APPROXIMATE DISASSEMBLY USING DYNAMIC PROGRAMMING

A Research Project

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 Computer Science

by

Abhishek Shah

Fall 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Project Titled

APPROXIMATE DISASSEMBLY USING DYNAMIC PROGRAMMING

by

Abhishek Shah

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp, Department of Computer Science Date

Dr. Sami Khuri, Department of Computer Science Date

Dr. Robert Chun, Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

Abstract

APPROXIMATE DISASSEMBLY USING DYNAMIC PROGRAMMING

by Abhishek Shah

Most commercial anti-virus software uses signature based techniques to detect whether a file is infected by a virus or not. However, signature based detection systems are unable to detect metamorphic viruses, since such viruses change their internal structure from generation to generation. Previous work has shown that hidden Markov models (HMMs) can be used to detect metamorphic viruses. In this technique, the code is disassembled and the resulting opcode sequences are used for training and detection. Due to the disassembly step, this process is not efficient enough to use when a decision has to be made in real time.

In this project, we explore whether dynamic programming can be used to speed up the process of disassembling, with minimal loss of accuracy. Dynamic programming is generally used to solve problems having two key attributes: optimal substructure and overlapping sub problems. During each iteration our algorithm reads part of the input stream from the executable file and determines assembly instructions, thus dividing problems into sub problems.

We have created a score matrix representing digraphs of the most common opcode instructions and we have implanted a dynamic program based on this scoring matrix. For various file sizes, we determine the time taken by our dynamic program and we show that our approach is significantly faster than a standard disassembler (OllyDbg). Finally, we analyze the accuracy of our results.

Acknowledgements

I would like to thank Dr. Mark Stamp for guiding and encouraging me throughout the project. I would also like to thank my committee members, Dr. Sami Khuri and Dr. Robert Chun for helping me during the project.

Table of Contents

List of Figures

List of Tables

APPROXIMATE DISASSEMBLY
USING DYNAMIC PROGRAMMING

1. Introduction

Viruses are among the most challenging problems in computer security. According to Cohen [12], a computer virus is a program that disrupts the normal functioning of a system by modifying the underlying programs or by using resources without the consent of the user. A virus can cause harm to a host machine or a system. The effect of malware can be as simple as displaying a threatening message, or as complex as subtlety changing the functionality of an important program. For example, a recent virus named Stuxnet was identified as code that could reprogram programmable logic control software to give an attached nuclear controller new instructions [13].

To detect viruses, most anti-virus software uses signature based techniques. A signature generally consists of binary data that represent the file [18]. To avoid signature based detection, virus writers have developed sophisticated methods, including polymorphic viruses, oligomorphic viruses, and metamorphic viruses [2]. Metamorphic viruses, which are, arguably, the most dangerous of all, change their structure or signature each time they propagate, without changing the functionality of the virus.

Research currently being conducted in the field of metamorphic virus detection includes: [1]. Hidden Markov models (HMMs) have proved to be an effective technique for detecting metamorphic viruses [4] and [19]. One disadvantage of using HMMs is that an executable file has to be disassembled and its opcode sequence extracted before it can be scored, and this process of disassembling can be time consuming [16]. In this paper, we present a fast approach to disassembly, using dynamic programming. In dynamic programming a complex problem is divided into smaller problems in recursive manner [8]. The results of the solved smaller problems are stored for later reference.

The aim of this project is to use dynamic programming to reduce the time required to disassemble executable files. First, the .text section, which contains program code, is extracted from the executable file. We then determine an opcode sequence by scoring possible paths based on pre-computed statistics obtained by disassembling a large number of executable files.

This paper is organized as follows. In Section 2, we provide background information about viruses and their types and we discuss the techniques used to generate metamorphic viruses. Section 3 describes the Hidden Markov Model. Section 4 describes how HMMs can be used to detect metamorphic viruses. Section 5 describes different methods for disassembling an executable file and discusses Intel Architecture instruction format. Section 6 discusses dynamic programming in general. Section 7 explains our algorithm and how it can be used to accurately determine assembly code from an executable file. Section 8 provides test results for the speed and accuracy of our technique, for a wide variety of file sizes. Section 9 presents our conclusions and suggestions for future work.

2. Computer Viruses

A computer virus is a malicious program which infects a host system without the consent of the user. It is responsible for altering the default system behavior. Computer viruses find executable files and infect them by copying code known as payload into them. Finally the virus will determine if the desired condition, like number of infections, is met [1]. Anti-virus software detects the presence of viruses in the system and removes them. Most anti-virus programs use signature based detection. Various methods are used to avoid signature based detection, which we will discuss in the next section.

According to [2] and [3], there are five types of viruses: simple, encrypted, oligomorphic, polymorphic, and metamorphic.

2.1. Simple Viruses

A simple virus replicates itself while infecting files and does not use sophisticated methods to hide itself from detection. When a program infected by this type of virus is opened; the virus alters the default behavior of the computer and replicates itself to other files. Each virus of this type has a specific signature. This makes it very easy for anti-virus software to detect and remove them.

2.2. Encrypted Viruses

Encrypted viruses were invented to hide the malicious functionality. The body of this virus consists of constant decryptor and encrypted virus body. The malicious intent is hidden in the encrypted body of the virus. During infection, the decryptor first decrypts the encrypted body and thereafter spreads the infection. These viruses can be easily detected because they use common decryptors. The anti-virus software can check the signature of the decryptor to detect these types of viruses.

2.3. Oligomorphic Viruses

Oligomorphic virus is an improved version of the encrypted virus. In this type of virus the decryptor is changed each time during propagation. However, there is a limited number of forms in which the decryptor can exist. According to [2], Win95/Memorial had the ability to build 96 different decryptor patterns. Thus, signature based detection technique can still be used if all of the different flavors of decryptor are included. Another technique used to detect these viruses is dynamic decryption of encrypted virus, after which signature based detection techniques can be used.

2.4. Polymorphic Viruses

Polymorphic virus is capable of creating an infinite number of decryptors after each infection. It consists of a decryptor, a mutation engine, and a virus body. The mutation engine changes the decryptor, which thwarts detection by signature based antivirus software. However, polymorphic viruses can be detected by first using dynamic decryption and then using signature based techniques on the unencrypted body. Figure 1 shows different body structures of same polymorphic virus.

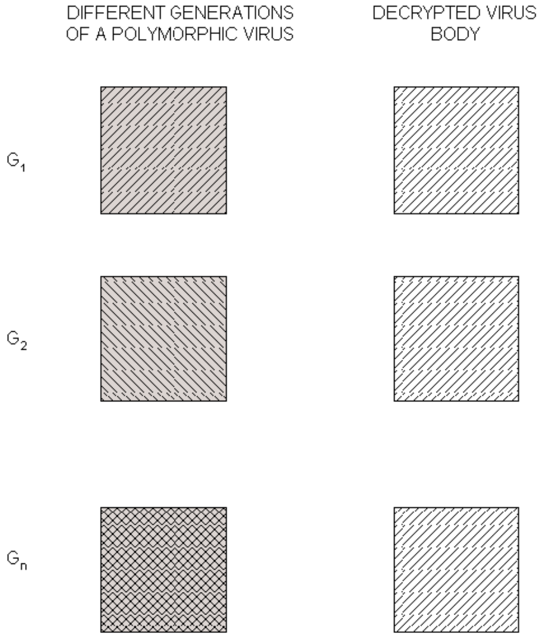


Figure 1: Generations of Polymorphic Virus [2]

2.5. Metamorphic Viruses

Unlike other kinds of viruses, metamorphic viruses do not have a decryptor, a mutation engine, or an encrypted virus body. Metamorphic viruses change their form each time they spread infection while keeping their functionality intact. In this way they avoid detection using signature based techniques commonly employed by anti-virus software. Code obfuscation techniques are used by metamorphic viruses to change body structure as shown in Figure 2.

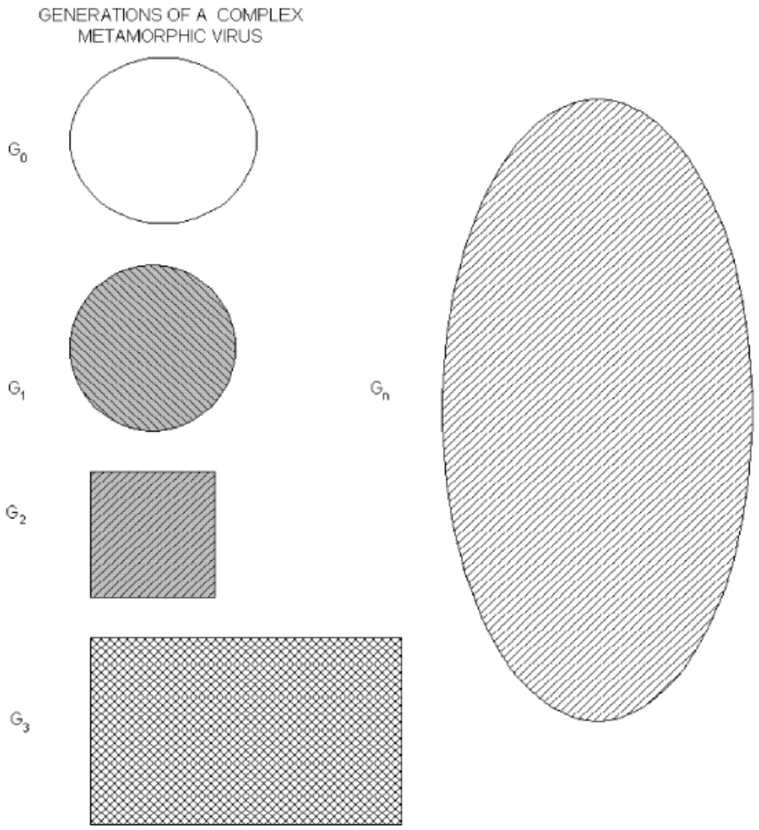


Figure 2: Virus Body of Different Metamorphic Virus [2]

2.5.1. Different Techniques of Metamorphism

According to [1], various types of techniques are used by metamorphic viruses in order to avoid detection. Metamorphic virus might use one or more of the following techniques.

2.5.1.1. Garbage Code Insertion

Garbage code insertion technique is a simple technique used to generate metamorphic virus. In this technique some code is inserted that does not change the default functionality of the virus. A simple example is to insert a for loop which does not do anything. The code inserted is called garbage since it does not do anything useful. The use of this technique avoids signature based detection used by the anti-virus software.

2.5.1.2. Register Usage Exchange

The register usage exchange technique uses different registers in different generations of virus. The code and functionality remain unchanged in this technique. Here the complexity of code is not very high. Anti-virus software which does not support wild card string matching cannot detect virus generated by this technique.

2.5.1.3. Permutation Technique

The permutation technique divides the code into many fragments and then rearranges it in different permutations from generation to generation. Jump instructions are used to connect these fragments. However, the control flow during each generation remains the same. If the code is divided into n fragments, then there is a possibility of generating n! metamorphic virus.

