APPROXIMATE DISASSEMBLY

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in Computer Science

By

Dhivyakrishnan Radhakrishnan

May 2009

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled

APPROXIMATE DISASSEMBLY

by

Dhivyakrishnan Radhakrishnan

Dr. Mark Stamp,	Department of Computer Science	Date

Dr. Robert Chun,	Department of Computer Science	Date

Dr. Araya Agustin,	Department of Computer Science	Date

APPROVED FOR THE UNIVERSITY

Associate Dean	Office of Graduate Studies and Research	Date

ABSTRACT

APPROXIMATE DISASSEMBLY

by Dhivyakrishnan Radhakrishnan

For the past two decades, computer viruses have been a constant security threat. A computer virus is a type of malware that may damage computer systems by destroying data, crashing the system, or through other malicious activity. Among the different types of viruses, metamorphic viruses are one of the most difficult to detect since such viruses change their internal structures with each mutation, making signature-based detection infeasible. Many construction kits are available that can be used to easily generate metamorphic strains of any given virus.

Previous work has shown that metamorphic viruses are detectable using Hidden Markov Models (HMM). In such an HMM-based approach, instruction opcodes are observed and a model is trained to detect a given virus family. These instruction opcodes are obtained by disassembling the binary executable file. However, the disassembling process is time-consuming, making the process impractical. In this project, we develop and demonstrate a technique to derive an approximate opcode sequence directly from the executable file, which, in general, reduces the time required as compared to a standard disassembly process.

Table of Contents

List of Figures

LIST OF TABLES

1. Introduction

A computer virus is a software program that infects a computer system without the user's knowledge. The virus replicates itself into a computer system and affects the root functions of various programs. Computer viruses can cause many types of damage such as deletion of files, reformatting of hard drive, system slow down, or connectivity issues. Viruses usually spread through storage devices, computer networks or the Internet.

Over the past two decades, viruses have evolved rapidly. They have caused crucial financial damage to businesses and organizations. For example, in 2000, the virus "Love Letter" affected 10 million computers causing damages on the order of $10 billion [2].

With the advent and growth of viruses, a variety of anti-virus tools have evolved to detect and manage the threats. These tools use a wide range of techniques to identify and remove viruses. However, there is no single antivirus software that protects a computer system from all viruses [1].

Viruses continue to evolve in an effort to stay ahead of advances in anti-virus software. One of the most advanced classes of viruses are the metamorphic viruses which change their structure with every mutation. They are harder to detect since the mutations are not close to their parents [7]. However, such viruses can be detected using a Hidden Markov Model (HMM) [9].

Most of the virus detection techniques use a disassembler to analyze the assembly code of the virus [7]. Our research focuses on improving the speed of the standard disassembler. This report is organized as follows. Section 2 contains a brief description of various types of viruses. Section 3 gives an overview of HMMs. Section 4 gives a

detailed explanation of the disassembly process. Section 5 explains our new method that reduces the disassembly time as compared to a standard disassembly process. Section 6 covers our results.

2. Types of viruses

There are several classes of viruses and each one of them affects the computer system in a different way. The following are some of the most popular computer viruses[7]:

- Macro virus
- Boot sector virus
- File infector virus
- Encrypted virus
- Polymorphic virus
- Metamorphic virus

2. 1 Macro viruses

Macro virus is a virus that is embedded as part of a document or any other application such as Microsoft Office. These viruses may inflict harm to other documents available on the system when opened. Macro viruses usually spread through mail attachments or file transfers, and now they also occur in web pages. Recent versions of the Windows operating system include antivirus tools that disable macro viruses by default [4]. A common example of a macro virus is the Melissa virus of 1999 [5].

2.2 Boot sector viruses

A Boot sector virus is a virus that causes damage to the boot sector of a hard drive, CD/DVD or floppy disks. Once the computer boots, the boot sector virus remains in the memory and infects floppy and other bootable media when they are being accessed. Boot

sector viruses have become relatively uncommon due to the rare use of floppy disks in recent times [4]. A fine example of a boot sector virus is the Michelangelo virus of 1991 [5].

2.3 File infector viruses

A file infector virus is a virus that infects files in a computer system. When an infected file is executed, the virus replicates itself to other applications in the system. File infector viruses are usually common among .exe and.com files. File infector viruses cause system hang-ups and slow performance [4]. An example of a file infector virus is the Cleevix virus of 2006 [5].

2.4 Encrypted viruses

An encrypted virus is a type of virus where the virus code is encrypted to avoid signature detection. The encrypted virus code changes with each infection by using a different encryption key but uses the same decryption key. Hence, it is still possible to detect encrypted viruses based on the decryption key [6].

2.5 Polymorphic viruses

Polymorphic viruses are an extension of encrypted viruses where the decryption key is different with each infection. But the decrypted virus code is the same regardless of the different decryption key. Antivirus programs that incorporate code emulation techniques can detect polymorphic viruses [6].

2.6 Metamorphic viruses

Metamorphic viruses are more powerful than polymorphic viruses. Unlike polymorphic viruses, they do not decrypt to the same virus code. Metamorphic viruses change the structure of their code without affecting the functionality. The changed code is recompiled to create a virus executable that looks different from the original [6]. Such modification is achieved by using several metamorphic techniques that are explained in Section 2.6.1. Figure 1 shows multiple shapes of a metamorphic virus body.

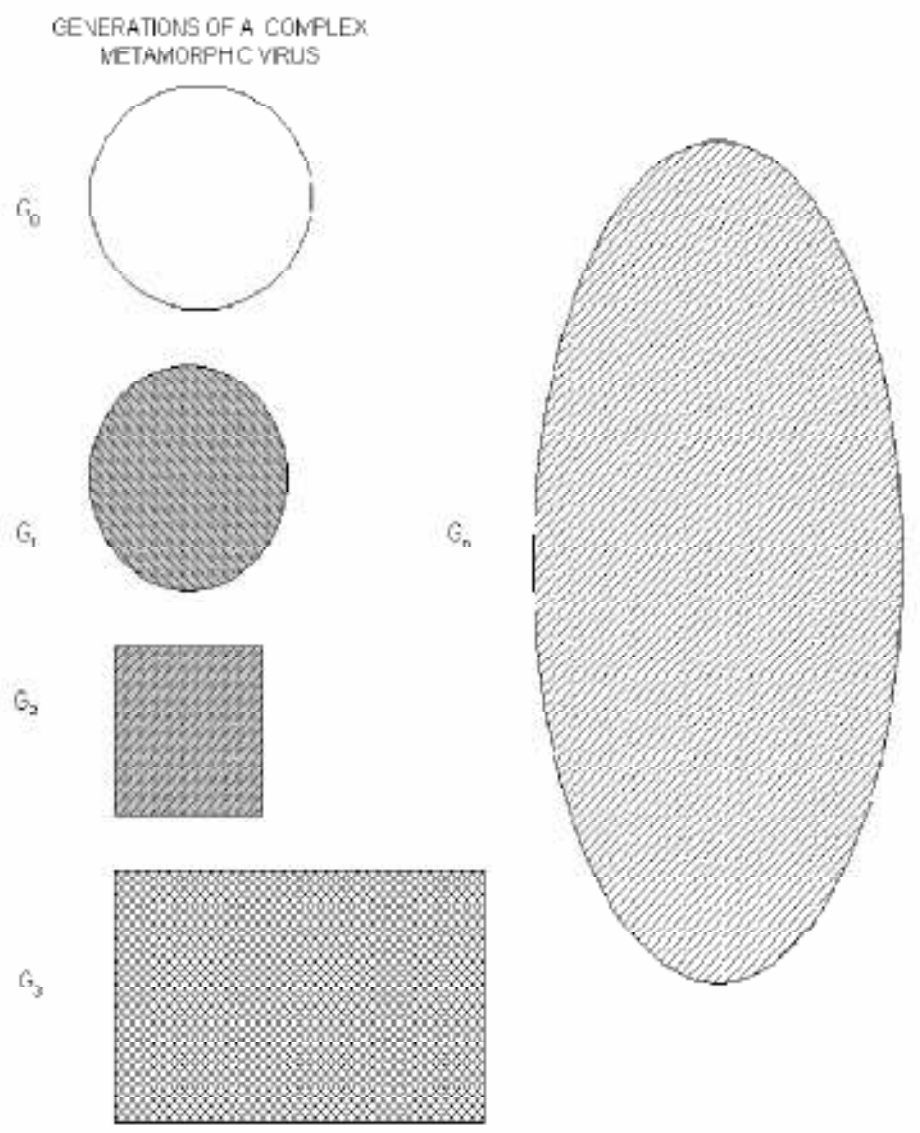

Figure 1: Multiple shapes of a metamorphic virus body [7]

2.6.1 Different kinds of metamorphism

Metamorphic viruses use different kinds of techniques to make the new infection look different from the existing one. The following are some of the commonly used metamorphic techniques explained by Szor in [7]:

- Garbage Insertion
- Register Usage Exchange
- Instruction Replacement
- Subroutine reordering

2.6.1.1 Garbage Insertion

This is a simple morphing technique that is widely used by metamorphic viruses. It inserts garbage or jump instructions into the code, which occupy space but do not affect the functionality of the code. These viruses are harder to analyze because unwanted or garbage instructions exist in larger quantities. These instructions might even introduce new errors in the program. Win32/Evol is a virus that inserts garbage instructions between core instructions [7].

2.6.1.2 Register Usage Exchange

Register usage exchange is another simple technique that uses different registers for new infections but continues to use the same virus code. However, such viruses can be

detected by using wild card strings. W95/Regswap is a virus that uses the register usage exchange technique [7].

2.6.1.3 Instruction Replacement

Instruction replacement technique replaces a set of instructions in a virus with an equivalent set. Instruction replacement also has no impact on the functionality of the code. Win95/Bistro is a virus that uses the instruction replacement technique [7].

2.6.1.4 Subroutine Reordering

In this technique, the subroutines are reordered and branch instructions are used to connect them to maintain the functionality. The order of subroutines is different for each infection. If there are n subroutines in a given virus, then this technique creates n! variants of that virus. Win32/Ghost is a virus that uses the subroutine reordering technique [7].

3. HMM and metamorphic virus detection

An HMM is a state machine with hidden states where the transitions between states have a fixed probability. The external observer can only see a set of observations which depend (probabilistically) on the hidden states. Once a HMM has been trained with a set of observation sequences, the model has the ability to detect similar sequences in a given input. HMM are well suited for statistical pattern analysis [9].

