0.04211587933240
	jmp	0.00000000000000	0.05519274352290
	pop	0.00000000000000	0.03204064411089
	push	0.06899283786000	0.11016834858514
	retn	0.03285980879498	0.06120574059315
	and	0.00000000000000	0.01328628016508
	test	0.00000000000000	0.01628790913408

Table B- 3. Coverged HMM Matrices for N = 3 and Test Set 0

	N=3, M=14, T=65538			
I		0.00000000000000	1.00000000000000	0.00000000000000
A		0.70346079377318	0.29653920622679	0.00000000000000
		0.07914393658555	0.91445966538834	0.00639639802613
		0.00962002852435	0.00000000000000	0.99037997147564
B	call	0.18044259546229	0.08754297795188	0.04019049649330
	and	0.00448429829006	0.01540292005154	0.00000000000000
	add	0.03860749464897	0.24155284859017	0.03234418078082
	mov	0.02469929232564	0.25302357242172	0.78967988859470
	cmp	0.00315211876847	0.04250330460559	0.00707130426832
	jz	0.01007609703007	0.09340170221697	0.00000000000000
	lea	0.01364679545996	0.03187212163422	0.03343588672210
	retn	0.07481661944953	0.05715118751145	0.03145387119567
	jnz	0.00000000040173	0.05377853338374	0.00000000000000
	jmp	0.01349456618584	0.06643437646140	0.00000000000000
	push	0.49741215965854	0.00000000000000	0.06582437194508
	pop	0.13409826140688	0.00222230493117	0.00000000000000
	xor	0.00479799349001	0.03379418116171	0.00000000000000
	test	0.00027170742202	0.02131996907845	0.00000000000000

Table B- 4. Coverged HMM Matrices for N = 3 and Test Set 1

	N=3, M=14, T=65637			
I		1.00000000000000	0.00000000000000	0.00000000000000
A		0.70346079377318	0.29653920622679	0.00000000000000
		0.07914393658555	0.91445966538834	0.00639639802613
		0.00962002852435	0.00000000000000	0.99037997147564
B	call	0.08867756472923	0.17378980549424	0.03999032275435
	add	0.24121738466127	0.03794667554559	0.03246059846073
	cmp	0.04101184800289	0.00177478727713	0.00711025515134
	jz	0.09241896276235	0.01063289662025	0.00000000000000
	lea	0.03188667648912	0.01347452830663	0.03362093725516
	mov	0.25488566270350	0.02044655468201	0.78988948068637
	xor	0.03378862068065	0.00468934596868	0.00000000000000
	jnz	0.05332496381131	0.00025874665049	0.00000000000000
	jmp	0.06621929897186	0.01336997690319	0.00000000000000
	pop	0.00276033294822	0.13469898314905	0.00000000000000
	push	0.00000000000000	0.50857789021476	0.06538067722100
	retn	0.05831881713061	0.07320156127166	0.03154772847102
	and	0.01562700800442	0.00433427248924	0.00000000000000
	test	0.01986285910459	0.00280397542706	0.00000000000000

Table B- 5. Coverged HMM Matrices for N = 4 and Test Set 0

	N=4, M=14, T=65538				
I		1.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000
A		0.44045057521127	0.53973027295562	0.00000000000000	0.01981915183314
		0.62792547362477	0.18618315452008	0.06922733549534	0.11666403635980
		0.00000000000000	0.00000000000000	0.93607368937815	0.06392631062179
		0.34692889141062	0.00000000000000	0.00000000000000	0.65307110858937
B	call	0.10436800534989	0.11822021256511	0.01027314932301	0.11612084862888
	and	0.01349976044783	0.01245313695031	0.00000000000000	0.00522868141933
	add	0.12144044526825	0.38719748586197	0.00028310901995	0.03441523285621
	mov	0.43023595328126	0.00000428494713	0.94952037682505	0.02275571968959
	cmp	0.06604882783370	0.00000000000000	0.00000000000000	0.00688387326382
	jz	0.05917203374938	0.11726839434507	0.00000000000000	0.00244250914821
	lea	0.02271307888903	0.04655973578208	0.03992336483199	0.00178900787395
	retn	0.00000000261127	0.19447448317301	0.00000000000000	0.03303669112864
	jnz	0.04796843467275	0.04358686152385	0.00000000000000	0.00038036414313
	jmp	0.05028195162244	0.06789037513264	0.00000000000000	0.01315996772862
	push	0.00000000000000	0.00000000000000	0.00000000000000	0.64946985760290
	pop	0.01372077231028	0.00000000000000	0.00000000000000	0.10874111229134
	xor	0.03990062711514	0.01234502971883	0.00000000000000	0.00557613422539
	test	0.03065010684881	0.00000000000000	0.00000000000000	0.00000000000000