2.5.1.4. Insertion of Jump Instruction

Metamorphic viruses sometimes use jump instructions to generate different body structures. The jump instruction is removed or inserted at random locations, and it points to the next instruction within the virus code. This type of virus does not generate a constant body, even in memory, and they are not possible to detect using wild card string matching.

2.5.1.5. Instruction Replacement

Another method used by metamorphic virus is the replacement of the instructions which match the functionality. If there are two instructions which have the same functionality but different opcode, then this technique can be used by metamorphic viruses to avoid detection. For example “AND ESI, ESI” can be replaced by “TEST ESI, ESI” or vice versa, since both have the same functionality. Another example is to use different versions of conditional jump instructions and modify the code accordingly.

2.5.1.6. Host Code Mutation

The host code mutation technique targets different executable files on the host computer during each generation. This produces new viruses during each generation. Since it infects different executable files, it is impossible to have a common disinfection technique [20].

2.5.1.7. Code Integration

Code integration is a sophisticated technique used by metamorphic virus to generate new body structure during each generation. In this technique, the virus first decompiles the executable file, divides the code into different fragments, inserts virus code, and compiles the entire code again to generate new executable code. This makes it hard to detect the virus, and even more difficult to repair the executable [2].

3. Hidden Markov Model (HMM)

A Hidden Markov Model is a statistical Markov model in which the hidden states are used to produce the observation state. A Markov model determines the current state on the basis of the previous state. In a Hidden Markov Model the states are invisible to the user. However, observation states are visible. Each observation state depends on the hidden states. According to [7], we can use the following notation to represent HMM

Let

T = length of the observation sequence

N =number of states in the model

M =number of observation symbols

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

V = $\{0, 1, . . . , M - 1\}$ = all possible observations

A = state transition probabilities

B = observation probability matrix

π = initial state distribution

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

Figure 3 shows a Hidden Markov Model where each Markov process X_i [except X_0] is generated by taking into consideration the previous Markov process and A matrix, which represents state transition probabilities. The user can only see the observation sequence O. Each observation state is produced by Hidden Markov Process X_i and B matrix, which represents the observation probability matrix.

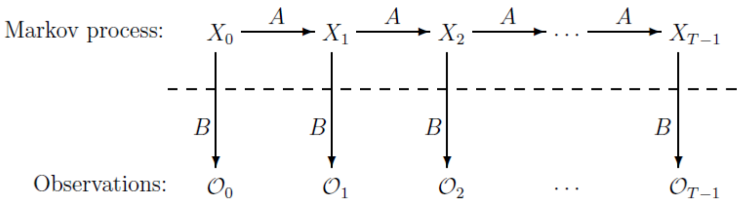


Figure 3: Hidden Markov Model [7]

HMM is used in applications such as speech recognition, cryptanalysis, gene prediction, etc. where the output depends on states which are not observable.

4. HMM for Metamorphic Virus Detection

According to [4], the HMM model can be used to detect metamorphic viruses that belong to the same family. In this method, HMM is first trained by giving assembly code of various metamorphic virus files as input. All the executable files of the same metamorphic virus family are disassembled, and opcode are extracted. A disassembler, such as OllyDbg, is used for disassembling the exe file. These opcode are concatenated to form a long sequence. Each of these sequences is given as input to the HMM model and thus, at the end of the process, the HMM model represents a statistical model of the virus family. Figure 4 shows the entire process of detecting metamorphic viruses with the help of HMM model.

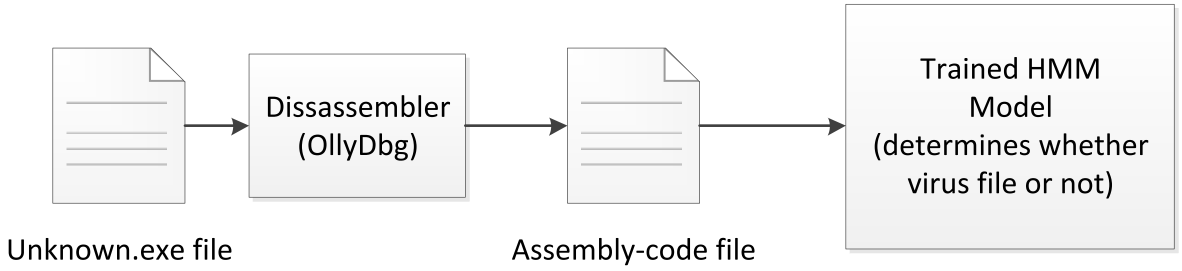


Figure 4: Process for Detecting Metamorphic Virus

The HMM is tested against two types of files: one type belongs to the same metamorphic virus family and other type is a non-virus program or a virus file of some other family. The HMM should give a high score for files that belong to the same metamorphic virus family for which we trained our HMM. However, the HMM should give a low score for any non-virus program or virus file of some other family [4].

However, there is one problem with the above method. One has to disassemble the entire executable file in order to determine whether it is a virus file or not. The process of disassembly takes a long time. For example, it would take 18 sec on average to disassemble a 3.6MB executable file. This can be optimized by using dynamic programming, which is faster than disassemblers such as OllyDbg. Later, we present our algorithm, which produces assembly code at a faster rate than OllyDbg and with a great deal of accuracy. The following section explains the compilation and reverse engineering process.

5. Technical Details of Disassembly

In this section, we discuss the process of converting source code into executable and the process of converting executable file into assembly code. We also discuss the types of disassembly and Intel instruction format.

5.1. Compilation

A computer programmer writes a program in high level language like C, C++, etc. The source code is converted into assembly code and finally to machine code (executable file) which is platform-dependent. This process is known as compilation. Disassembly is the process of converting machine code into assembly code. The process of converting assembly code back to source code is known as decompilation. Figure 5 shows the entire process.

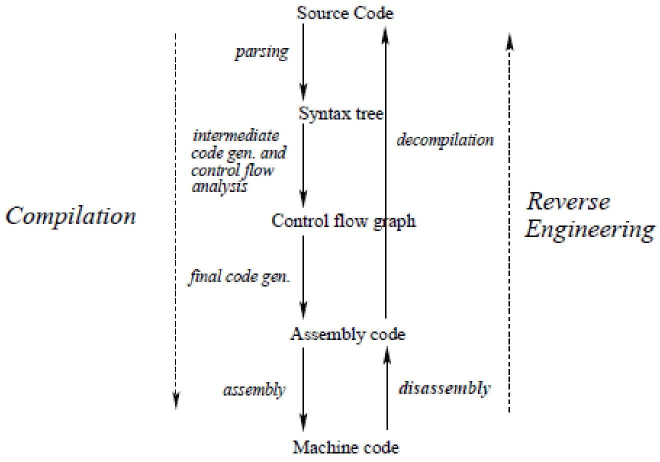


Figure 5: Compilation and Reverse Engineering [5]

5.2. Disassembly

The executable file contains a header, a section table, and different sections such as text, data, relocation section, etc. [6]. In addition, it contains information about the size of the executable file, location and size of each section, a stub program that will be executed if the program is run on MS-DOS (without Windows), etc. Converting this executable file (which the machine understands) into a file containing assembly code that a human being can understand is called disassembly.

There are two types of disassembly: static disassembly and dynamic disassembly. In static disassembly, the disassembler analyses the entire executable file and converts it into assembly code. In dynamic disassembly, the disassembler analyses only a few of the instructions which are to be executed and converted into assembly code. OllyDbg, when used without a debugger, is an example of static disassembly, however when it is used with debugger it is an example of dynamic disassembly. In static disassembly, the speed of disassembly is directly proportional to the size of executable file. However, in dynamic disassembly the size of the executable file does not affect the speed of disassembly. In this project we focus on static disassembly.

5.2.1. Types of Static Disassembly

There are three approaches used in static disassembly. They are linear sweep, recursive traversal, and hybrid disassembly.

5.2.1.1. Linear Sweep

In linear sweep the disassembler first finds the starting address of the program. After finding that address, the disassembler starts converting machine code into assembly code one by one. The linear sweep method does not take into consideration the control flow of executable program. objdump, part of GNU Binutils, is an example of a linear sweep disassembler [21]. The problem with this approach is that errors are not detected until an unknown machine code is encountered. Many viruses use special techniques to confuse linear sweep disassemblers.

5.2.1.2. Recursive Traversal

Unlike linear sweep, recursive traversal takes into account control flow of the program during disassembly of machine code into assembly code. This method starts disassembling the executable file and whenever it encounters jump instructions it follows that address and continues the process. When a conditional jump is encountered it takes into consideration both possible paths and generates assembly code. The main advantage of this method is that it is able to bypass the junk code in the executable code. According to [14] and [22], OllyDbg and IDA Pro use recursive traversal method for disassembling executable files.

5.2.1.3. Hybrid Disassembly

Both the linear sweep and recursive traversal methods described above sometimes do not disassemble the executable file correctly. This problem can be overcome by using a hybrid disassembly method. In hybrid disassembly method first the executable file is disassembled using linear sweep and then this disassembled code is verified using recursive traversal. If the verification passes then no change is made but if verification fails then that code is removed from subsequent optimization. The problematic code is inserted in the program after optimization has been applied to the remaining part of program. This approach will require updating the addresses within the machine code [11].

5.3. Intel Architecture Instruction Format

Figure 6 shows Intel 64 and IA-32 Architectures Instruction Format. Each instruction consists of optional instruction prefixes, opcode bytes, the ModR/M byte and the SIB (Scale-Index-Base) byte, a displacement, and an immediate data field.

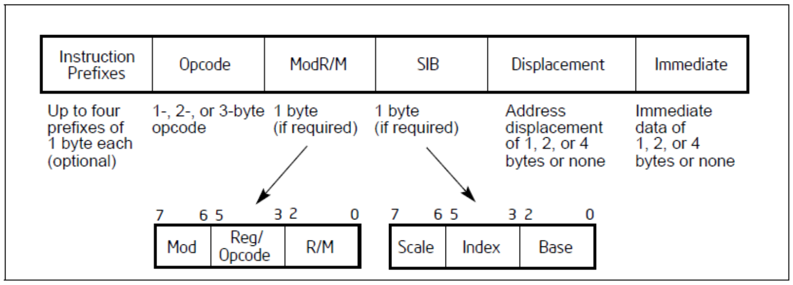


Figure 6: Intel 64 and IA-32 Architectures Instruction Format [10]

Instruction Prefixes

The instruction prefix is an optional part of instruction format and is divided into four groups. Each instruction can have at the most one prefix code from each group.

Group 1

F0H is used as lock prefix, F2H is used for encoding REPNE/REPNZ and F3 is used for encoding REP/REPE/REPZ

Group 2

2EH is used as CS segment override prefix, 36H is used as SS segment override prefix, 3EH is used as DS segment override prefix, 26H is used as ES segment override prefix, 64H is used as FS segment override prefix, 65H is used as GS segment override prefix. 2EH is used for branch not taken; 3EH is used for branch taken.

Group 3

66H is used as operand-size override prefix.

Group 4

67H is used as address-size override prefix.