Consider the following notations of HMM used in [8]:

T = the length of the observation sequence

N = the number of states in the model

M = the number of observation symbols

$Q = \{q_0, q_1, \ldots, q_{N-1}\}$ = the states of the Markov process

$V = \{0, 1, \ldots, M-1\}$ = set of possible observations

A = the state transition probabilities

B = the observation probability matrix

π = the initial state distribution

$O = (O_0, O_1, \ldots, O_{T-1})$ = observation sequence.

$X_i =$ Hidden state

Figure 2 illustrates a generic HMM with the notation given above.

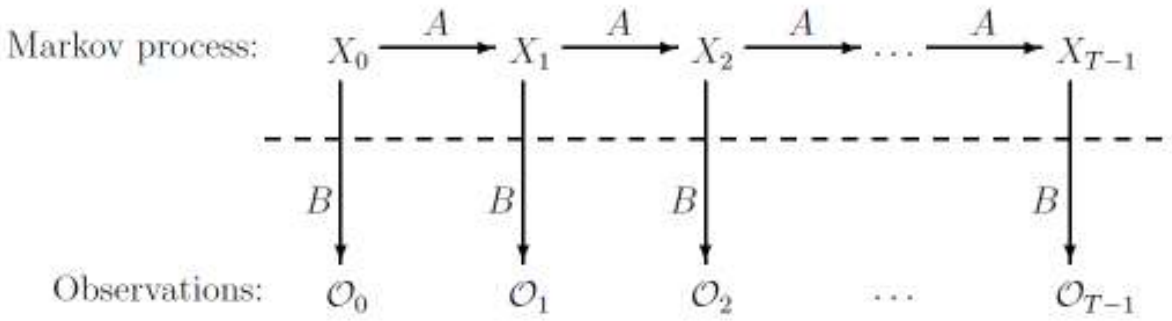

Figure 2: Hidden Markov Model [8]

The matrix A = $\{a_{ij}\}$ is of size N × N with

$$a_{ij} = P(\text{state } q_j \text{ at } t + 1 \mid \text{state } q_i \text{ at } t)$$

and A is row stochastic. Note that the probabilities a_{ij} are independent of t.

The matrix B = $\{bj(k)\}$ is of size N ×M with

$$b_j(k) = P(\text{observation } k \text{ at } t \mid \text{state } q_j \text{ at } t).$$

Thus, an HMM is denoted by (π, A, B), where the matrices π, A and B are row stochastic.

Metamorphic viruses have similarities in their code structure, in spite of their ability to mutate with each infection. These similarities help HMM to detect metamorphic viruses. In the HMM, the virus characteristics are the states and the instruction opcode sequence is the observation set. Stamp et al in [9] used HMM to detect metamorphic viruses.

The procedure involved in training the HMM is as follows:

1. Collect a set of executable files that belong to the same family
2. Disassemble the collected executable files into assembly code

3. Extract opcode sequence from the assembly code
4. Train the model with the extracted opcode sequence

Figure 3 illustrates the process involved in creating a HMM.

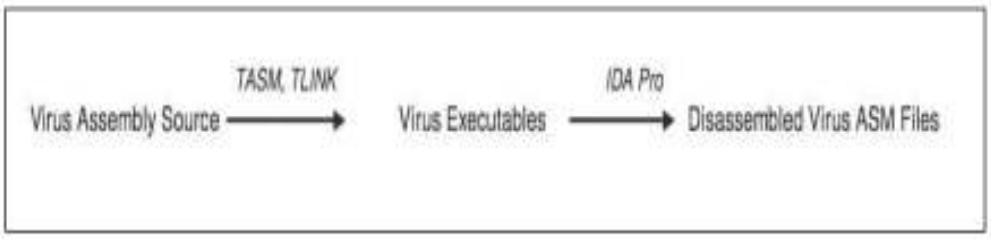

Figure 3: Creating a HMM model (Preprocessing of virus file) [9]

A set of viruses of the same family are assembled into the corresponding set of executable files using an assembler (e.g. TASM, TLINK). Each executable file is then disassembled using a disassembler (e.g. IDA Pro). An opcode sequence is extracted from the disassembled files to create a HMM.

After the creation of HMM, the HMM is tested using a set of files. Some of the files in the set belong to the family of viruses for which the HMM is created and trained for. The HMM processes these files and should give a high score to the virus files of the same family and low score to the others [9].

The next section describes about the disassembly process that plays a crucial role in detecting metamorphic viruses using HMM.

4. Disassembly

Computer programs are human readable programs written in a high-level language like C, C++, etc. These programs are translated into a CPU readable, executable file. The CPU uses these executable files to execute the instructions to perform the required operation. Figure 4 shows how human readable computer programs are converted into CPU readable machine code.

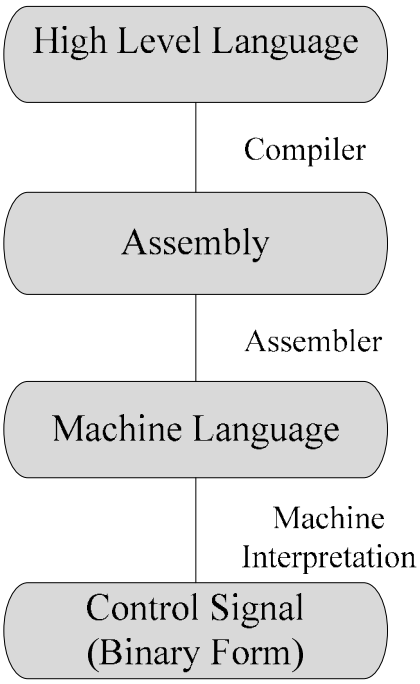

Figure 4: Levels of Data Representation [10]

Disassembly is the process of converting machine language back into assembly language. Figure 5 illustrates the disassembly process.

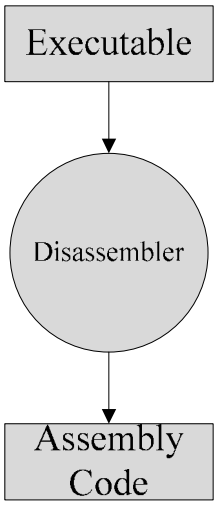

Figure 5: Disassembly Process [10]

Section 4.1 describes details of the assembly language instruction set, by using Intel instruction set as an example. Section 4.2 describes details of the disassembler.

4.1 Intel Instruction Set

Instruction set architecture is a part of computer architecture that acts as an interface between hardware and software. It contains a list of instructions that the processor can understand and execute. The basic types of instructions are as follows:

- arithmetic (add, subtract)
- logic (and, or)
- data (mov, load)
- control flow (call, return)

Figure 6 shows the layout of the computer architecture in which the instruction set architecture lies between software and hardware.

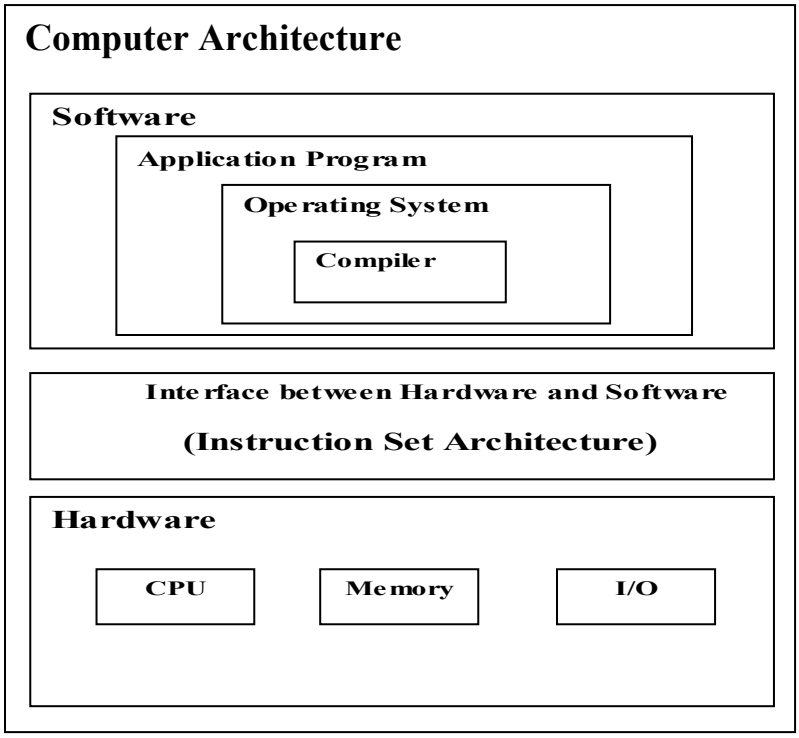

Figure 6: Layout of Computer Architecture [10]

Intel x86 processors use complex instruction set computer (CISC) architecture that comprise large number of variable-length instructions and complex addressing modes. In the Intel instruction set, there are more than 100 instructions and each instruction can be 1 to 17 bytes long [11].

4.1.1 Instruction Format

Instructions in the Intel instruction set start with an optional instruction prefix. The instruction prefix is followed by an opcode field, which can be 1 or 2 bytes. The instruction also includes an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required) [11]. Details of the individual fields in

the instruction are explained in the following subsections. Figure 7 illustrates the format of the instruction.

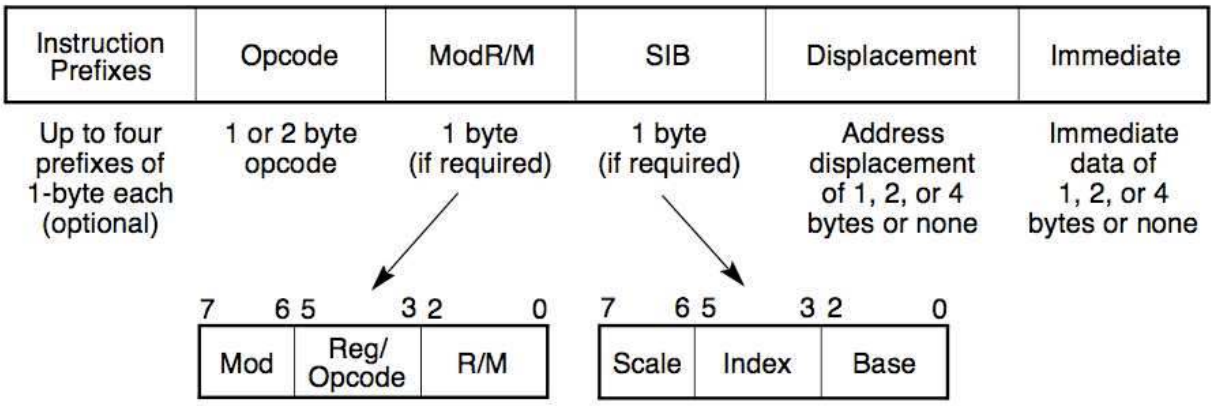

Figure 7: Intel Instruction Format [11]

4.1.1.1 Instruction Prefix

Instruction prefix is divided into four groups. Each group has a set of allowable prefix codes. Each prefix is of one-byte length. An instruction can have 1 prefix from each group with a maximum of 4 prefix bytes. The Intel manual specifies the following four groups of instruction prefixes [11]:

1. Segment override prefix (2EH,36H,3EH,26H,64H,65H) changes the default segment of the instruction
2. Operand-size override prefix (66H) overrides the default operand-size of the instruction. The default size is either 16-bit or 32-bit operand.
3. Address-size override prefix (67H) overrides the default address-size of the instruction. The default size is either 16-bit or 32-bit address.
4. Repeat (F2H,F3H) and Lock prefix(F0H) controls the loop in the string instructions and bus usage of the processors.