Table B- 6. Coverged HMM Matrices for N = 4 and Test Set 1

	N=4, M=14, T=65637				
I		0.00000000000000	0.00000000000000	0.00000000000000	1.00000000000000
A		0.80883316869008	0.06320914331696	0.03559313923522	0.09236454875772
		0.27355240608643	0.64178389969937	0.00000000000000	0.08466369421419
		0.00000000000000	0.05901229599492	0.94098770400506	0.00000000000000
		0.15518518785122	0.06191124864767	0.00000000000000	0.78290356350103
B	call	0.12685046243687	0.11199929673836	0.00853612550654	0.07280030502653
	add	0.23323765834646	0.01851037291423	0.00000000000000	0.20897318307602
	cmp	0.00051981304233	0.02147360786514	0.00000000000000	0.09134592516609
	jz	0.00951432313309	0.01066382234148	0.00000000000000	0.19608849354750
	lea	0.04699127266348	0.00000000000000	0.02799478588275	0.02116523853286
	mov	0.41101127083640	0.00407452652008	0.96346908861070	0.05420224430150
	xor	0.01041078461638	0.00447553751546	0.00000000000000	0.05970641577765
	jnz	0.00847781584294	0.00075510653538	0.00000000000000	0.10764786651215
	jmp	0.02473783521184	0.01120702111540	0.00000000000000	0.11054120772494
	pop	0.00693284728153	0.12426754287804	0.00000000000000	0.00000000024898
	push	0.00000000000000	0.65300481051350	0.00000000000000	0.00000000000000
	retn	0.10885509567445	0.03444266696595	0.00000000000000	0.01633128223171
	and	0.00772252853900	0.00335420883476	0.00000000000000	0.0230884120800
	test	0.00473829237528	0.00177147926222	0.00000000000000	0.03810942577395

Table B- 7. Coverged HMM Matrices for N = 5 and Test Set 0

	N=5, M=14, T=65538					
I		0.00000000000000	0.00000000000000	1.00000000000000	0.00000000000000	0.00000000000000
A		0.9405214315319	0.00000000000000	0.00000000000000	0.05947856846809	0.00000000000000
		0.1129644367010	0.70298632458623	0.09892001655545	0.05531723665474	0.02981198550257
		0.0000000000000	0.00000000000000	0.79609378295742	0.06543191922219	0.13847429782046
		0.0000000000000	0.00000000000000	0.12417740830274	0.61586286167349	0.25995973002377
		0.0000000000000	0.16901480782446	0.10904631245805	0.06774692082023	0.65419195889725
B	call	0.01056237299509	0.00000000151958	0.08555187476068	0.09267881503836	0.20923462951049
	and	0.00000000000000	0.01524718919793	0.02162219579171	0.00509363523682	0.00058701342624
	add	0.00000000000000	0.45736257029521	0.21841193435418	0.03472486592215	0.08567076329304
	mov	0.96541995985256	0.32951107919369	0.06029850748439	0.00000000000000	0.49279248754508
	cmp	0.00000000000000	0.00000000000000	0.07921147534479	0.01712096169939	0.00492726978894
	jz	0.00000000000000	0.01951138082775	0.17251284419881	0.00594289633036	0.00000000000000
	lea	0.02401766715236	0.09208425527757	0.04100052758162	0.00191262451608	0.00436475670059
	retn	0.00000000000000	0.04667059388427	0.02742919903069	0.02172002194133	0.15532636270807
	jnz	0.00000000000000	0.00000000000000	0.10327137331020	0.00000000000000	0.00000000000000
	jmp	0.00000000000000	0.01529291004272	0.09963315594656	0.00798581331095	0.02856757841450
	push	0.00000000000000	0.00000000000000	0.00000000000000	0.68180914402532	0.00000000000000
	pop	0.00000000000000	0.00000000000000	0.00131185622475	0.12620848711047	0.01281949026622
	xor	0.00000000000000	0.01730019345761	0.05199841067901	0.00480273486877	0.00570964834679
	test	0.00000000000000	0.00701982630365	0.03774664529266	0.00000000000000	0.00000000000000