Opcode

The opcode can be 1, 2 or 3 bytes in length. Opcode specifies the operation to be performed by the instruction. Sometimes 3 extra bits of opcode field are stored in ModR/M byte. The opcode field contains mandatory prefix, sign extension, displacement size, and register encoding.

ModR/M and SIB Bytes

ModR/M is of one byte. It contains information about the addressing mode and the registers used by the instruction. It consists of mod field, reg/opcode field and r/m field. SIB byte which follows ModR/M byte stands for Scale Index Base. It contains scale field which specifies factor, index field which specify particular index register and base field which specify particular base register. Following formula is used for calculating SIB value.

SIB value = (INDEX * 2^SCALE) + BASE

Displacement and Immediate Bytes

A displacement which follows optional SIB byte can be of 1, 2 or 4 bytes in length. The size of displacement is decided by Mod field.

Immediate field which follows displacement byte can be 1, 2 or 4 bytes in length. For instruction ADD BX, 0xFFFF the immediate field value is 0xFFFF.

Example

OR EAX, [ECX + EDX*2 + 508090B0h]

The above instruction does OR operation and is represented in the assembly code. In Table 1, we represent the same in the binary form.

Opcode	ModM/R	SIB	Displacement
00001011	10000100	01010001	10110000 10010000 10000000 01010000

Table 1: Binary Representation of OR Instruction

In the next section we discuss in detail dynamic programming and two different ways in which it can be implemented.

6. Dynamic Programming

Dynamic programming is an efficient method for solving problems that can be divided into smaller problems and which exhibit properties of overlapping sub problems. This method is usually used to solve search and optimization problems. Like divide and conquer method, the highly complex problems are divided into sub problems. However unlike divide and conquer, the dynamic programming takes advantage of overlapping sub problems. The method first solves the sub problems; stores the results of the sub problems, and use the results to solve more complex problems. For overlapping sub problems, dynamic programming is much better than divide and conquer since it only needs to solve each problem once.

Dynamic programming can be implemented in either of two ways: top down approach or bottom up approach [23].

6.1. Top Down Approach

This approach is used when we can apply recursion to solve the bigger problem. In top down approach we first try to look up and see if the problem is already solved. If it is not solved; we first solve it and store the result. If the problem is already solved we use the pre-computed result and solve the problem.

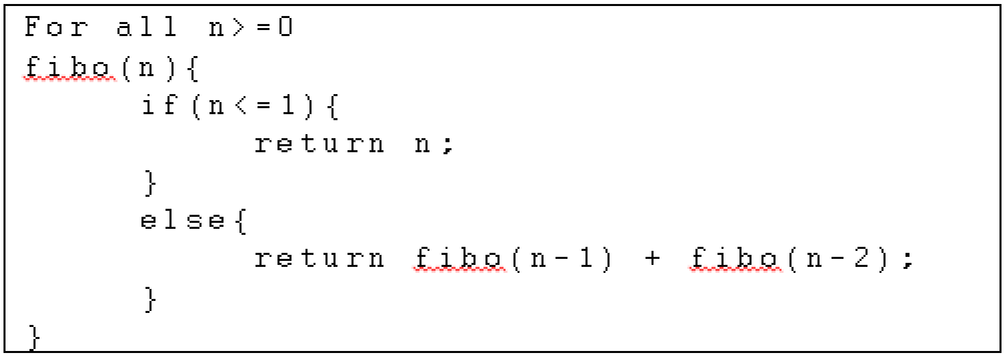
6.2. Bottom Up Approach

In this approach, we first divide the problem into sub problems recursively. Then we try to solve sub problems and store results into a table. We use the solutions of these sub problems to solve the bigger problem.

In the next section, we explain a simple problem and how it can be solved using dynamic programming.

6.3. Example

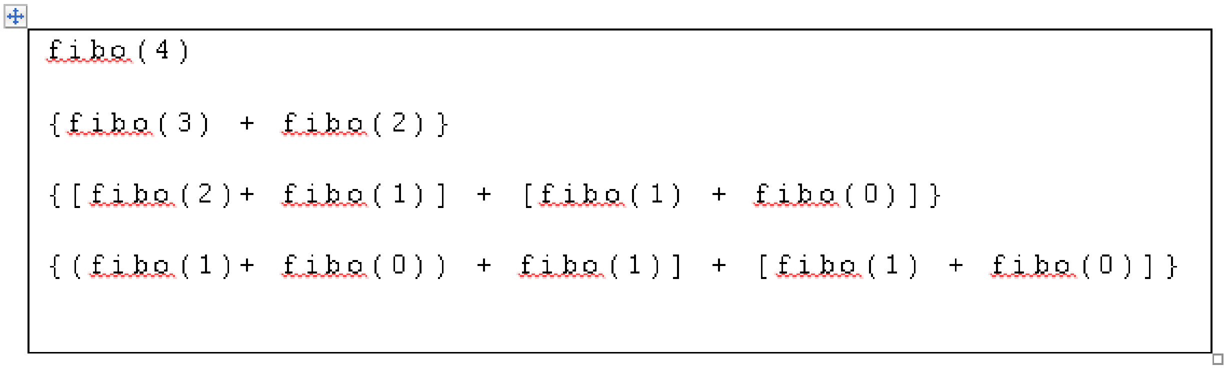
Consider the problem of finding the nth Fibonacci number where n is a whole number. The initial condition of the algorithm is fibo(0)=0 and fibo(1)=1. For any n, Fibonacci number is found by using equation fibo(n)=fibo(n-1)+fibo(n-2). The simple recursive implementation is shown in Figure 7.



```
For all n>=0
fibo(n){
        if(n<=1){
                return n;
        }
        else{
                return fibo(n-1) + fibo(n-2);
        }
}
```

Figure 7: Finding Fibonacci Numbers Recursively

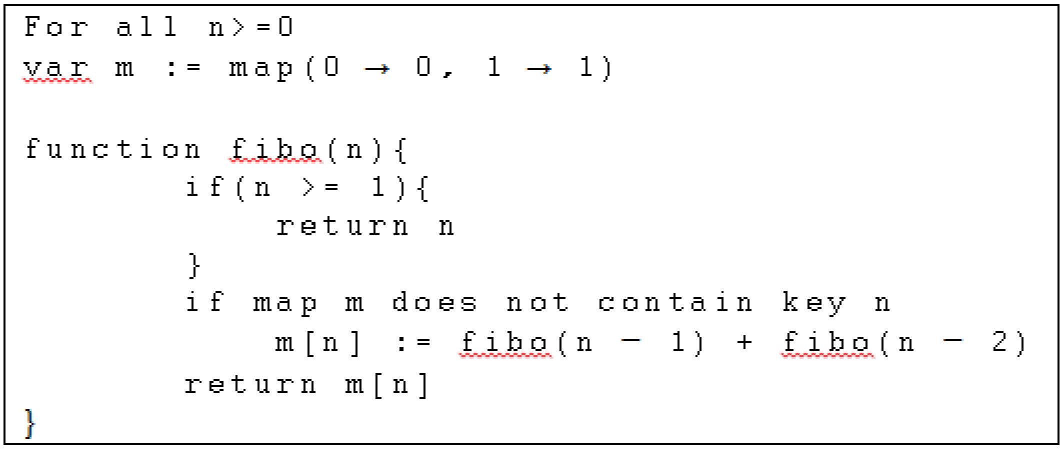
Here the problem at each stage is divided into smaller sub problems until it can no longer be divided. Smaller sub problems are solved and combined together to get solutions to the bigger problem. In general, the problem of calculating fib(n) depends on solving 2 sub problems: fib(n-1) and fib(n-2). The general solution tree for fibo(4) is shown in Figure 8.



```
fibo(4)

{fibo(3) + fibo(2)}

{[fibo(2)+ fibo(1)] + [fibo(1) + fibo(0)]}

{(fibo(1)+ fibo(0)) + fibo(1)] + [fibo(1) + fibo(0)]}
```

Figure 8: Solution Tree for Fibonacci Series

As shown in Figure 8, there is one sub problem that is calculated more than once. For example fibo(2) is calculated 2 times. We can optimize this problem by using dynamic programming. Consider the algorithm shown in Figure 9.



```
For all n>=0
var m := map(0 → 0, 1 → 1)

function fibo(n){
        if(n >= 1){
            return n
        }
        if map m does not contain key n
            m[n] := fibo(n − 1) + fibo(n − 2)
        return m[n]
}
```

Figure 9: Finding Fibonacci Numbers by Dynamic Programming

As shown in Figure 9, the same problem can be solved using dynamic programming. Since this problem exhibits the property of optimal substructure, first we divide the bigger problem into smaller problems. If we solve the smaller sub problem; we store the result. This result is later used if the same problem is encountered again. Since the problem of finding fibonacci number exhibits properties of optimal substructure and overlapping sub problems, we can use dynamic programming to solve it efficiently. In Big O notation, the naive recursion implementation takes exponential time while dynamic programming takes O(n) time. We use dynamic programming in our algorithm which is discussed in the next section.