4.1.1.2 Opcodes

The primary opcode of the instruction is either 1 or 2 bytes. The opcode field defines a small encoding field, and the encoding field size varies depending on the operation. These encoded fields can determine displacement size, operational direction, conditional code, register encoding or sign extension of the immediate data field in the instruction. Sometimes an additional three-bit field from the ModR/M byte can also be used as part of the opcode [11].

4.1.1.3 ModR/M bytes

The ModR/M byte is the first byte of the addressing-form specifier. This field gives information about which registers or memory locations are to be used by the instruction. The ModR/M byte consists of three fields [11]:

- Mod: The mod field together with the r/m field defines one of the eight registers or one of the twenty-four addressing modes. It is a two-bit length field.
- Register/Opcode: The three-bit length register/opcode field specifies a register or three additional bits of opcode information.
- R/M: The r/m field is a three-bit length field that can either specify a register as an operand or can be used in conjunction with the mod field to encode an addressing mode.

4.1.1.4 SIB bytes

The SIB byte, the second byte of the addressing-form specifier is used to determine the address of the operand's memory based on the following three fields [11]:

- Scale: It is a two-bit length field that specifies the scale factor.
- Index: It is a three-bit length field that specifies the register number of the index register. It is used for index addressing.
- Base: It is a three-bit length field that specifies the register number of the base register. It is used for base addressing.

Figure 8 shows the calculation of operand's address based on scale, index and base value.

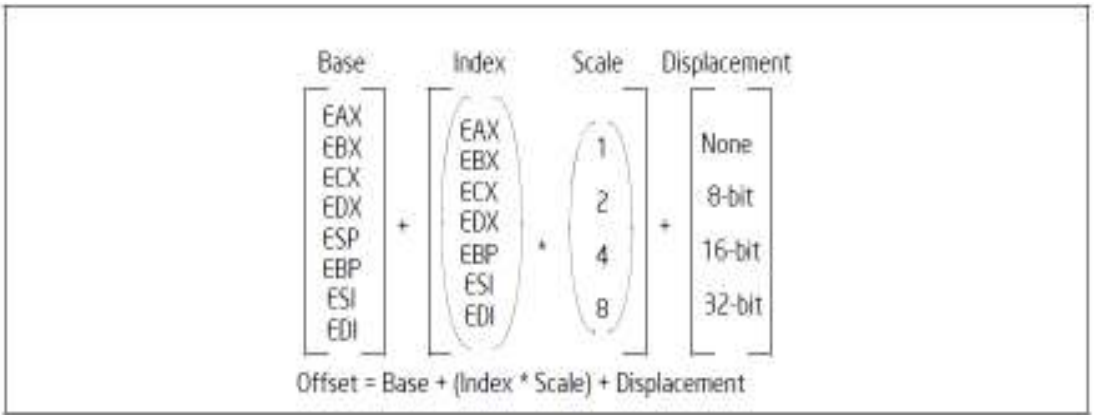

Figure 8: Offset (or Effective Address Computation) in [12]

4.1.1.5 Displacement and Immediate bytes

The displacement field is used when some instructions specify a displacement value that is added to the operands memory address. For example, in the instruction MOV EAX, DWORD PTR DS:[BX+SI+3124], 3124 is the displacement value added to the memory address. The displacement (if required) can be 1, 2, or 4 bytes long. An

immediate field is used when the instructions specify an immediate operand. In the instruction, CMP BX, 20, the immediate operand is 20. The immediate operand can also be 1, 2, or 4 bytes long [11].

4.2 Disassembler algorithm

Figure 9 shows the control flow of the disassembler algorithm.

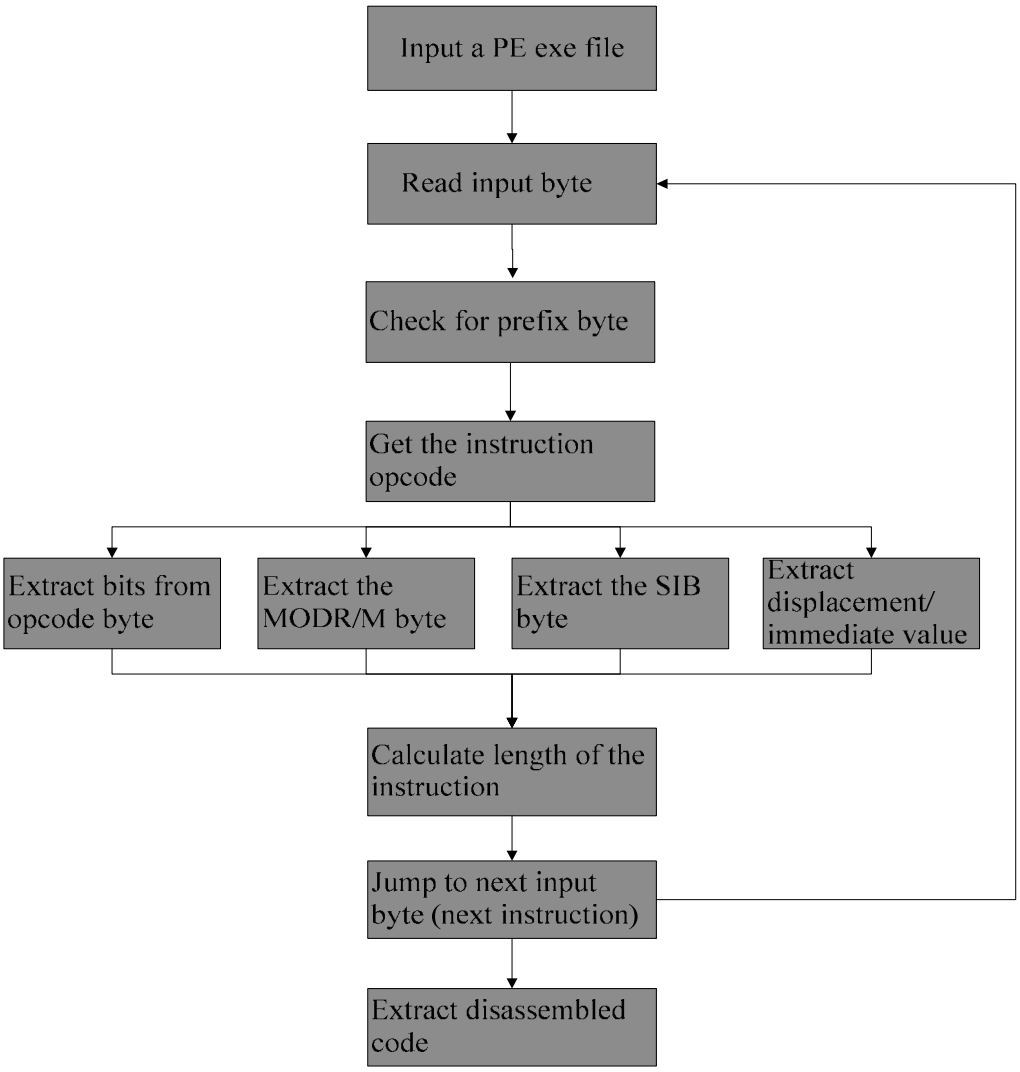

Figure 9: Disassembler Algorithm

The disassembler algorithm takes a PE file as an input and identifies the compiler that built the PE file. The disassembler can achieve a high degree of accuracy in identifying variables, objects and procedures of the target file by knowing the compiler type. The disassembler reads each input byte and determines if it is a prefix byte. If the byte read is a prefix byte, then the next byte is an opcode byte. Then, the operand size of the instruction is determined based on the MOD R/M, SIB, displacement or immediate fields in the following bytes. The disassembler skips the operands and jumps to the next instruction. The disassembler continues the process until it reaches the end of the file.

4.3 Disassembly techniques

There are two main techniques for disassembling the machine code:

1. Linear Sweeping
2. Recursive Traversal

Each of the above technique is explained in detail in the following subsections.

4.3.1 Linear Sweeping

Linear sweeping algorithm extracts the text section from the executable file. The instructions in the text section are sequentially processed until the end of file. Although this algorithm is quite simple and fast, it cannot detect data encoded in the instruction itself [13].

In Figure 10, the sample disassembly file has a bunch of non-executable instructions (0040289D to 004028B5) between executable instructions which cannot be caught by the

linear sweeping algorithm because it considers each and every instruction as executable bytes.

```
00402880  |. 53             PUSH EBX                                ; /Erase
00402881  |. 53             PUSH EBX                                ; |pRect
00402882  |. FF75 CC        PUSH DWORD PTR SS:[EBP-34]              ; |hWnd
00402885  |. FF15 40724000  CALL DWORD PTR DS:[<&USER32.InvalidateRe>; \Invalidat
0040288B  |> 8B45 FC        MOV EAX,DWORD PTR SS:[EBP-4]            ;  Default c
0040288E  |. 0105 283F4200  ADD DWORD PTR DS:[423F28],EAX
00402894  |> 33C0           XOR EAX,EAX
00402896  |> 5F             POP EDI                                 ;  Case 2 of
00402897  |. 5E             POP ESI
00402898  |. 5B             POP EBX
00402899  |. C9             LEAVE
0040289A  \. C2 0400        RETN 4
0040289D   . 96284000       DD DAMsetup.00402896                    ;  Switch ta
004028A1   . 90144000       DD DAMsetup.00401490
004028A5   . 9C144000       DD DAMsetup.0040149C
004028A9   . B7144000       DD DAMsetup.004014B7
004028AD   . CA144000       DD DAMsetup.004014CA
004028B1   . D6144000       DD DAMsetup.004014D6
004028B5   . F0144000       DD DAMsetup.004014F0
004029D9  /$ 8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
004029DD  |. 8B0D 689B4000  MOV ECX,DWORD PTR DS:[409B68]
004029E3  |. FF3481         PUSH DWORD PTR DS:[ECX+EAX*4]           ; /Arg2
004029E6  |. 6A 00          PUSH 0                                  ; |Arg1 = 00
004029E8  |. E8 9B310000    CALL DAMsetup.00405B88                  ; \DAMsetup.
004029ED  |. 50             PUSH EAX
004029EE  |. E8 EA300000    CALL DAMsetup.00405ADD
004029F3  \. C2 0400        RETN 4
004029F6  /$ 56             PUSH ESI
```

Figure 10: Disassembled File

Consider the non-executable instruction at 004028A5 in Figure 10:

004028A5 9cC144000 DD DAMsetup.004019c

The linear sweep algorithm disassembles the above non-executable instructions as executable instructions:

004028A5 9C PUSHF

004028A6 1440 00 ADC AL , 00

A single disassembly error as shown above can significantly affect the disassembled output. Thus the linear sweeping algorithm does not always produce the correct source code. SoftICE and WinDbg are disassemblers that use the linear sweeping algorithm [13].