Table B- 8. Coverged HMM Matrices for N = 5 and Test Set 1

	N=5, M=14, T=65637					
I		0.00000000000000	1.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000
A		0.64262392510210	0.13305877596495	0.00000000000000	0.00000000000000	0.22431729893291
		0.07029776430647	0.83694713112539	0.00000000000000	0.00000000000000	0.09275510456816
		0.06593363140192	0.28377975032280	0.00686837680517	0.64341824147011	0.00000000000000
		0.06153810783160	0.00000000000000	0.00000000000000	0.93846189216840	0.00000000000000
		0.05127506960621	0.07036255303579	0.0573136406037	0.00000000000000	0.82104873675419
B	call	0.11414312762067	0.09178780664076	0.00000000000000	0.00000000000000	0.1212773367205
	add	0.01839111716720	0.22139415067586	0.00000000000000	0.00000000000000	0.2162526399498
	cmp	0.01902016801018	0.06980192768685	0.00000000000000	0.00000000000000	0.0010256620639
	jz	0.00531781714927	0.15392450397864	0.00000000000000	0.00000000000000	0.0082371275539
	lea	0.00107196862260	0.03455297850306	1.00000000000000	0.00000000000000	0.0000000000000
	mov	0.00170007141622	0.09216764637624	0.00000000000000	1.00000000000000	0.51117361573010
	xor	0.00372769157841	0.05030631107538	0.00000000000000	0.00000000000000	0.0078690458847
	jnz	0.00000000014669	0.09281179061727	0.00000000000000	0.00000000000000	0.0000000000000
	jmp	0.00986206845249	0.09799374859572	0.00000000000000	0.00000000000000	0.01665250939040
	pop	0.12110891030325	0.00168433880047	0.00000000000000	0.00000000000000	0.00865846380033
	push	0.66134786029249	0.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000
	retn	0.04108521449830	0.04076323009906	0.00000000000000	0.00000000000000	0.09930854452460
	and	0.00272393218346	0.02057069727973	0.00000000000000	0.00000000000000	0.00652867029279
	test	0.00050005255879	0.03224086967097	0.00000000000000	0.00000000000000	0.00301638408870

Table B- 9. Coverged HMM Matrices for N = 6 and Test Set 0

	N=6, M=14, T=65538
I	0.00000000000000 1.00000000000000 0.00000000000000 0.00000000000000 0.00000000000000 0.00000000000000

A	0.74654207880452 0.06015696292670 0.04465832656218 0.11643327108490 0.03220936062184 0.00000000000000
	0.02060514556199 0.78892780154198 0.05844976057459 0.00000000000000 0.13201729232145 0.00000000000000
	0.00000000000000 0.09872443224713 0.56509608986430 0.00000000000000 0.33617947788853 0.00000000000000
	0.00000000000000 0.24213284361941 0.00710552510345 0.00000000000000 0.02958095288687 0.72118067839026
	0.17068522563537 0.11758994250491 0.09298246310421 0.00000000000000 0.61874236875556 0.00000000000000
	0.00000000000000 0.00000000000000 0.05739856445345 0.00000000000000 0.00000000000000 0.94260143554657

B	call 0.00000000000000 0.07963980903354 0.04474903481381 0.00000000000000 0.2726749917285 0.01054750258092 and 0.01379599847082 0.02259540505228 0.00558907307631 0.00000000000000 0.0004359338601 0.00000000000000 add 0.38675138566697 0.21282350028897 0.03986379770185 0.27309104766597 0.0739491617351 0.00000000000000 mov 0.42371179827650 0.05194811358821 0.00000000000000 0.00000000000000 0.4178819850247 0.97785286398327 cmp 0.00000000000000 0.08312910828087 0.01519714332158 0.00000000000000 0.0113982201845 0.00000000000000 jz 0.02253365781221 0.17868171055175 0.00200425425423 0.00000000000000 0.0083749572690 0.00000000000000 lea 0.00000000000000 0.03668926127124 0.00000000000000 0.72530432647419 0.0203587104885 0.01159963343582 retn 0.08894246264290 0.02453665903854 0.00000000000000 0.00000000000000 0.1450387602515 0.00000000000000 jnz 0.00000000000000 0.11272898499177 0.00000000000335 0.00000000000000 0.0000000000000 0.00000000000000 jmp 0.03388174710212 0.10379847545280 0.00779811304391 0.00160462585985 0.0197130677633 0.00000000000000 push 0.00000000000000 0.00000000000000 0.75945600474962 0.00000000000000 0.0000000000106 0.00000000000000 pop 0.00000000000000 0.00002764719045 0.11947591062144 0.00000000000000 0.0278200833764 0.00000000000000 xor 0.02053220621472 0.05507378660489 0.00586666841388 0.00000000000000 0.0023540626160 0.00000000000000 test 0.00985074381389 0.03832753865470 0.00000000000000 0.00000000000000 0.0000000656915 0.00000000000000