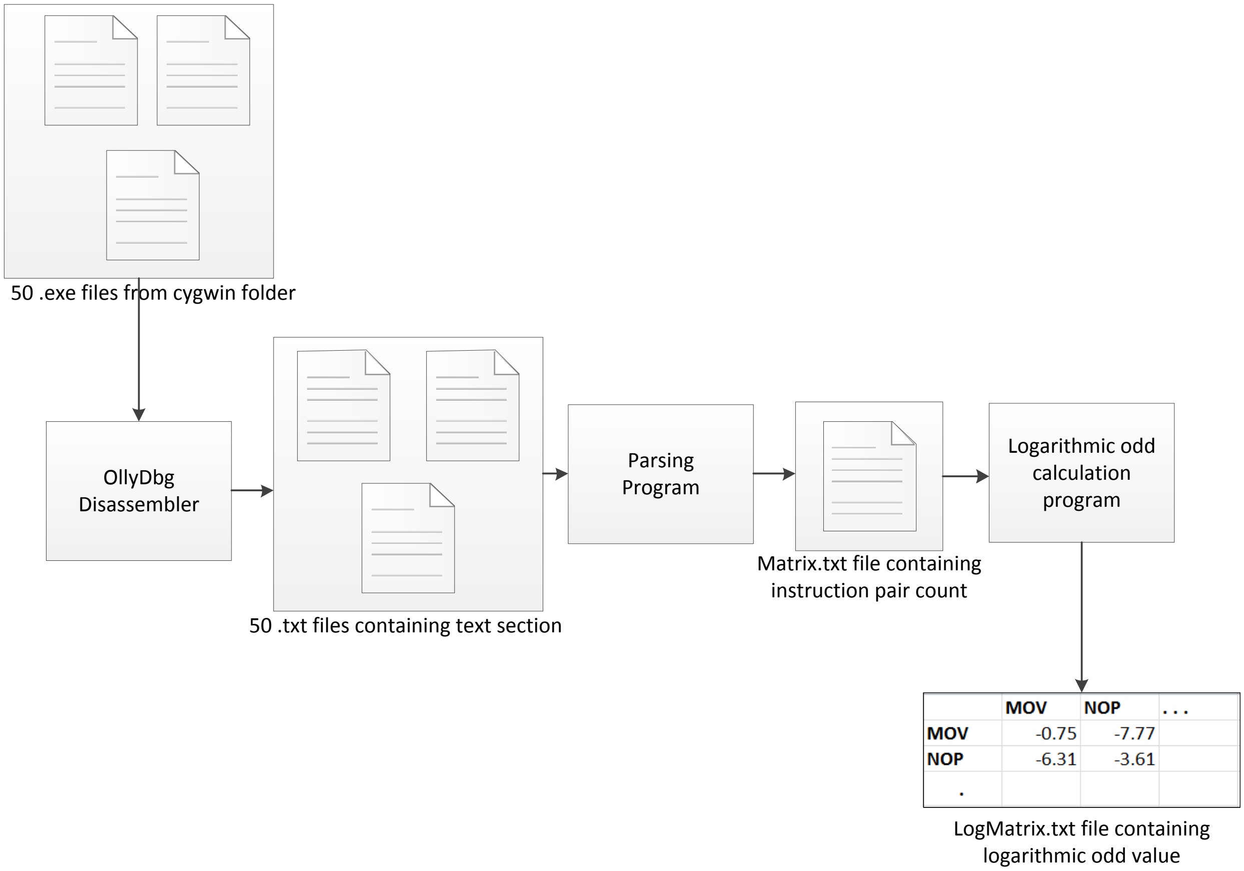
7. Our Algorithm

The aim of our project is to develop a program that can accurately predict assembly code from executable files. We use dynamic programming to predict assembly code from executable files. We took 50 .exe files from Cygwin folder, which had a size range from 300KB to 662KB. The size range was selected randomly. All these files were opened in OllyDbg individually and the .text section of each .exe file was extracted. We take only the .text section since it contains the program code. The text section was saved individually into 50 different .txt files. The name of the text file was kept the same as the exe file name. Each of the files was given as an input to the program, which generated a 2-dimensional table representing the count of pair of instructions. The first row and first column consisted of 14 instructions. All the other cells consisted of integer values representing the number of times instructions in row followed by instructions in column appeared.

	MOV	**NOP**	. . .
MOV	1344183	1765	
NOP	7582	111344	
⋮			

Table 2: Matrix of Instruction Occurrences

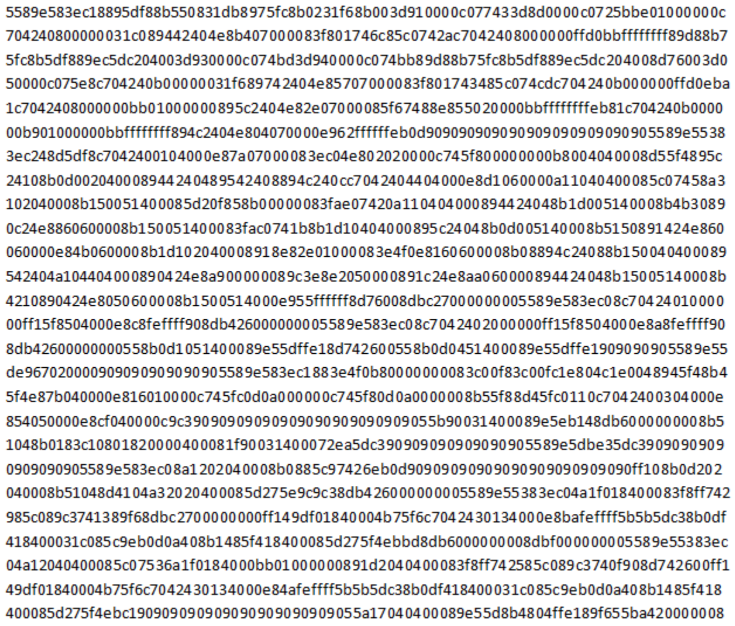
As shown in Table 2, the number of times MOV instruction occurs and a consecutive MOV instruction occurs is 1344183. Similarly the number of times NOP instruction occurs and another MOV instruction occurs is 7582. The most frequently occurring instructions in the .text section of the executable file were MOV, NOP, CAL, LEA, PUS, POP, JMP, TES, SUB, CMP, JE, JNZ, ADD and RET. Hence this project only takes these 14 instructions in consideration. Appendix A shows the matrix. Another program took a matrix of 50 files as input and added them all; giving a single large matrix. After addition, probability of each cell in the matrix was calculated and a new matrix was generated. The formula used was (value of a particular cell) / (Total value of all cells). It was then converted into odds using formula (p) / (1-p). The logarithmic odd for each cell was calculated. The reason we decided to take logarithmic odd instead of probability was that we can sum the log odds instead of trying to multiply probabilities, which might give underflow. Appendix B shows the final log odd matrix. The process described above is shown in Figure 10. It is important to notice that this process is performed only once for the generation of statistical data. Later we use the output of this process to determine assembly code instructions from executable file.


50 .exe files from cygwin folder
OllyDbg Disassembler
50 .txt files containing text section
Parsing Program
Matrix.txt file containing instruction pair count
Logarithmic odd calculation program
LogMatrix.txt file containing logarithmic odd value

	MOV	NOP	. . .
MOV	-0.75	-7.77	
NOP	-6.31	-3.61	
.			

Figure 10: Process Used for Matrix Generation

We used dynamic programming to solve the problem of finding assembly instructions from the executable file without disassembling it. First we extracted the .text section of executable files. We use the program mentioned in [22] for extraction of .text section whose output is a text file. The .text section that was originally in binary form is converted to a hexadecimal representation. Figure 11 shows part of the output of this program.



```
5589e583ec18895df88b550831db8975fc8b0231f68b003d910000c077433d8d0000c0725bbe01000000c
704240800000031c089442404e8b407000083f801746c85c0742ac7042408000000ffd0bbffffffff89d88b7
5fc8b5df889ec5dc204003d930000c074bd3d940000c074bb89d88b75fc8b5df889ec5dc204008d76003d0
50000c075e8c704240b00000031f689742404e85707000083f801743485c074cdc704240b000000ffd0eba
1c7042408000000bb01000000895c2404e82e07000085f67488e85502000bbffffffffeb81c704240b0000
00b901000000bbffffffff894c2404e804070000e962ffffffeb0d9090909090909090909090905589e5538
3ec248d5df8c7042400104000e87a07000083ec04e802020000c745f800000000b8004040008d55f4895c
24108b0d0020400089442404895424088894c240cc7042404404000e8d1060000a11040400085c07458a3
102040008b1500514000085d20f858b00000083fae07420a110404000894424048b1d005140008b4b3089
0c24e8860600008b1500514000083fac0741b8b1d10404000895c24048b0d005140008b5150891424e860
060000e84b0600008b1d102040008918e82e01000083e4f0e8160600008b08894c24088b1500404000089
542404a1044040008904 24e8a900000089c3e8e2050000891c24e8aa0600008944 24048b15005140008b
4210890424e8050600008b1500514000e955ffffff8d76008dbc27000000005589e583ec08c70424010000
00ff15f8504000e8c8feffff908db42600000000 5589e583ec08c704240200000ff15f8504000e8a8feffff90
8db4260000000558b0d1051400089e55dffe18d742600558b0d0451400089e55dffe1909090905589e55
de967020000909090909090905589e583ec1883e4f0b800000000 83c00f83c00fc1e804c1e0048945f48b4
5f4e87b040000e81601000 0c745fc0d0a000000c745f80d0a0000008b55f88d45fc0110c704240030400 0e
854050000e8cf04000 0c9c3909090909090909090909090909055b90031400089e5eb148db60000000008b5
1048b0183c10801820000400081f90031400072ea5dc3909090909090905589e5dbe35dc3909090909
0909090905589e583ec08a1202040008b0885c97426eb0d909090909090909090909090ff108b0d202
040008b51048d4104a32020400085d275e9c9c38db4260000000005589e55383ec04a1f018400083f8ff742
985c089c3741389f68dbc2700000000ff149df01840004b75f6c7042430134000e8bafeffff5b5b5dc38b0df
418400031c085c9eb0d0a408b1485f418400085d275f4ebbd8db6000000008dbf000000005589e55383ec
04a1204040008 5c07536a1f0184000bb01000000891d204040 0083f8ff742585c089c3740f908d742600ff1
49df01840004b75f6c7042430134000e84afeffff5b5b5dc38b0df418400031c085c9eb0d0a408b1485f418
400085d275f4ebc1909090909090909090909055a17040400089e55d8b4804ffe189f655ba42000 0008
```

Figure 11: Example .text Section of an Executable File

We then read the file containing hex codes and get the input stream. The input stream is then supplied to our main program. With the help of the matrix shown in Appendix B, our main program tries to determine the assembly instruction using dynamic programming. This process is shown in Figure 12.

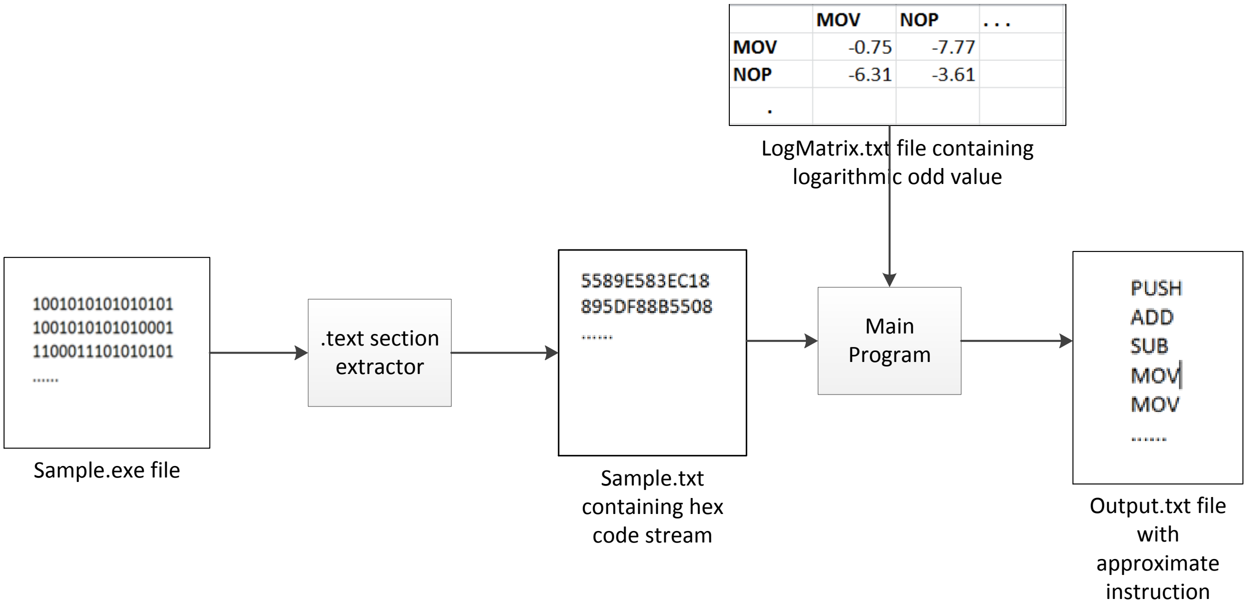


Figure 12: Using Our Algorithm for Generating Assembly Code from Executable File

During each iteration, the algorithm parses the input stream of length 2 or 4 and maintains 2 tables with the following information.