4.3.2 Recursive Traversal

Like the linear sweeping algorithm, recursive traversal algorithm starts processing the instructions from the beginning of the text section. The main difference is that the recursive traversal algorithm does not process the instructions sequentially. The algorithm depends on the control flow of the program. If the instruction processed is a jump instruction, then the algorithm jumps to the jump address specified and starts processing the bytes at the jump address instead of the consecutive bytes. This approach can detect data encoded in the instructions. But the recursive traversal algorithm cannot construct the correct code sequence in case of indirect branch instructions. So this algorithm also does not always produce the correct source code. IDA Pro and OllyDbg are disassemblers that use the recursive traversal algorithm [13].

4.4 Challenges in Disassembly

The following elements in the instruction stream may pose significant challenges in the disassembly process:

- Variable length instructions
- Embedded data in text sections(jump tables, alignment bytes)
- Branch instructions and indirect jumps
- Functions with no explicit call

Variable length instructions can cause a disassembler program to yield more than one reasonable disassembled output. Data embedded in text sections cannot be detected by disassembler algorithms like the linear sweep algorithm. Branch instructions, indirect jumps and functions with no explicit call may cause incorrect disassembled code when disassembled using the recursive traversal algorithm.

In addition to the above issues, memory requirements associated with the disassembly process can also be quite significant. According to PE Explorer, "Disassembling files larger than 1 Mb in size can take several minutes depending on the capabilities of your system. Generally, each byte of a target file requires 40 bytes of memory for processing. For example, a 1 Mb file would require 40 Mb of processing memory, a 2 Mb file — 80 Mb and so on" [14].

5. Extraction of opcode sequence

In this section, we describe our method of extracting opcode sequences from a executable. Stamp et. al in [9] used IDA Pro to disassemble executable files. From the IDA Pro disassembled files, the opcodes are extracted and concatenated to form an opcode sequence. The extracted opcode sequence is then used as an observation set for HMM to detect metamorphic viruses. But the disassembly process consumes a lot of time and becomes impractical for large files. We implemented an alternative method that efficiently parses the executable file and extracts an approximate opcode sequence from the file. The steps involved in extracting the approximate opcode sequence are as follows:

1. Extract the text section from executable file.
2. Extract the opcodes from the text section.
3. Concatenate the opcodes to form an approximate opcode sequence.

   Each of above steps is explained in the following subsections.

5.1 Extraction of text section

This section describes the structure of the Portable Executable (PE) format and the method used to extract the text from the PE file.

5.1.1 PE File Format

PE file format is the file format used in Microsoft Windows Operating Systems. We chose the PE executable format as our format, as it is the most commonly used file format and is the most vulnerable to virus attacks. PE file format is derived from the Common Object File Format (COFF), which is the file format of Unix System[15]. The

general layout of a PE file is shown in Figure 11

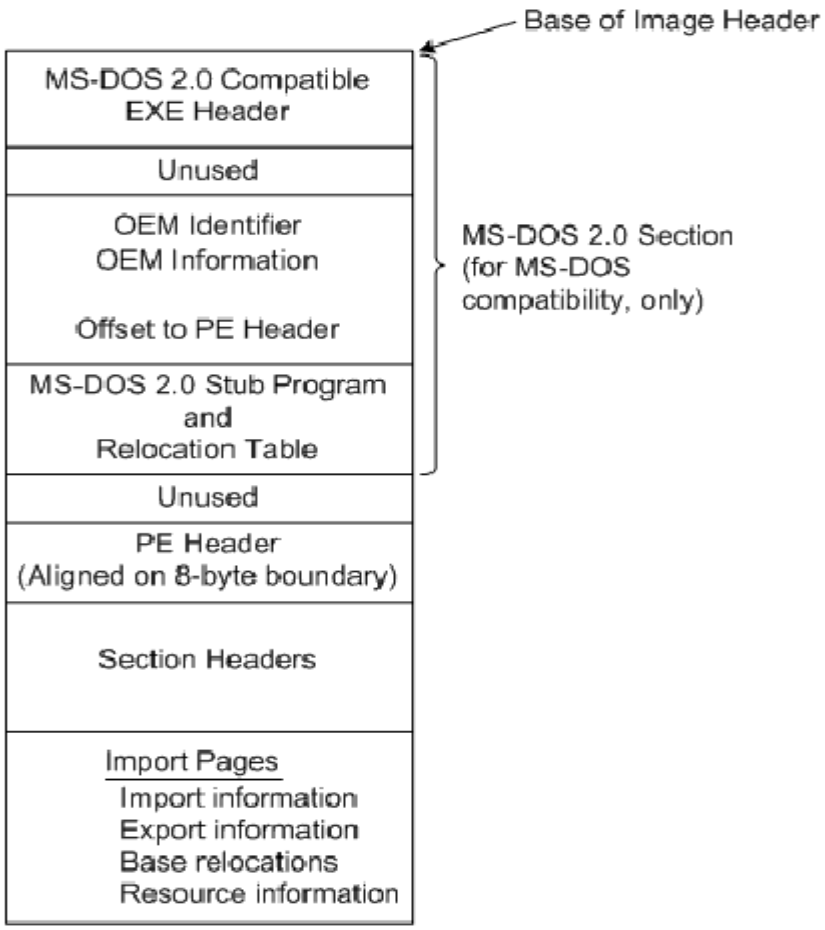

Figure 11: PE file format in [15]

The PE file consists of the following fields [15]:

- MS DOS header
- PE header
- Optional header
- Section header

- Sections

Each field of the PE file is explained in detail in the following subsections.

5.1.1.1 MS DOS header

An MS-DOS executable header is found at the start of the PE file. This header indicates that the PE file is a valid MS-DOS executable file. PE files with the MS-DOS header cannot be run on machines running different operating systems. The DOS header can be identified by a sixteen-bit signature. It is represented as “0x5A4D” in hex or “MZ” in ASCII [15].

5.1.1.2 PE header

The next field in the PE file format is the PE header. The PE header contains a thirty-two- bit signature, which is “0x00004550” in hex or “PE\0\0” in ASCII. This signature indicates that the executable is a PE file. The PE header consists of a file signature, COFF header, and an optional header. Table 1 contains the description of the COFF header fields [15].

TABLE 1: COFF HEADER [15]

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine.
Offset	**Size**	**Field**	**Description**
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.

12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
16	2	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.
18	2	Characteristics	The flags that indicate the attributes of the file.

The NumberofSections and the OptionalSizeheader fields are the only fields used in the extraction of the text section.

5.1.1.3 Optional header

As the name indicates, this header is optional for executable files. This header provides information to the operating system's loader. For example, this header is required for image files. Since the text section appears after the optional header, the optional header field is skipped in the extraction process [15].

5.1.1.4 Section header

The section header follows the optional header. It defines the structure of each section. The section header is 40 bytes long, and the following fields are needed for the text section extraction [15]:

- Name: This field determines the name of the section. It is used to identify the text section.
- Characteristics: This field describes the characteristics of the section. It is also used to identify the text section.
- SizeOfRawData: This field determines the size of the text section. It is used to extract the text section.
- PointerToRawData : This field points to the start of the text section. It is used to locate the code.

5.1.1.5 Sections

The sections are divided into the following five categories [15]:

- .text: This section contains executable instructions. These instructions cannot be altered, so it is a read-only section.
- .data: This section contains the global variables initialized by the programmer. These variables can be changed at runtime, so it is a read-write section.
- rdata: This section holds the debug data that is only present in executable files.
- .bss: This section contains uninitialized variables or common storage.
- idata: This section lists the symbols imported into a file.

Figure 12 shows the detailed layout of PE file including the different sections.

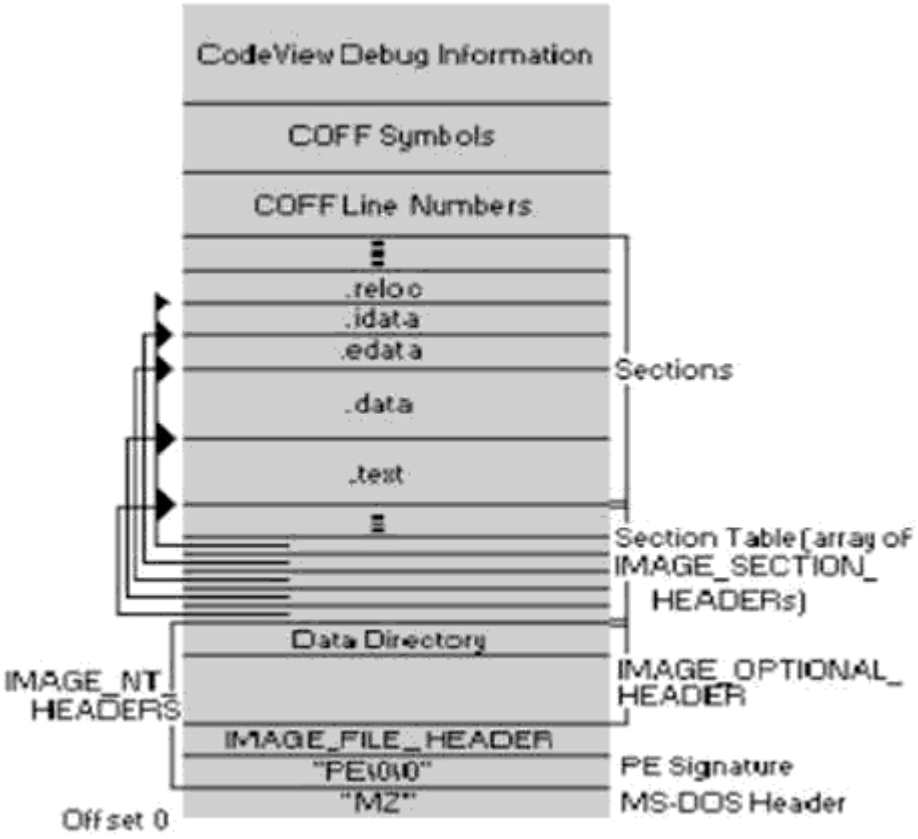

Figure 12: Detailed layout of PE Executable [15]

5.1.2 Extraction of text section

This section explains how the text section is extracted from a PE executable file. Govindaraj [16] explains the steps to extract the text section from an executable file as shown in Figure 13. The program first searches for the DOS header by looking for the signature “0x5A4D” or “MZ.” Then the program looks for the PE signature

“0x00004550” or “PE”. The program obtains the size of the optional header from the SizeOfOptionalHeader field and uses that to skip the optional header.