Table B- 10. Coverged HMM Matrices for N = 6 and Test Set 1

	N=6, M=14, T=65637						
I		1.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000	0.00000000000000
A		0.79902575937822	0.06884299516905	0.00000000000000	0.00395962775884	0.06656516435248	0.06160645334139
		0.00004962226266	0.05402429439583	0.50987354539271	0.14656884949096	0.15294718165373	0.13653650680411
		0.00000000000000	0.00000000000000	0.94315591906849	0.00000000000000	0.00000000000000	0.05684408093152
		0.21315790437331	0.00000000000000	0.00000000000000	0.49708269885924	0.21023608262221	0.07952331414521
		0.00001369512679	0.00000000000000	0.00000000000000	0.93787525245430	0.06052839468865	0.00158265773027
		0.27132909596639	0.00000000000000	0.00000000000000	0.02130601346098	0.06433438422900	0.64303050634357
B	Call	0.13186949046900	0.00000000000000	0.01033245540491	0.0960162971704	0.0414220481090	0.11261960836740
	Add	0.22409281811031	0.33135053413181	0.00000000000000	0.1833326392870	0.2675745644057	0.01850064871170
	Cmp	0.00325621138480	0.00000000000000	0.00000000000000	0.0000000000000	0.2670579038232	0.01856606136724
	Jz	0.01673161748286	0.00000000000000	0.00000000000000	0.2677942538386	0.0035045087466	0.00792665991814
	Lea	0.00529078865858	0.66864946586819	0.01319064156814	0.0182121989531	0.0567012442008	0.00065911286116
	Mov	0.43823259448582	0.00000000000000	0.97647690302694	0.0468122413825	0.0748684472089	0.00037493358845
	Xor	0.01264762393210	0.00000000000000	0.00000000000000	0.0294594620569	0.1089340239043	0.00559008139898
	Jnz	0.00000000000000	0.00000000000000	0.00000000000000	0.1741889487909	0.0000000000000	0.00262644652140
	Jmp	0.03387824544811	0.00000000000000	0.00000000000000	0.1223503779078	0.0409882716703	0.01186122068469
	Pop	0.00890641093102	0.00000000000000	0.00000000000000	0.0000000000000	0.0053662408508	0.11957216373433
	Push	0.00000000000000	0.00000000000000	0.00000000000000	0.0000000000000	0.0000000000791	0.66099864418808
	Retn	0.11157074655939	0.00000000000000	0.00000000000000	0.0308303051582	0.0137436214224	0.03607282455750
	And	0.01034303858470	0.00000000000000	0.00000000000000	0.0286673453200	0.0000000000000	0.00333377158292
	Test	0.00318041395330	0.00000000000000	0.00000000000000	0.0023359301341	0.1198391255784	0.00129782251794