- Score – This column contains a decimal value which is updated after parsing each of the instructions. We use the table shown in Appendix B to update the score.
- Instruction Opcode – This column contains opcode of each instruction. The opcode are generated by refereeing [10]. The opcode is represented in hexadecimal format. For example instruction MOV BYTE PTR DS:[EDX],AL in the executable file is represented as 8802H where 88H is opcode while 02H is operand. The column keeps track of the opcode of instruction. Thus, for the above example, we store 88H.
- Length of instruction – This column contains all the possible lengths of the instruction. MOV instruction whose opcode is 89H can be of length 4, 6, 8, 12 or 14. We determined the length by looking at Intel® 64 and IA-32 Architectures Software

Developer's Manual Volume 2A: Instruction Set Reference, A-M and Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z [10]. For example consider instruction MOV EBP,ESP which is represented as 89E5 while instruction MOV DWORD PTR SS:[ESP],EAX is represented as 890424 in the executable file. The first instruction is of length 4 while later instruction is of length 6. We capture this information in this column.

- Flag to know if set or not-set – We use this flag to track if that opcode has occurred or not. This will help in building the solution path.
- Pointer for current location of input string – This column is initially set to 0. It keeps track of the location where next input stream should be parsed.
- Instruction name – This column represents the name of the instruction. We use 3 letters to represent each instruction uniquely. For example move instruction is represented as “MOV”; push instruction is represented as “PUS”.

7.1. Sample Input Stream

Consider stream 5589E583EC18895DF88B5508....... as input to our algorithm. This is the beginning part of the .text section of an executable file. The program first reads 55H from the input stream. It references the lookup table and finds that it is a PUSH instruction and of length 2. It now moves the pointer and reads the next data. Thus it reads 89H. From the lookup table the algorithm knows that it can represent opcode of a MOV instruction. The length can be 2,4,6,10,12 excluding 89H. This is represented in Figure 13 by the value between each node. The algorithm keeps track of the score between 2 nodes. Moreover the algorithm looks up in the matrix shown in Appendix B and finds that the logarithmic odd value of instruction MOV occurs after instruction PUSH is -3.91. This value is added to the path [initial value of the path is assumed to be 0] as shown in the Figure 13. In this way node numbers 2,3,4,5 and 6 are added to the possible solutions. For node number 2, the algorithm skips E5H (part of the input 5589E5...) since the operand length is 2 and thus the next input stream would be 83. Since it is 83H it grabs 2 more from input and hence it is 83EC which is opcode for SUB instruction. From the lookup table in Appendix B the value of SUB instruction occurring after MOV is -4.17. This value is added to -3.91 and the path value becomes -8.08. In a similar fashion, the algorithm continues to process the input stream and builds the possible path. In the end, the algorithm considers all the paths with large numbers of nodes and finds the value that is closest to 0.

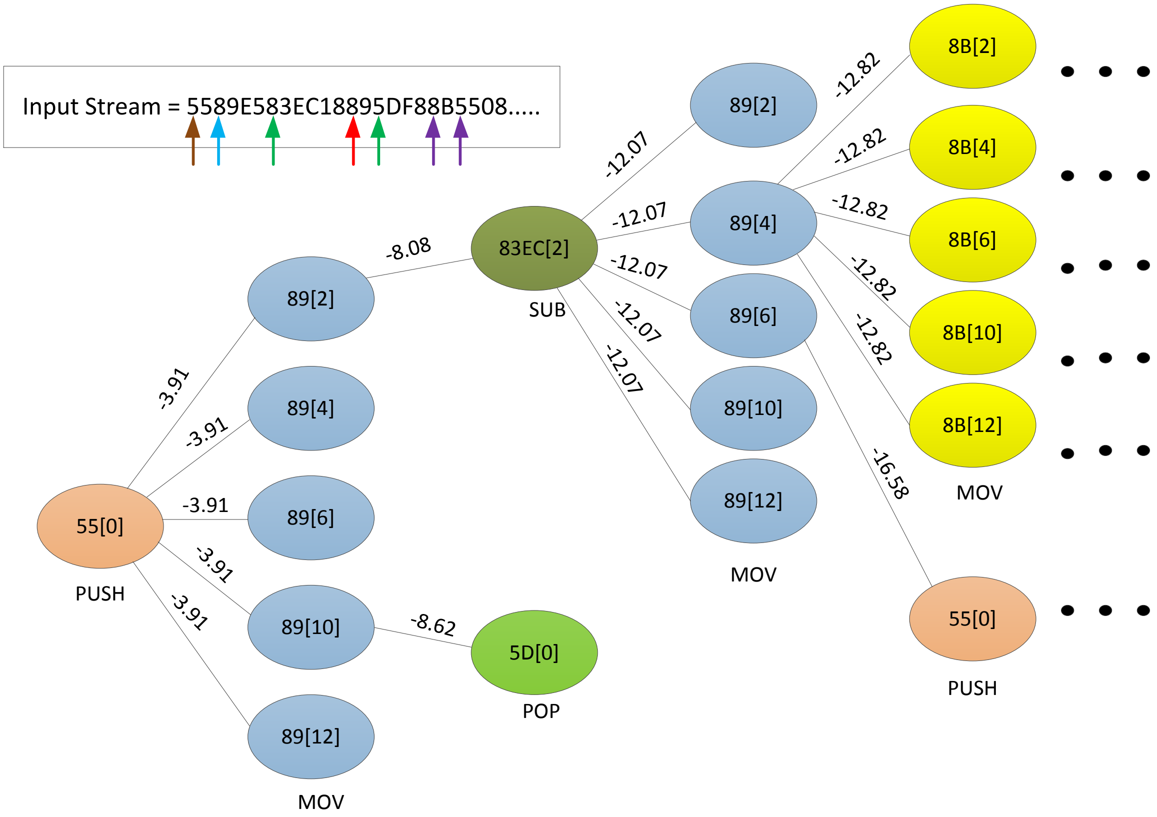


Figure 13: Solution by Our Algorithm

In this example, the nodes in the end i.e. 8BH have value -12.82 while 55H has -16.58. The algorithm selects the best path that is closest to 0. In our example, the path shown in red in Figure 14 is selected. Hence the expected instructions are PUSH, MOV, SUB, MOV, MOV.

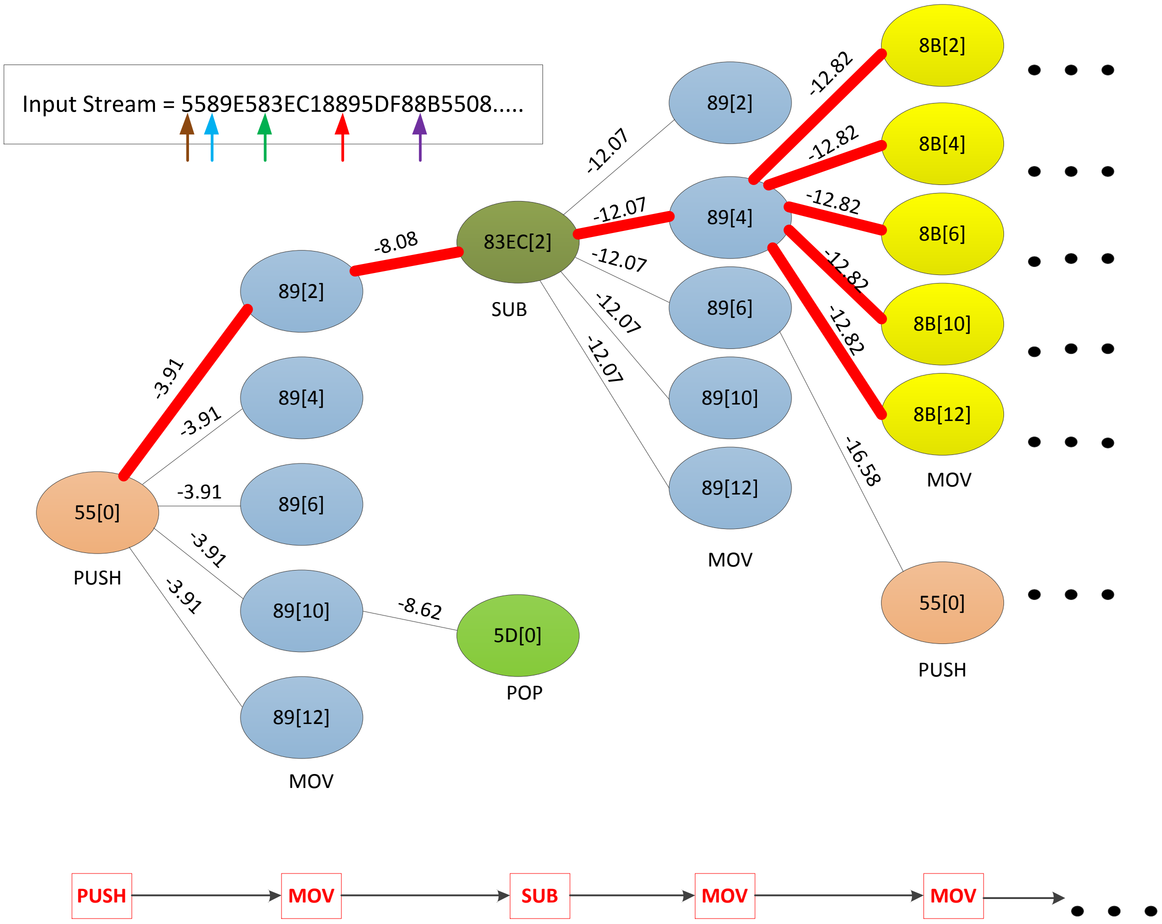


Figure 14: Solution Path

8. Test and Results

In this section, we discuss the tests we conducted to determine the speed and accuracy of our algorithm and its results.

8.1. Speed Test

The tests were performed on a Window 7 Home Premium, 64 bit operating system, Intel Core 2 Duo T6500 processor, 2.1GHz and 4GB RAM. The tests were conducted on 65 executable files with average file sizes ranging from 250KB to 3670KB. For each file, we measured the time taken to disassemble by a standard disassembler like OllyDbg v1.10 and also by our program.

Average File Size Range: 250KB to 1050KB

In this test, the total number of files is 20 and the average file size ranges from 250KB to 1050KB. The average time taken by our program for this file range to generate instructions is 1500 milliseconds. The average time taken by OllyDbg to disassemble the files is 1775 milliseconds. The time taken by OllyDbg is 18% more than the time taken by our program. For only some files of average size 1050KB the time taken by OllyDbg is slightly less than our algorithm, which might be due to a slight error in the calculation of time taken to disassemble through OllyDbg. The results of this test are shown in Figure 15.