The program specifically looks for the text section. Each section can be identified by the name field or by the characteristics field. For the text section, the name field contains “.text” or “.code” name, and the characteristics field contains “0x0000020.” Since the name for the text section is not standardized, the program checks both the name and characteristic fields to identify the text section. The start of the text section is located using the PointerToRawData field. The text section is extracted using the SizeOfRawData field [16].

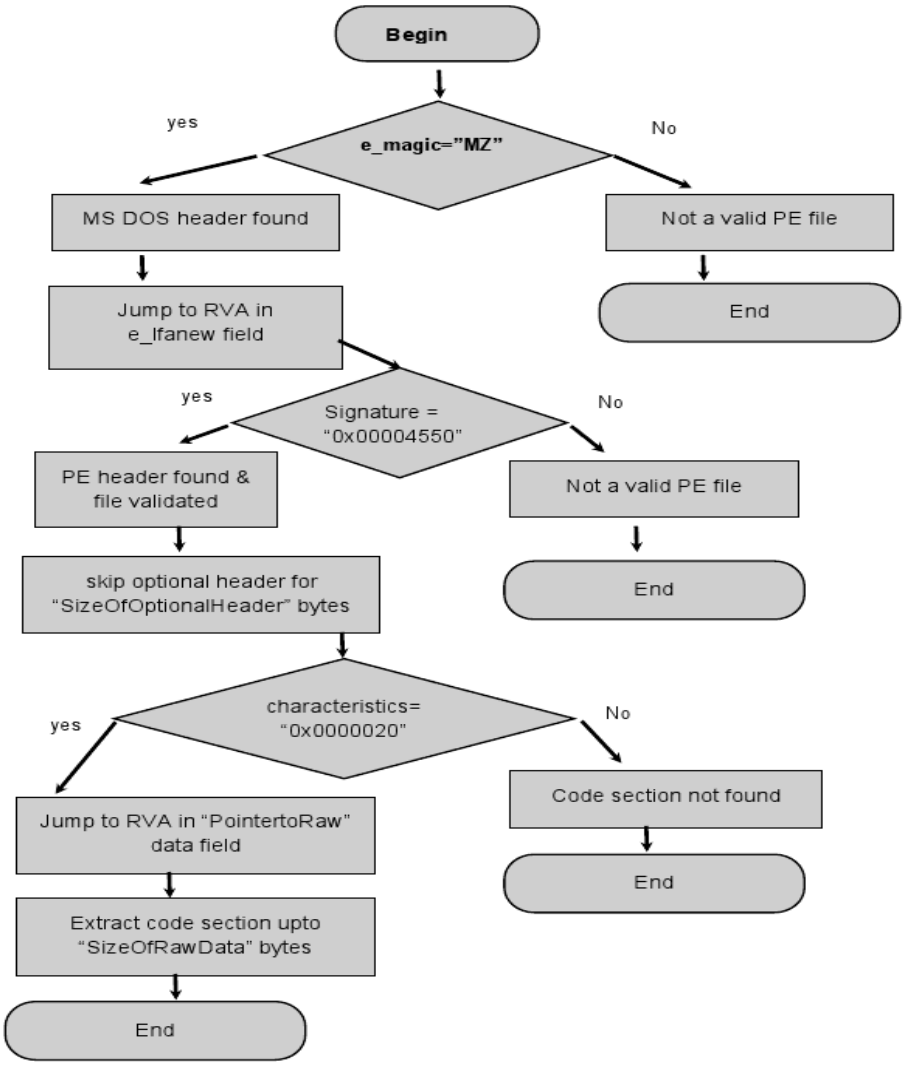

Figure 13: PE Text Section flow [16]

5.2 Approximate opcode sequence

The Intel instruction set contains more than 100 instructions. Disassembly process gets very complicated and time consuming if all the instructions are taken into account. As the executable file size increases, the complexity increases manifold when all the instructions are accounted for. In order to reduce the complexity, we only consider the Most Frequently Occurred (MFO) instructions in our method.

According to Billar, there are only 14 instructions in the Intel instruction set that are commonly used by PE files and these instructions are called MFO instructions. The MFO instruction set consists of ADD, AND, CALL, CMP, JMP, JNZ, JZ, LEA, MOV, PUSH, POP, RETN, TEST and XOR [17]. Figure 14 shows frequency of occurrence of MFO opcode mnemonics in normal files

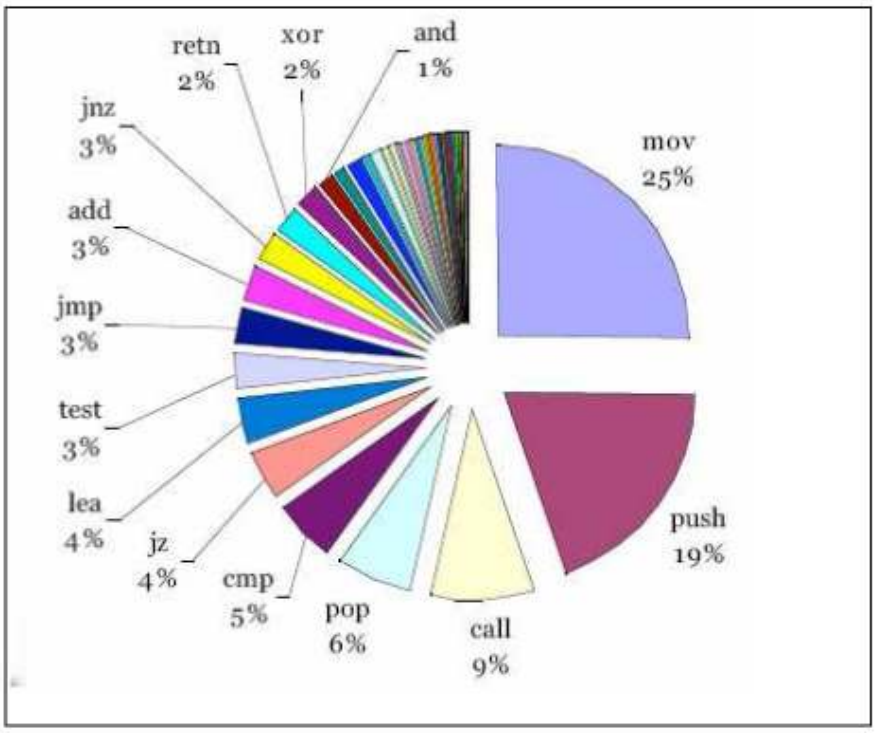

Figure 14: Frequency of occurrence of MFO opcodes in normal files (in percentage)

As discussed in Section 4.1, instruction opcodes are either 1 or 2 bytes long. The maximum length of a given instruction is 17 bytes. So the operand size for a given instruction can vary from 0 to 16 bytes. We divide the MFO instruction opcodes into three sets of opcode tables based on operands as follows:

- a table that has a list of opcodes with no operand
- a table that has a list of opcodes with one operand
- a table that has a list of opcodes with more than one operand

In addition to the three sets of MFO opcode tables, we also collect statistical data for the occurrence of opcode pairs as shown in Table 2.

TABLE 2: SAMPLE STATISTICAL DATA OF OPCODE PAIRS

Opcode Pair (op_i , op_{i+1})	Length of bytes, n	Probability value
ff, ff	2	0.025
8b, 8b	4	0.014
50, ff	0	0.010
8d, 50	2	0.010
e8, 8b	4	0.009
ff, 8b	1	0.009
8b, 83	2	0.009
ff, e8	5	0.008
85, 74	2	0.008
8d, e8	5	0.008

A large set of normal files is disassembled using IDA Pro. Then the frequency of occurrence of opcode pairs is collected from the disassembled files in addition to the number of bytes between them. The probability of op_i followed by op_{i+1} when there are n bytes between them is calculated and stored as the statistical triplet (opi, opi+1, n). Using this statistical data, an approximate opcode sequence is extracted with our algorithm as shown. Figure 15 shows frequency of occurrence of MFO pairs in normal files

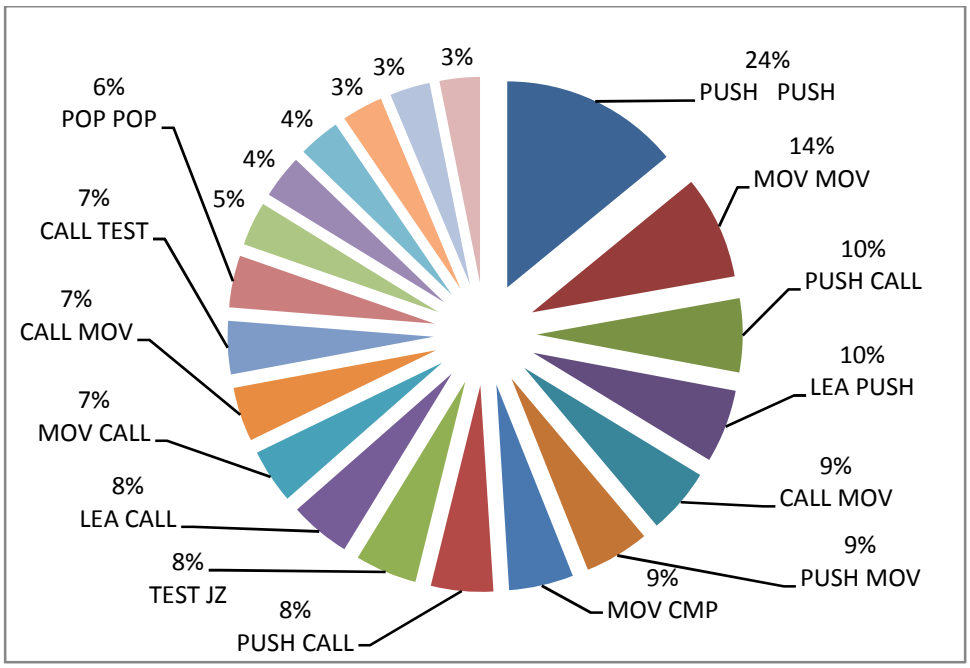

Figure 15: Frequency of occurrence of MFO pairs in normal files (in percentage)

Figure 16 shows the program flow of our method. In our method, the program takes a PE file as an input. Since the text section has he executable instructions, the program extracts that section from the PE file. We start processing the input one byte at a time. If the byte read is “0F”, then this can be a 2 byte opcode, so the next byte is read and appended to this byte. Else, the read byte is assumed to be a 1 byte opcode. The opcode that is read is then checked against each of the three tables. If the opcode occurs in the table with no operands, then the next byte is assumed to be the next opcode. If the opcode occurs in the table with one operand, the next byte is skipped and the following byte is assumed to be the next opcode.