Appendix C: HMM Testing Results

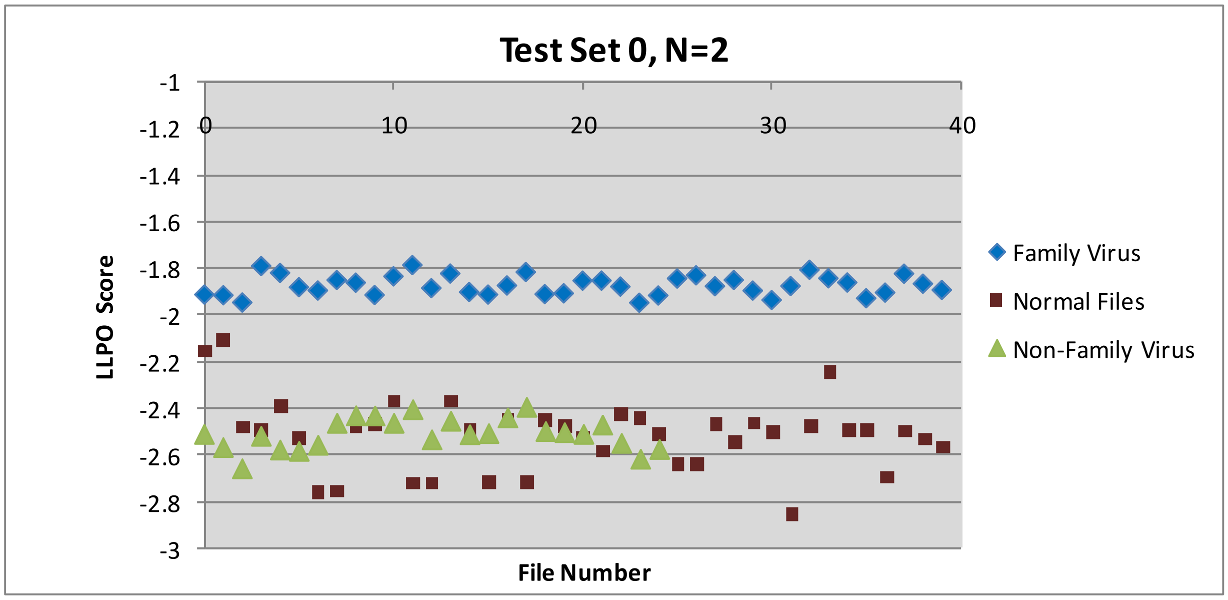


Figure C- 1 LLPO Scores of Family Virus, Normal Files and Non-Family Virus for test set 0 and N = 2
Source: Author's research

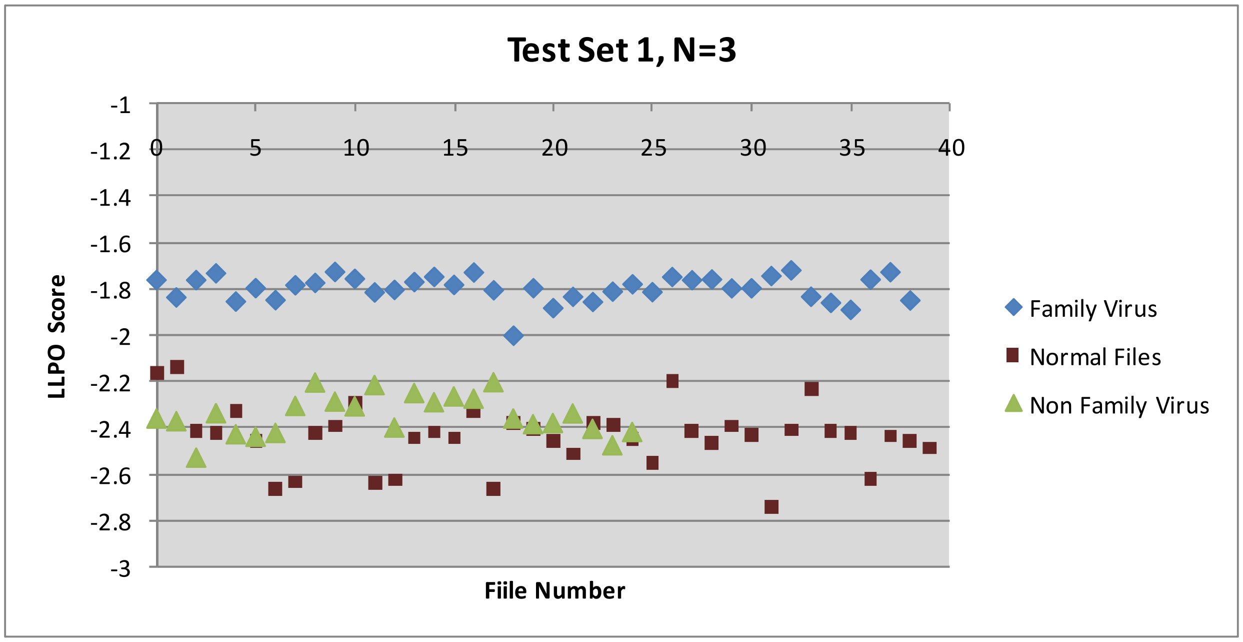


Figure C- 2 LLPO Scores of Family Virus, Normal Files and Non-Family Virus for test set 1 and N = 3
Source: Author's research