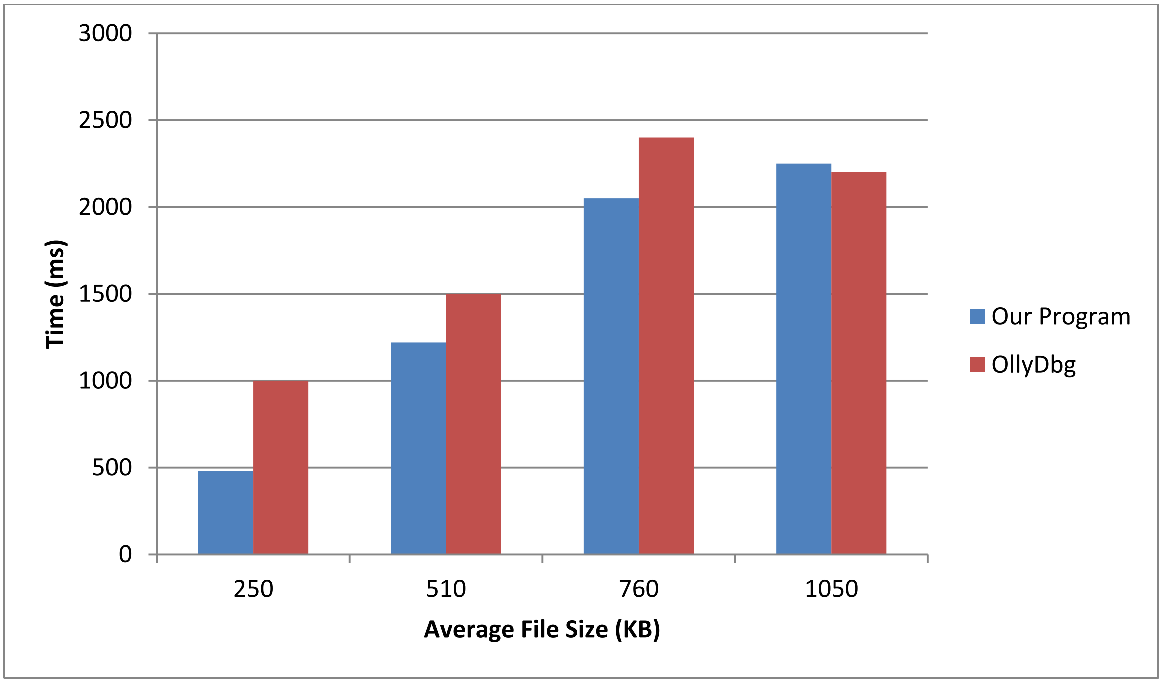


Figure 15: Time Comparison (250KB to 1050KB)

Average Files Size Range: 1210KB to 1850KB

In this test, the total number of files is 20 and the average file size ranges from 1210KB to 1850KB. The average time taken by our program to generate the instructions is 3963 milliseconds. The average time taken by OllyDbg to disassemble the files is 4750 milliseconds. The time taken by OllyDbg is 20% more than the time taken by our program. The results of this test are shown in Figure 16.

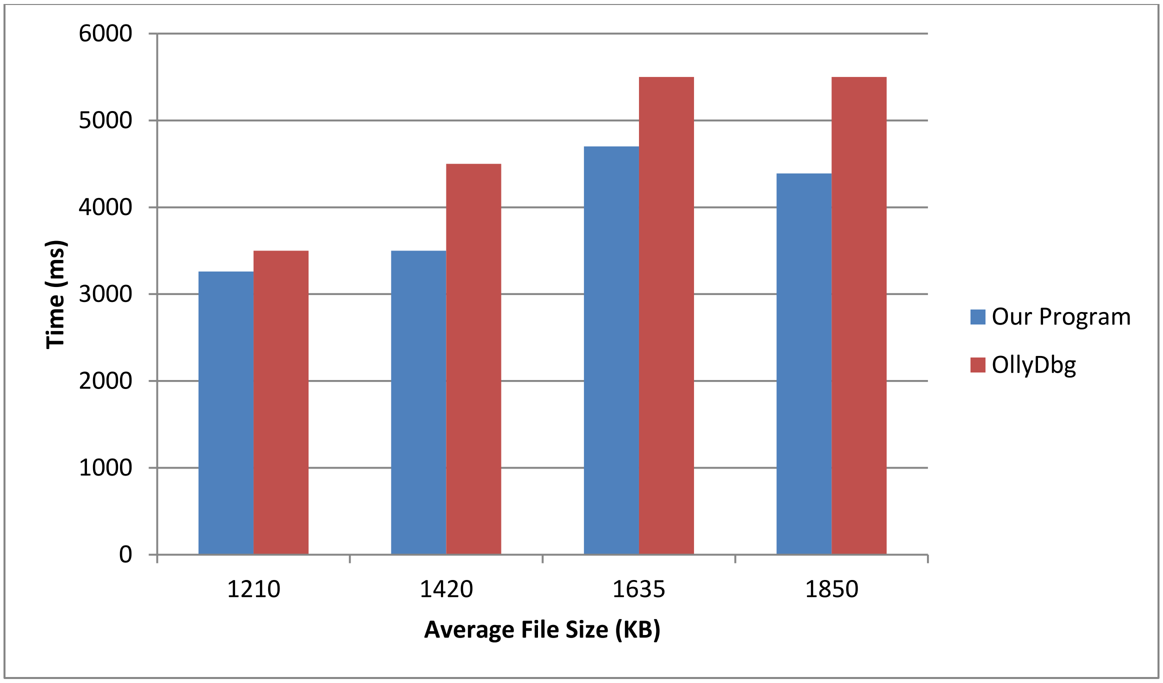


Figure 16: Time Comparison (1210KB to 1850KB)

Average File Size Range: 2455KB to 3670KB

In this test, the total number of files is 25 and the average file size ranges from 2455KB to 3670KB. The average time taken by our program to generate the instructions is 9588 milliseconds. The average time taken by OllyDbg to disassemble the files is 12800 milliseconds. The time taken by OllyDbg is 33% more than the time taken by our program. The results of this test are shown in Figure 17.

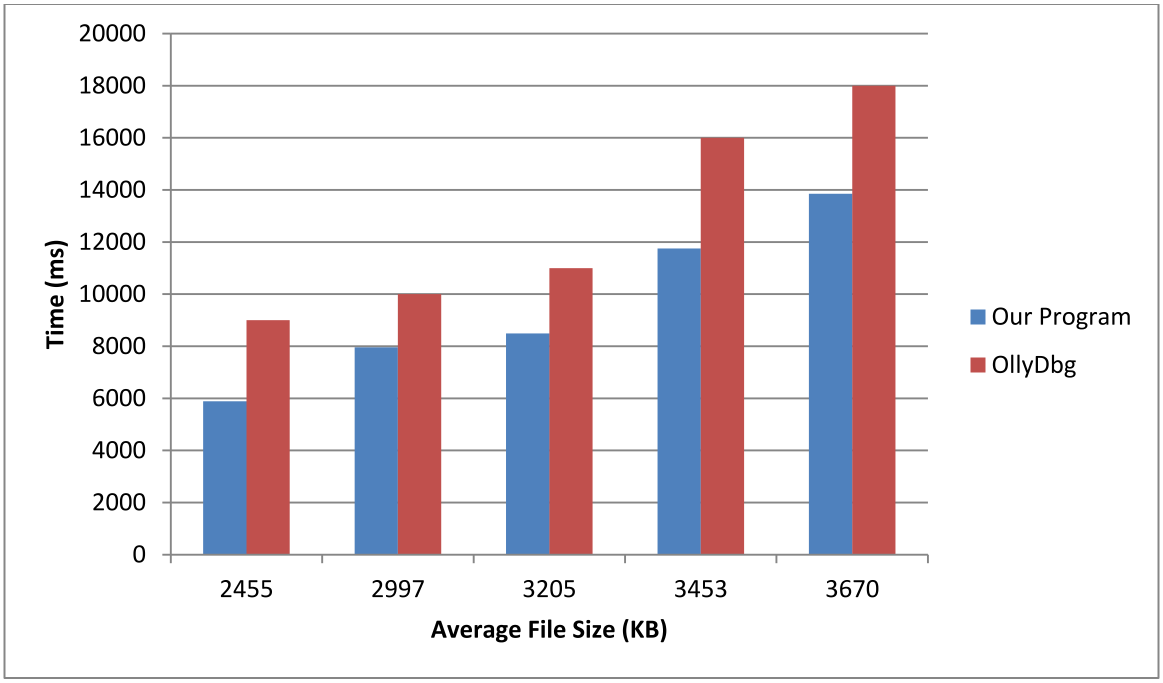


Figure 17: Time Comparison (2455KB to 3670KB)

Considering all the files of different sizes, the average time taken by our program is 5370 milliseconds, while that by OllyDbg is 6930 milliseconds. From Figure 18 it is clear that as average file size increases, the time taken by our program increases linearly, since we use dynamic programming in our solution.

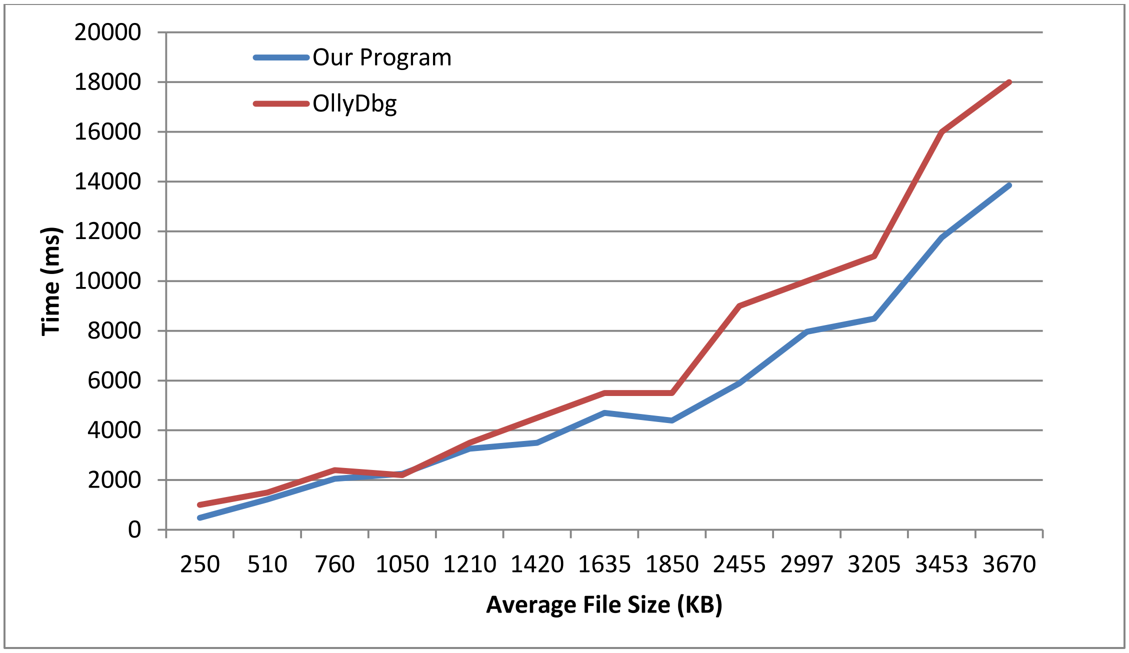


Figure 18: Average Time Comparison

8.2. Accuracy Test

In this section, we measure the accuracy of our program. First the executable file is opened in OllyDbg. The .text section of the executable file is extracted and copied in the text file. Later the executable file is supplied to our system, which first extracts the .text section and then determines the assembly instructions. We then compare the output from our algorithm to that from OllyDbg. We used Needleman–Wunsch algorithm [27] for the output comparison. This algorithm is generally used for in bioinformatics to align protein or nucleotide sequences and calculate the alignment score. We align the instructions first and at the end calculate the alignment score, which is a summation of the score where a match is found. If there is match to the instruction given by OllyDbg and our program, then we give it a score of 1. If there is no match between the instruction given by OllyDbg and our program, then we give it score of 0. We will not be taking gap penalty into account for our program. We use the tool described in [25] to generate the alignment score. From this we calculate accuracy. For example, if the number of instructions is 100 and alignment score is 93, then we conclude that the output given by our program is 93% accurate.