If the opcode occurs in the table with more than one operand, then a solution tree is constructed with all possible operand sizes. The operand size varies from 0 to 16 bytes in the Intel instruction set. There are very few opcodes which use 16 byte operands, so we optimize the problem by restricting it to a maximum of 8 byte operands. So the next opcode can be anywhere from 1 to 9 bytes away from the current byte. For each of the possible opcode pairs, the probability of occurrence of these pairs and the distance between them is annotated by adding the logarithmic value of the probability from the statistical table into the solution tree. We use the logarithmic value of the probabilities to avoid a numerical underflow in calculations associated with multiplication of small numbers. If the opcode pair does not occur in the statistics then it is assigned a low probability value (-100.0). If the opcode does not occur in any of the tables, the read byte is skipped. This process is repeated for every byte until the end of the file and a full sequence list with multiple branches are constructed. As we continue processing bytes in

the input stream, we optimize the solution tree by removing branches with low probability values. At the end of the file, an approximate opcode sequence is obtained by choosing the branch with the highest probability.

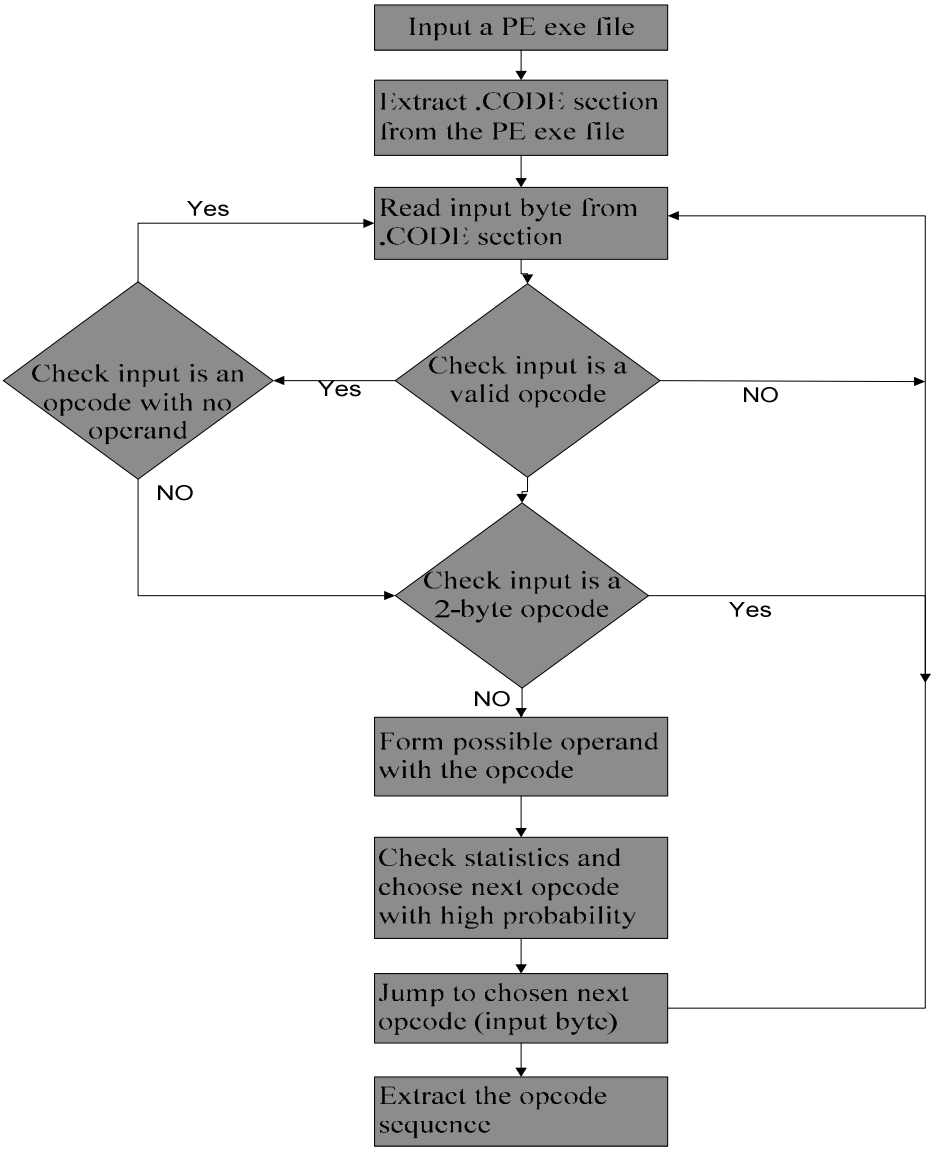

Figure 16: Method for extracting approximate opcode sequence

For example, consider the following sequence of input stream

Input stream: 568b7424088a46103c017439 3c02751c8be83f00….

Let's divide the input stream byte by byte as shown below

Input byte stream: 56 8b 74 24 08 8a 46 10 3c 01 20 39 3c 02 75 1c e8 3f 00….

Let's take the first input byte 56. Since the first input byte is an opcode with no operand, we know the next input byte 8b is an opcode that follows 56. The opcode sequence becomes 568b. The second input byte is 8b. The opcode 8b is in the list of opcodes with more than one operand. So we form a solution tree of next possible opcodes based on 8 possible operands. Table 3 shows the next possible opcode depending on the operands of the opcode 8b. Note the opcode pair (8b, 20) is assigned the value of -100.0 as it does not appear in the statistical data. This process continues till the end of the input file.

TABLE 3: SOLUTION TREE CALCULATION

Opcode	Possible Operand	# of Operands	Next opcode	Opcode sequence	Statistics Value
8b	74	1	24	8b 24	-10.761
8b	74 24	2	08	8b 08	-100.0
8b	74 24 08	3	8a	8b 8a	-9.505
8b	74 24 08 8a	4	46	8b 46	-10.499
8b	74 24 08 8a 46	5	10	8b 10	-100.0
8b	74 24 08 8a 46 10	6	3c	8b 3c	-10.291

8b	74 24 08 8a 46 10 3c	7	01	8b 01	-12.136
8b	74 24 08 8a 46 10 3c 01	8	20	8b 20	-100.0

Figure 17 shows a section of the solution tree for the above example.

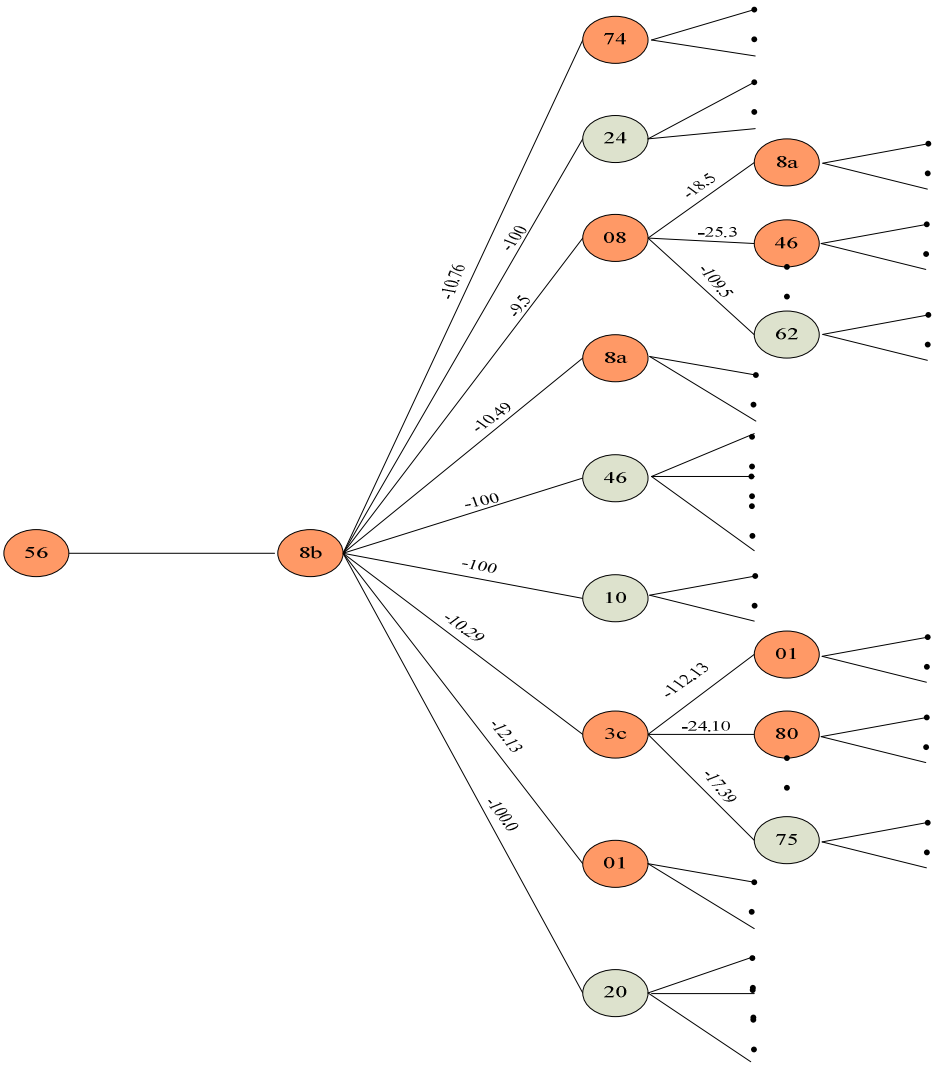

Figure 17: Example of solution tree in extracting opcode sequence

6. Implementation Results

An input set of 200 normal files are taken and 150 out of the 200 files are used to collect the statistical data of opcode pair occurrences. This statistical data is collected using the IDA Pro disassembler. Using the statistical data, an approximate opcode sequence is extracted from the remaining 50 files .Then the extracted opcode sequence of each file is compared with IDA Pro. An average of 70.2% accuracy is achieved in extracting the opcode sequence when compared with IDA Pro.

The test sets below measures the time vs. file size comparison between our algorithm and the IDA Pro disassembler. The time taken by both the algorithms for each file is measured. The test set measures a set of files that range from 1MB to 5MB.