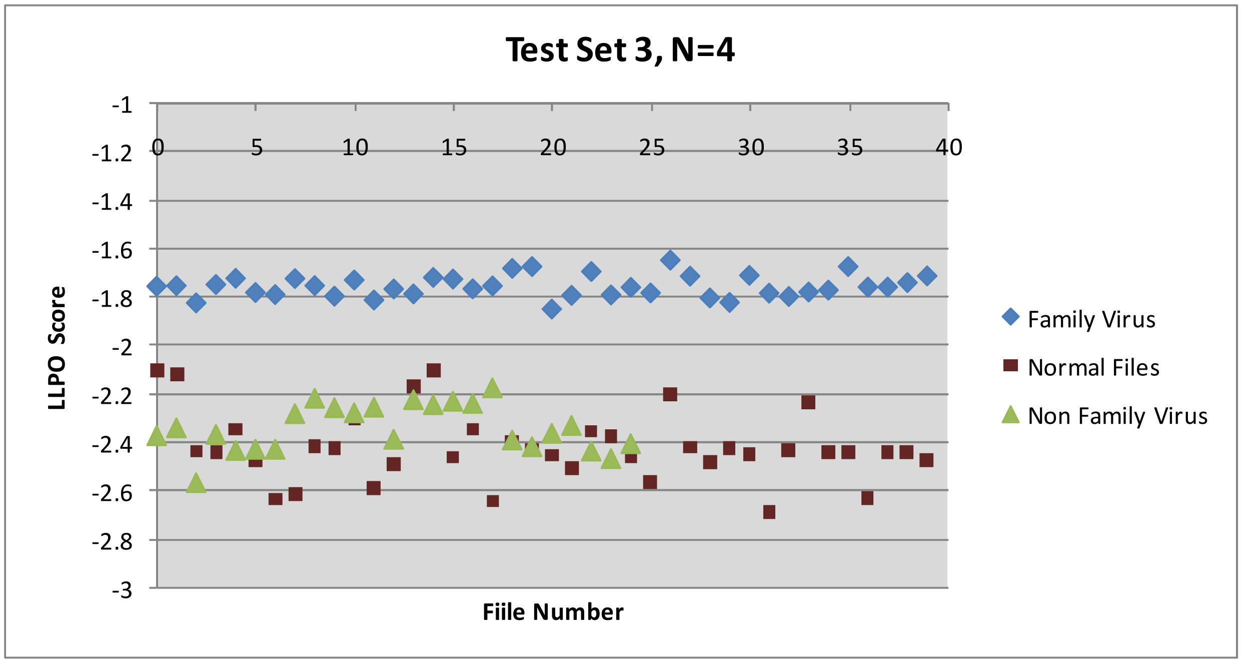


Figure C- 3 LLPO Scores of Family Virus, Normal Files and Non-Family Virus for test set 3 and N = 4
Source: Author's research