The same process is repeated for 55 executable files of different sizes and average accuracy is calculated. Figure 19 shows the result of the experiment.

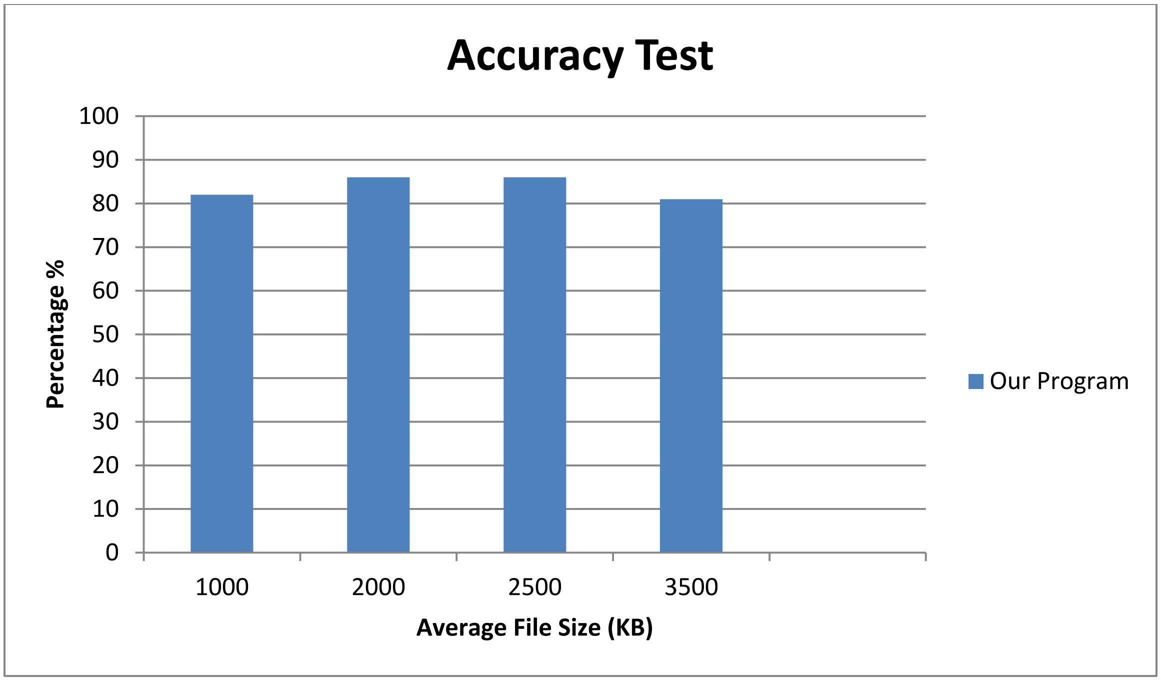


Figure 19: Accuracy Test

From Figure 19 it is clear that our algorithm is able to predict the instructions with a high level of accuracy. The average accuracy of our program is 83.75% when compared to the output of OllyDbg.

We also tested our results using monographic frequency counts. In this method we first disassembled the executable file using OllyDbg. Then we calculated the number of times each of 14 instructions appeared in the executable file and calculated the percentage of each instruction. Similarly we supplied the executable file to our program and calculated the frequency of each of 14 instructions and its percentage. We then calculated a chi-square statistic to determine if the distribution of observed frequencies obtained from our program differed from expected frequencies obtained from OllyDbg. The chi-square statistic is calculated as:

$\chi^2 = \sum [(O - E)^2 / (E)]$ ------ (1)

where χ^2 = chi-square statistic

O = Observed frequency

E = Expected frequency

Degree of freedom is given by (number of categories - 1). Using degree of freedom and the chi-square distribution table given in Appendix C, we determine the chi-square test for goodness of fit. We establish the null hypothesis, which is that the observed values are close to the expected values. The alternative hypothesis is that they are not close to the expected values. These hypotheses hold for all chi-square goodness of fit tests. If the calculated chi-square is less than the value in the table, then the null hypothesis is accepted and it is concluded that the predictions made were correct. Table 3 shows the calculation of chi-square statistics for sample an executable file of size is 17KB.

Instructions	% of Instructions determined by Our Program (Observed)	% of Instructions determined by OllyDbg (Expected)	$(O — E)^2/ E$
MOV	39	41	0.097560976
NOP	18	18	0
CAL	10	8	0.5
LEA	5	5	0
PUS	4	4	0
POP	4	4	0
JMP	3	3	0
TES	3	3	0
SUB	4	3	0.333333333
CMP	3	3	0
JE	3	3	0
JNZ	2	2	0
ADD	1	2	0.5
RET	1	1	0
Total	**100**	**100**	**1.430894309**

Table 3: Tabulated results calculating chi-square statistic

From equation (1), chi-square statistic χ2 = 1.4309 and degree of freedom = (14-1) = 13. Referring to the chi-square distribution table in Appendix C, the critical value for a chi-square at a probability level (alpha) = 0.05 and degree of freedom = 13 is 22.4. The critical value is greater than χ^2 and hence null hypothesis is accepted. Thus it passes the chi-square test for goodness of fit.

However, one can argue that the chi-square statistic does not give correct results if the frequencies are too low. Hence, in the above example we neglect instructions whose frequencies of occurrence are less than 10%.

Instructions	% of Instructions determined by Our Program (Observed)	% of Instructions determined by OllyDbg (Expected)	$(O — E)^2 / E$
MOV	39	41	0.097560976
NOP	18	18	0
CAL	10	8	0.5

Table 4: Tabulated results calculating chi-square statistic ignoring low frequency

From equation (1), chi-square statistic χ^2 = (0.0975 + 0 + 0.5) = 0.5975 and degree of freedom = (3-1) =2.

Referring to the chi-square distribution table in Appendix C, the critical value for a chi-square at a probability level (alpha) = 0.05 and degree of freedom = 2 is 5.99. The critical value is greater than χ^2 and hence null hypothesis is accepted. Thus it passes the chi-square test for goodness of fit. Since the test passes with and without considering low frequency, we can conclude that instructions determined by our program match closely with the instructions produced by Ollydbg.

We carried out the process described above i.e. with and without considering low frequency counts for files of different sizes ranging from 250 KB to 3670 KB. For all the files, it passes the chi-square test for goodness of fit taking alpha value as 0.05 which means that the output obtained from OllyDbg is statistically the same as the output obtained from our program.

9. Conclusions and Future Work

We implemented an algorithm to determine assembly instructions from an executable file. We first collected statistical data of the occurrence of one particular instruction after another by disassembling executable files from Cygwin folder. We extracted the .text section from the executable file. We determined the assembly instructions using collected statistical data, opcode of known instructions and dynamic programming. We also determined that our algorithm is much faster than OllyDbg. The time taken by our algorithm linearly increases with the size of the file. Moreover the accuracy of our algorithm is 83.75% when compared to that of OllyDbg.

We used the 14 most commonly occurring instructions to generate a 2x2 table. This table is used in the algorithm to score and find the best path and ultimately the instructions. This can be extended to include all the instructions documented in Intel® 64 and IA-32 Architectures Software Developer's Manual. This can speed up the algorithm and improve the accuracy of instructions. Moreover, for our experiment we only extracted the .text section, which holds the program code of an executable file. It can be expanded to include the .data section, which holds variables.

10. References

[1] Konstantinou, E. (2008). *Metamorphic Virus: Analysis and Detection*. http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf

[2] Szor, P., & Ferrie, P. (2005). *Hunting For Metamorphic.* http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

[3] Nachenberg, C. (1996). *Understanding and managing polymorphic virus.* http://www.symantec.com/avcenter/reference/striker.pdf

[4] Wong, W., & Stamp, M. (2006). *Hunting for metamorphic engines.* Springer-Verlag France 2006

[5] Linn, C., & Debray, S. (2003). Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and Communications Security* (CCS '03). ACM, New York, NY, USA, 290-299.

[6] An In-Depth Look into the Win32 Portable Executable File Format (2002). Retrieved September 10, 2010, from Microsoft: http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[7] Stamp, M. (2004). *A Revealing Introduction to Hidden Markov Models*. http://www.cs.sjsu.edu/faculty /stamp/RUA/HMM.pdf

[8] Dasgupta, S., Papadimitriou, C. H., & Vazirani, U.V. (2008). *Algorithms.* Boston: Mcgraw-Hill Higher Education.

[9] 20bits by Jesse Farmer (2007). Retrieved August 1, 2010, from http://20bits.com/articles/introduction-to-dynamic-programming

[10] Intel 64 and IA-32 Architectures Software Developer’s Manual. Retrieved August 3,2010, from http://www.intel.com/products/processor/manuals/

[11] Schwarz, B., Debray, S., & Andrews, G. (2002). Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)* (WCRE '02). IEEE Computer Society, Washington, DC, USA, 45-.

[12] Cohen, F. (1987). Computer viruses: theory and experiments. *Computers and Security.* 6, 1(February 1987), 22-35. DOI=10.1016/0167-4048(87)90122-2 http://dx.doi.org/10.1016/0167-4048(87)90122-2

[13] Stuxnet worm 'targeted high-value Iranian assets'. (2010). Retrieved October 20,2010, from BBC: http://www.bbc.co.uk/news/technology-11388018

[14] OllyDbg (2010). Retrieved September 7, 2010, from OllyDbg: http://www.ollydbg.de/

[15] Desai, P. (2008). *Towards an Undetectable Computer Virus.* http://www.cs.sjsu.edu/faculty/stamp/students/Desai_Priti.pdf

[16] Govindaraj, S. (2008). *Practical Detection of Metamorphic Computer Viruses.* http://www.cs.sjsu.edu/faculty/stamp/students/Govindaraj_Sharmidha.pdf

[17] Hidden Markov Models (2010). Retrieved August 17, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Hidden_Markov_model

[18] File Signature (2007). Retrieved July 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/File_signature

[19] Attaluri, S. (2007). *Detecting Metamorphic Viruses Using Profile Hidden Markov Models.* http://www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pdf

[20] Szor, P. The new 32-bit medusa. *Virus Bulletin*, pages 8-10, December 2000.