Test set 1: Average File Size = 1MB

In this test set, the file size ranges from 0.5MB to 1.5MB. An average time of 4.4 seconds is taken by our algorithm whereas 5.3 seconds time is taken by the IDA Pro disassembler. For this set, our algorithm is about 20% faster than IDA Pro. For some files in the data set, IDA Pro is faster than our algorithm, but the majority of the files take a shorter time with our algorithm than IDA Pro.

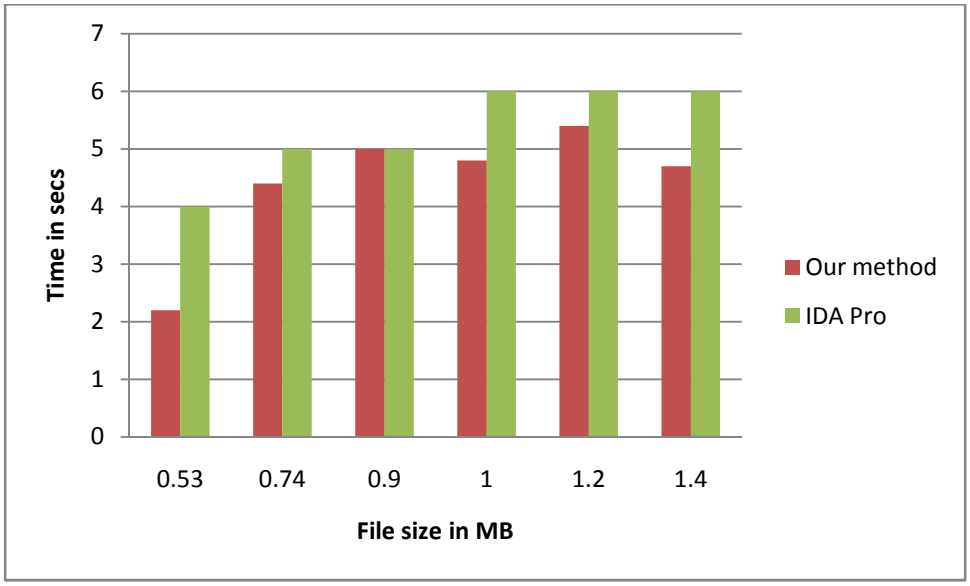

Figure 18: Comparison of time taken between our approach and IDA Pro, File set 1MB

Test set 2: Average File Size = 2MB

In this test set, the file size ranges from 1.5MB to 2.5MB. An average time of 5.7 seconds is taken by our algorithm whereas 7.3 seconds time is taken by the IDA Pro disassembler. For this set, our algorithm is about 28% faster than IDA Pro.

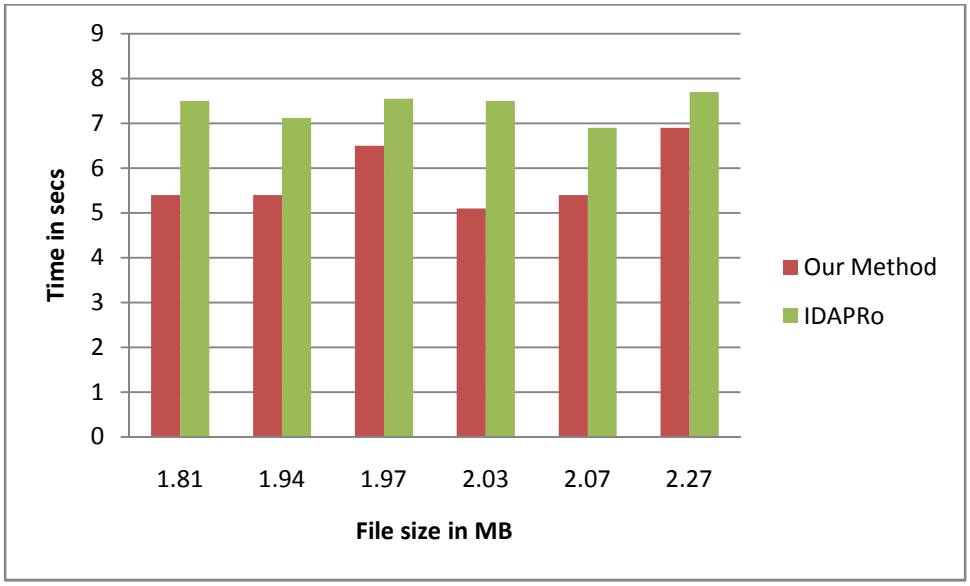

Figure 19: Comparison of time taken between our approach and IDA Pro, File set 2MB

Test set 3: Average File Size = 3MB

In this test set, the file size ranges from 2.5MB to 3.5MB. An average time of 10.87 seconds is taken by our algorithm whereas 13.4 seconds time is taken by the IDA Pro disassembler. For this set, my algorithm is about 34% faster than IDA Pro.

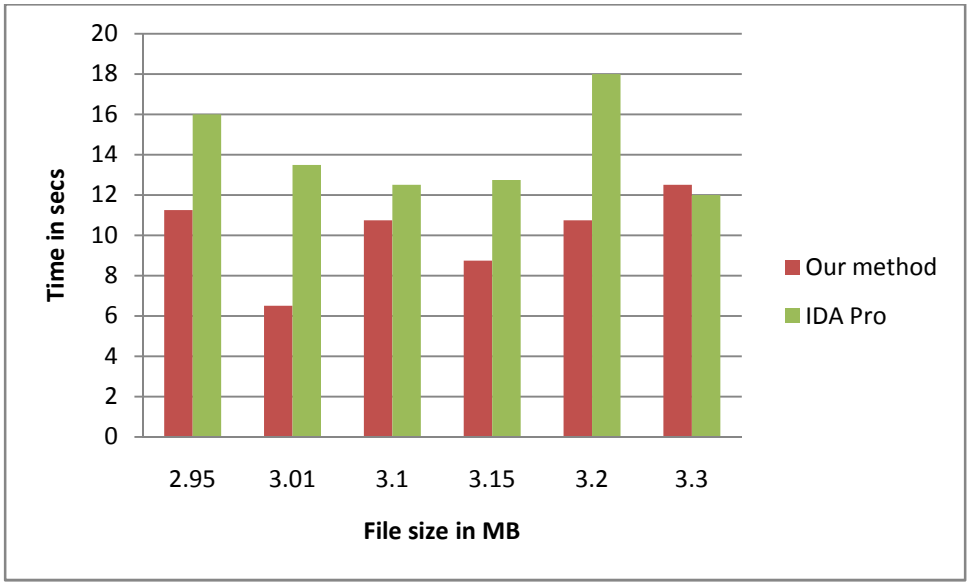

Figure 20: Comparison of time taken between our approach and IDA Pro, File set 3MB

Test set 4: Average File Size = 4MB

In this test set, the file size ranges from 3.5MB to 4.5MB. An average time of 12.9 seconds is taken by our algorithm whereas 16.7 seconds time is taken by the IDA Pro disassembler. For this set, our algorithm is about 38% faster than IDA Pro.

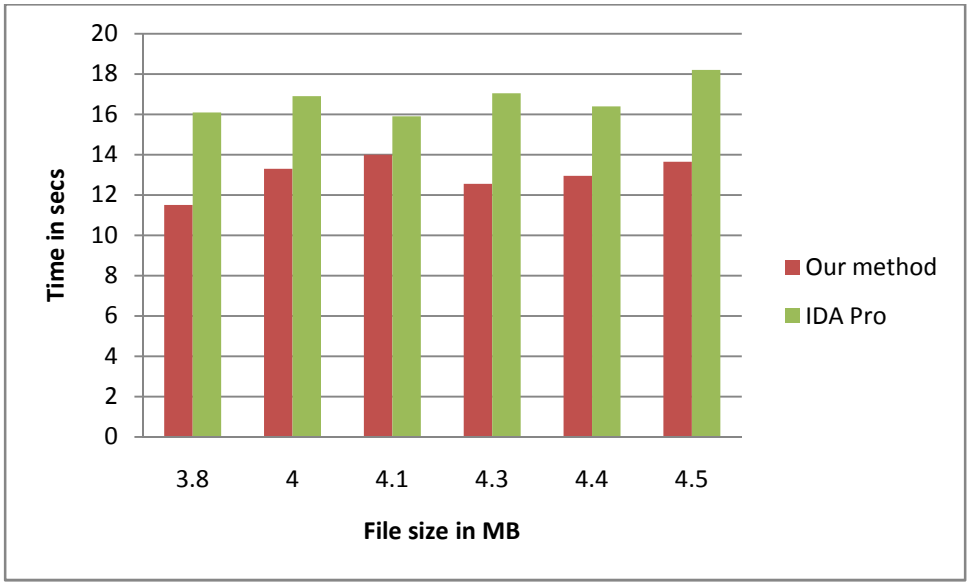

Figure 21: Comparison of time taken between our approach and IDA Pro, File set 4MB

Test set 5: Average File Size = 5MB

In this test set, the file size ranges from 4.5MB to 5.5MB. An average time of 19.25 seconds is taken by our algorithm whereas 23.6 seconds time is taken by the IDA Pro disassembler. For this set, our algorithm is about 44% faster than IDA Pro.

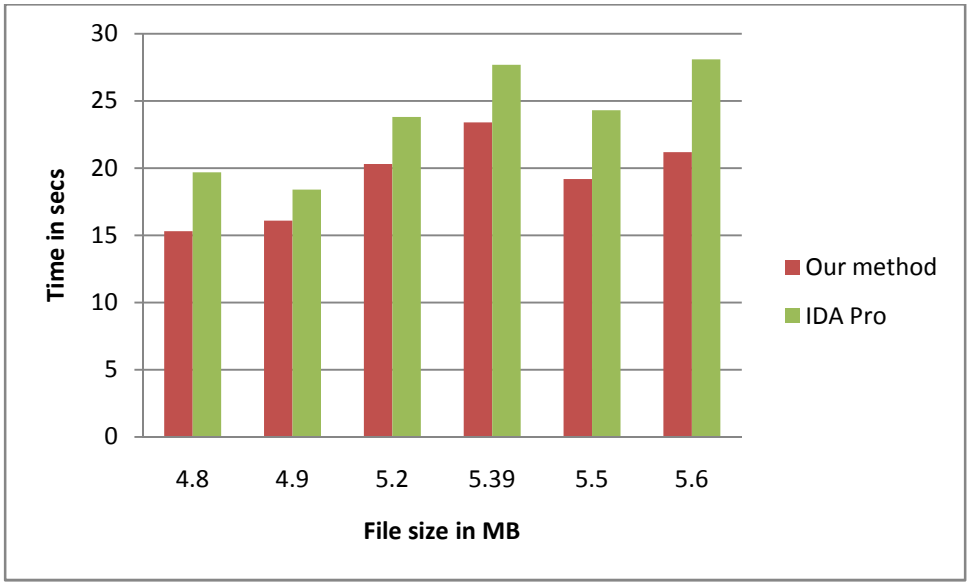

Figure 22: Comparison of time taken between our approach and IDA Pro, File set 5MB

Test set 6

Figure 23 shows the average time taken for a given file size by our algorithm and IDA Pro.