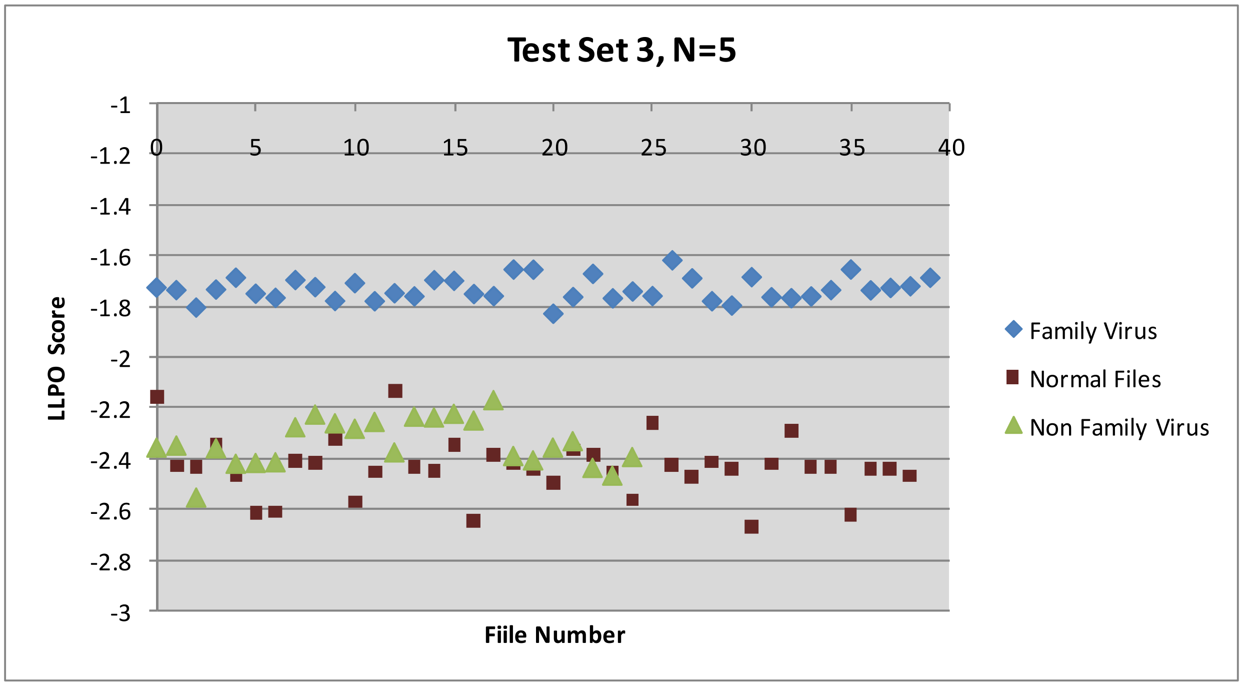


Figure C- 4 LLPO Scores of Family Virus, Normal Files and Non-Family Virus for test set 3 and N = 5
Source: Author

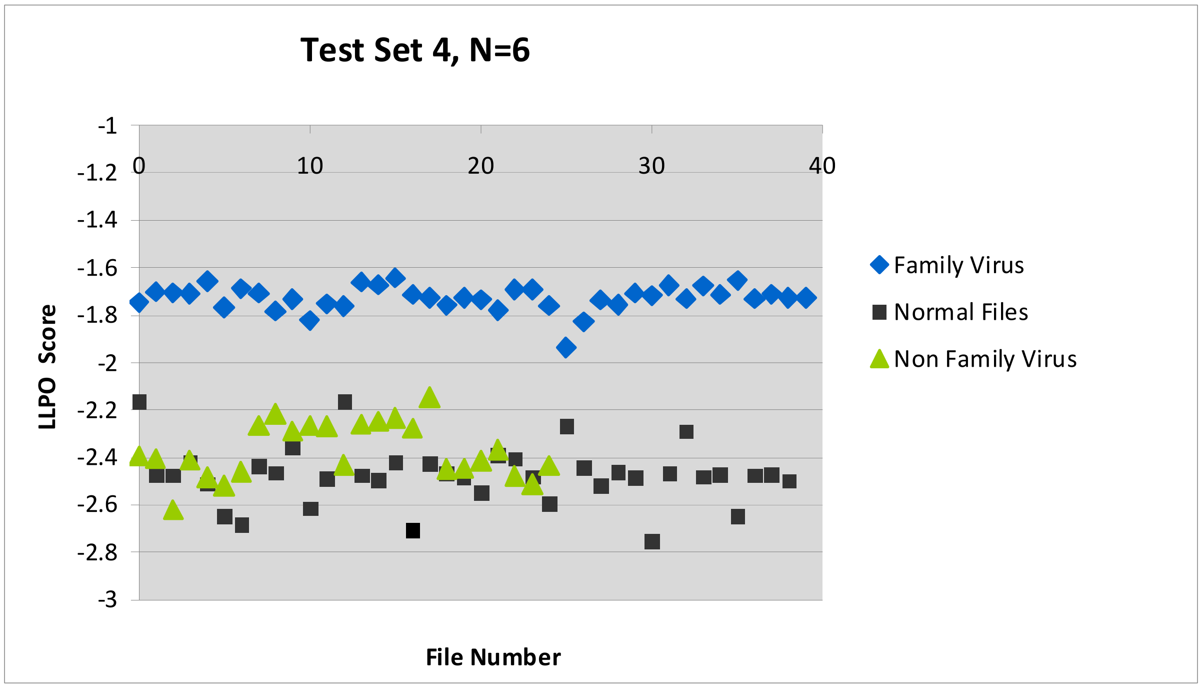


Figure C- 5 LLPO Scores of Family Virus, Normal Files and Non-Family Virus for test set 4 and N = 6
Source: Author's research