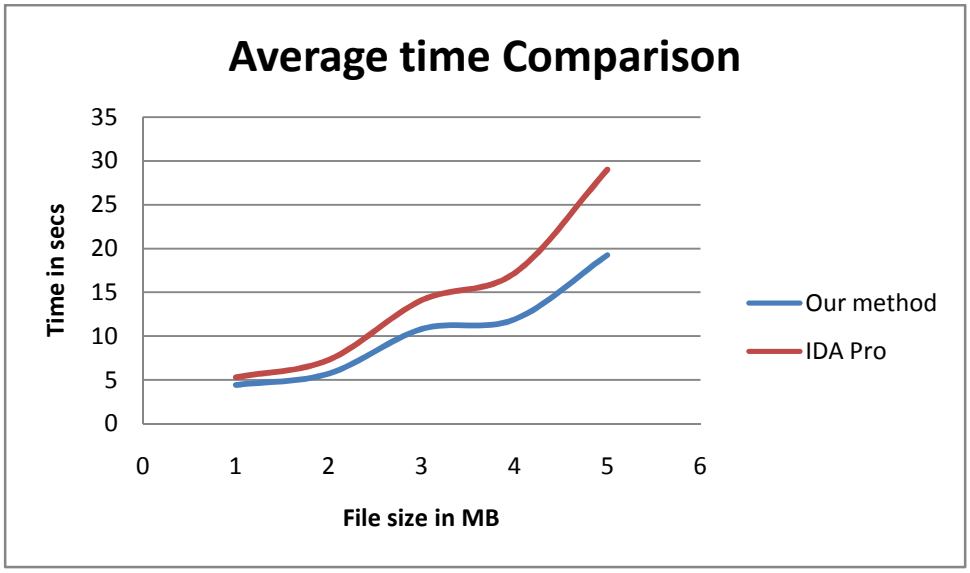

Figure 23: Average time comparison between our approach and IDA Pro

The graph shows that for small files, our algorithm is slightly better than IDA Pro. As the file size increases, IDA Pro's disassembly time increases linearly. Our algorithm's disassembly time also increase linearly. As the file size increases, the time difference between our algorithm and IDA Pro increases.

Accuracy of opcode sequence

The test set 7 below measures the accuracy of our algorithm as compared to IDA Pro for different file sizes. For a given file, we extract the output from IDA Pro and our algorithm. For every opcode pair in our output, we calculate the frequency of occurrence of the pair. We do the same for all the pairs in IDA Pro's output and then calculate the difference in frequencies between our output and IDA Pro's output. This difference is the number of misses that our algorithm has had with respect to IDA Pro. The total number of misses in the output divided by the total frequency in IDA Pro output gives us the accuracy of our algorithm for that file. We repeat this exercise for all the files in our test sets. Table 4 shows the error calculation for an example file of size 47KB.

TABLE 4: ERROR CALCULATION OF AN EXAMPLE FILE

Opcode Pair, Op_i	IDA Pro Output P_i for file(f)	Our method R_i output for file(f)	Miss Calculation, ($M_i = P_i - R_i$)
ff, ff	213	259	46
50, ff	72	78	6
8b, 8b	71	60	11
56, e8	63	57	6
68, ff	54	48	6
ff, 85	54	54	0
ff, 8b	53	69	16
57, e8	51	51	0
57, ff	50	54	4
6a, e8	49	73	24
85, 74	48	46	2
ff, e8	46	53	7
8b, 83	45	38	7
6a, ff	44	42	2
8d, 50	42	43	1
85, 75	38	37	1
ff, 6a	37	34	3
53, ff	34	37	3
ff, 83	34	31	3
50, e8	33	38	5
e8, 85	33	32	1
e8, 8b	33	39	6
39, 74	32	33	1
3b, 74	32	35	3
5f, 5e	32	30	2
83, 75	31	28	3
Total	$T(P_i) = 1324$	$T(R_i) = 1399$	$T(M_i) = 169$

From Table 4, the error percentage for the file (f) is calculated as:

$$\text{Miss percentage, } MP = \frac{T(M_i)}{T(P_i)} \times 100$$

where i= 0, 1, 2…N and N is the length of opcode pair in a given file (f).

Accuracy percentage, AP = (100 - MP) %

For the sample file (f) in table 4, the miss percentage is calculated as 12% ((169/1324)*100). Thus the accuracy of opcode sequence for the file (f) with respect to IDA Pro is 88%.

Figure 24 shows the average accuracy of approximate opcode sequence with respect to IDA Pro for different file sizes in our test set. An average accuracy of 70.2% is achieved for all the files in our test set.

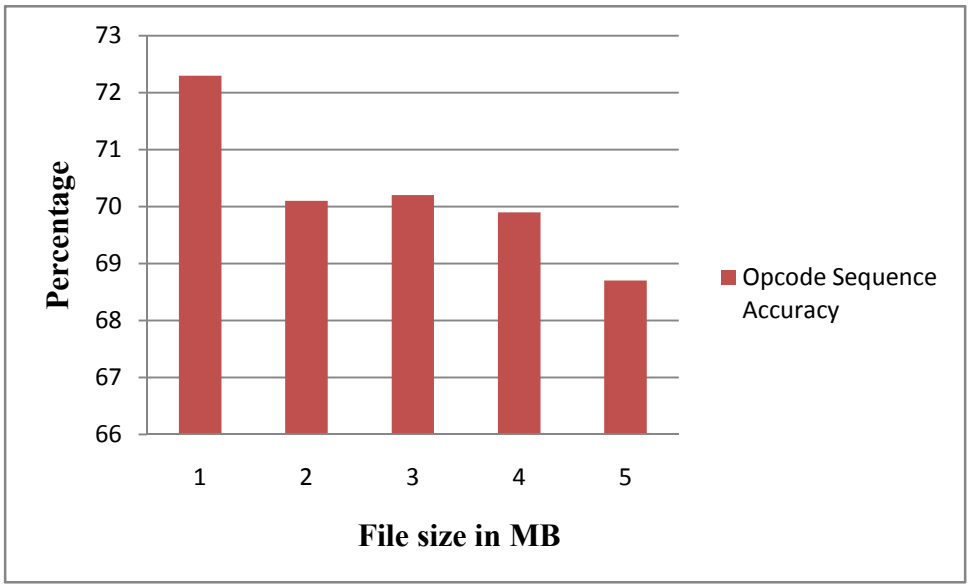

Figure 24: Accuracy of Opcode Sequence

As the file size increases, the accuracy decreases, but the drop in accuracy is not significant as the file size increases. This accuracy is good enough for detecting viruses using HMM.

These results prove that our approximate opcode sequence extraction algorithm is faster than IDA Pro for most cases and offers a significant speed up for larger files with a marginal loss of accuracy.

7. Conclusion and Future works

We implemented an approximate disassembly method that identified Most Frequently Occurred (MFO) instructions in the executable files and constructed three groups of opcode tables based on their operand sizes. We also collected the frequency of occurrence of opcode pairs from a set of files. An approximate opcode sequence is extracted from a given executable file, using the opcode tables and opcode pair statistics. Our method reduced the disassembly time by 45% as compared to the industry standard disassembler IDA Pro. It also achieved an accuracy of 70% as compared to IDA Pro.

We collected opcode pair statistics to improve the extraction time. This concept can be extended to opcode triplets and, possibly longer opcode sequences, which might improve our accuracy. We currently use separate opcode tables for opcodes with zero and one operands. This can be extended to opcodes with two or three operands. This might reduce the extraction time and increase the accuracy of the extracted sequence.

8. References

[1] Computer virus protection, Norton anti-virus software 2010

< http://www.norton-security-store.com/knowledge-center/computer-virus-damage.html>

[2] LOVELETTER virus <http://en.wikipedia.org/wiki/ILOVEYOU>

[3] Hidden Markov Model (HMM)
<http://en.wikipedia.org/wiki/Hidden_Markov_model>

[4] Types of viruses

<http://www.buzzle.com/articles/different-types-of-computer-viruses.html>

[5] Real-time example of viruses

<http://antivirus.about.com/cs/tutorials/a/bsvirus_2.htm>

[6] W. Wong, "Analysis and detection of metamorphic computer viruses"

< http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf>

[7] P. Szor & P. Ferrie," Hunting For Metamorphic", Symantec Security Response.

< http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf>

[8] M. Stamp, "A revealing introduction to hidden Markov models"

<www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf>

[9] M. Stamp & W. Wong, "Hunting for metamorphic engines"

< http://www.cs.sjsu.edu/faculty/stamp/papers/Wing.pdf>

[10] K. Gershon CPS 104, Fall 2009, Lectures.

<http://kedem.cs.duke.edu/cps104/Lectures.html>

[11] Intel Architecture Software Developer's Manual (1997).

<http://download.intel.com/design/PentiumII/manuals/24319102.PDF>

[12] Intel 64 and IA-32 Architectures Software Developer's Manual (2010).

<http://www.intel.com/Assets/PDF/manual/253665.pdf>

[13] A. Singh, "Identifying Malicious Code Through Reverse Engineering"

<http://books.google.com/books?id=YKl6XnEa_CAC&pg=PA130&lpg=PA130&dq=linear+sweeping+disassembler&source=bl&ots=0KsLjMAvmU&sig=SxCOF4RpvBNJ61b9V3FOXgezhkU&hl=en&ei=sbylS975AY6QsgPanvEh&sa=X&oi=book_result&ct=result&resnum=1&ved=0CAgQ6AEwAA#v=onepage&q&f=true>

[14] PE Explorer Disassembler

<http://www.pe-explorer.com/peexplorer-tour-disassembler.htm >

[15] Microsoft Portable Executable and Common Object File Format Specification.

<http://www.scribd.com/doc/8345966/Microsoft-Portable-Execution- and-Common-Object-FIle-Format-Specification>

[16] S.Govindaraj "Practical Detection of Metamorphic Computer Viruses, masters project" Department of Computer Science, San Jose State University

<http://www.cs.sjsu.edu/faculty/stamp/students/Govindaraj_Sharmidha.pdf>

[17] Billar, D. "Statistical Structures: Fingerprinting Malware for Classification and Analysis"

<http://cs.wellesley.edu/~dbilar/papers/Bilar_OpcodeDistribution_ICGeS07.pdf>