[21] GNU Project – Free Software Foundation, objdump. Retrieved November 26, 2010, from GNU Manuals Online: http://sourceware.org/binutils/docs-2.20/binutils/objdump.html#objdump

[22] IDAPro (2010). Retrieved September 7, 2010, from IDAPro: www.hex-rays.com/idapro/

[23] Dynamic Programming (2002). Retrieved August 27, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Dynamic_Programming

[24] What is the chi-square statistic? (2006). Retrieved November 28, 2010, from Connexions: http://cnx.org/content/m13487/latest/

[25] Sequences studio. Retrieved November 11, 2010, from sourceforge.net: http://sstu.sourceforge.net/index_SF.htm

[26] Gusfield, D. (1997). *Algorithms on strings, trees and sequences.* Cambridge University Press.

[27] Needleman-Wunsch algorithm (2004). Retrieved November 7, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm

11. Appendix

11.1. Appendix A: Table Containing Count for Pair of Instructions

	MOV	NOP	CAL	LEA	PUS	POP
MOV	1344183	1765	262700	77861	45236	37350
NOP	7582	111344	262	14888	13883	130
CAL	171233	2782	8854	14109	1070	2734
LEA	100565	182	8065	31677	37937	4595
PUS	82778	3	11135	1941	32341	52
POP	6281	9	2	237	1093	71716
JMP	79697	24909	4548	23823	1725	250
TES	25762	41	1	775	98	33
SUB	76733	44	2455	5073	4888	203
CMP	20877	278	1	1261	288	152
JE	112722	1359	2192	7614	395	1003
JNZ	66501	1162	1677	5029	140	1211
ADD	55716	257	1383	5775	4713	18093
RET	16894	14386	1453	21017	13329	868

Table 5: Count for Pair of Instructions (Part 1)

	TES	SUB	CMP	JE	JNZ	ADD	RET
MOV	95626	63262	80791	23981	9410	68926	4346
NOP	428	365	799	141	77	5201	27
CAL	34269	2742	12309	0	3	15469	55
LEA	3175	6540	10575	711	483	5555	151
PUS	124	22292	70	39	6	1555	44
POP	53	132	62	324	28	1256	52414
JMP	2451	2376	8442	14	17	2235	2
TES	3	0	6	86572	55127	142	3
SUB	1359	1797	9970	177	88	2796	130
CMP	4	12	113	64708	34448	226	0
JE	7997	2652	22705	66	35	3806	3
JNZ	5201	740	7248	20	16	3002	2
ADD	4170	4539	10677	474	205	40876	1631
RET	544	338	1578	20	9	892	2

Table 6: Count for Pair of Instructions (Part 2)

11.2. Appendix B: Table Containing Log Odds for Pair of Instructions

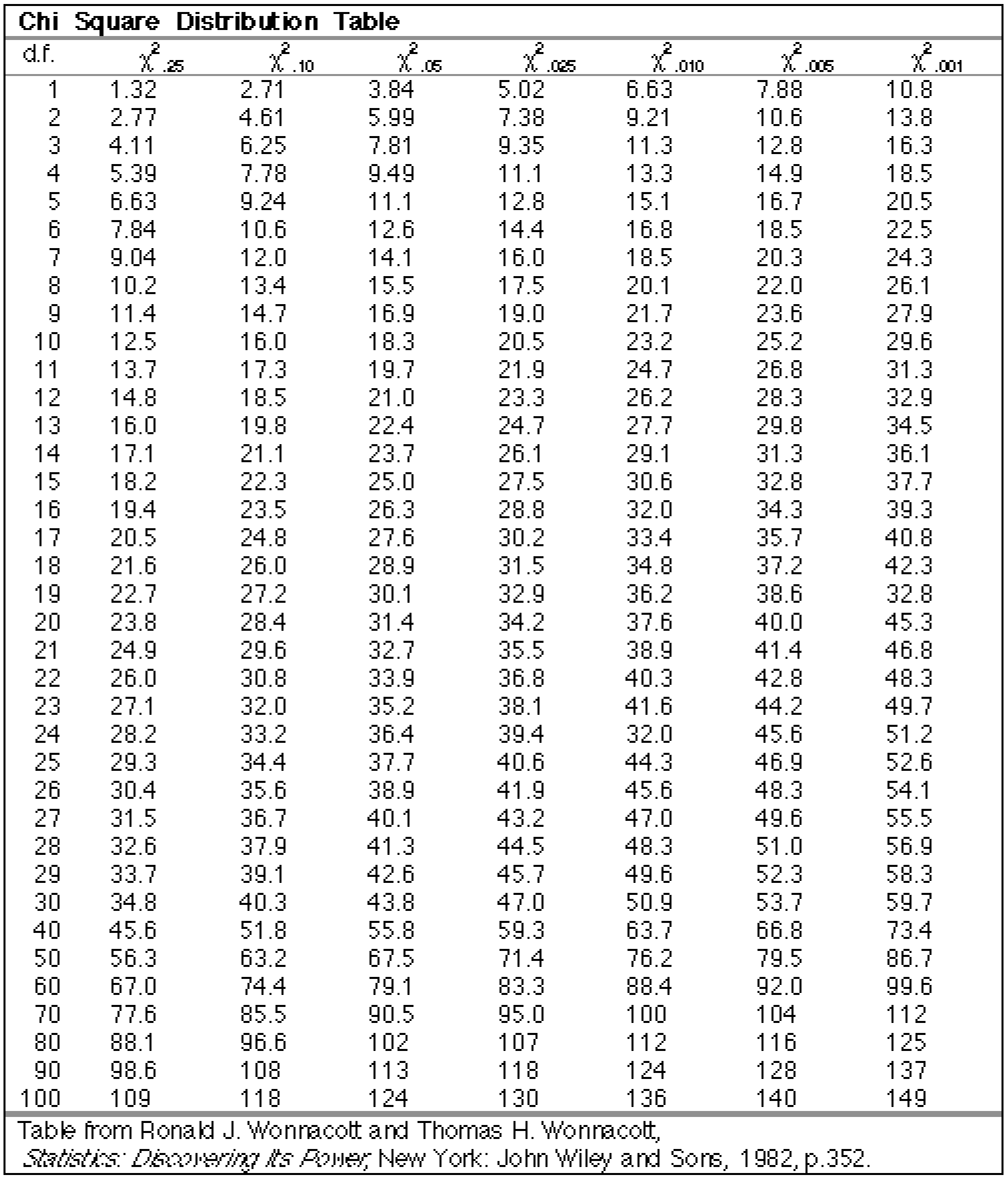
	MOV	NOP	CAL	LEA	PUS	POP	JMP
MOV	-0.75129151904491	-7.77271836370932	-2.7055845321917	-3.96762322902811	-4.51855101818601	-4.71201056995588	-4.00039294771691
NOP	-6.31370334102804	-3.60175507122273	-9.68063832813251	-5.63717793636319	-5.70730886744794	-10.3814798602256	-7.1839649844503
CAL	-3.15657475640606	-7.31745631062545	-6.15830710134644	-5.69110697241581	-8.27337619450129	-7.33487211965479	-5.2063471980019
LEA	-3.70621233369479	-10.0449952229882	-6.25183125290145	-4.87811665180808	-4.69627534578186	-6.81522493153474	-7.45117344685566
PUS	-3.90519066249299	-14.1504323074871	-5.9285378978875	-7.67762376159906	-4.85721218381966	-11.2977891927358	-9.97599925132316
POP	-6.50226191064342	-13.0518185880298	-14.5558976540599	-9.78092865272613	-8.25210314768194	-4.05132651314344	-6.34879577748766
JMP	-3.94387026752639	-5.12010322450171	-6.82551733151786	-5.16494139387068	-7.7956515461628	-9.72752477578027	-8.21262564687862
TES	-5.08622722384502	-11.5354634677517	-15.2490450730844	-8.59599745511516	-10.6640544630826	-11.7525298807225	-9.06684498134488
SUB	-3.98249059036235	-11.4648451851369	-7.44257766949879	-6.71614721117076	-6.75334051627934	-9.93579092303844	-8.47118932414622
CMP	-5.29765110067038	-9.62135790252502	-15.2490450730844	-8.10908422660714	-9.5860161512736	-10.2251285434391	-7.3095741852385
JE	-3.58911734477546	-8.03421677159556	-7.55595281259616	-6.30948405544621	-9.270065348727	-8.33805531506799	-7.10412547764447
JNZ	-4.12808783307532	-8.19087023996975	-7.82388356474672	-6.72486891765747	-10.30736950335	-8.14955474576016	-6.35911103163553
ADD	-4.30764733228578	-9.69990793939476	-8.01670512906182	-6.58637369445226	-6.78984086932419	-5.44144103643568	-5.95056867787705
RET	-5.51029474695566	-5.67159807274295	-7.96731309899371	-5.29093398852419	-5.74816436425196	-8.4826465882627	-8.03127679988243

Table 7: Log Odds for Pair of Instructions (Part 1)

	TES	SUB	CMP	JE	JNZ	ADD	RET
MOV	-3.7577778558957	-4.17880451001309	-3.92997066867103	-5.1582931509265	-6.09727060642954	-4.09168358558255	-6.87099724654045
NOP	-9.18982004794497	-9.34906091462947	-8.56549381447954	-10.3002517971086	-10.9052215277589	-6.69119809406928	-11.953202006982
CAL	-4.79884336326152	-7.3319483678539	-5.82801975437029	0	-14.1504323074871	-5.59875638326331	-11.2416990106821
LEA	-7.18522498861558	-6.46179209177736	-5.98027236036451	-8.68220331909892	-9.06891347290096	-6.62526606092949	-10.2317294659436
PUS	-10.4287341759066	-5.23173213242814	-11.0005333768441	-11.5854743652599	-13.4572844115328	-7.89944360584315	-11.4648451851369
POP	-11.2787407592976	-10.3662119111509	-11.1218961415945	-9.46822453026811	-11.916834124345	-8.11305840822444	-4.36953865411523
JMP	-7.44420928061565	-7.4753048787993	-6.20605563870583	-12.6099846434249	-12.4158279135878	-7.53651569422682	-14.5558976540599
TES	-14.1504323074871	0	-13.4572844115328	-3.85945317114158	-4.31841741355554	-10.2931843914129	-14.1504323074871
SUB	-8.03421677159556	-7.75474281218635	-6.03932912663443	-10.0728533698648	-10.7716875119731	-7.31243323903005	-10.3814798602256
CMP	-13.8627499965706	-12.7641358001827	-10.5216305459831	-4.15585433894936	-4.79359054029018	-9.82845641784348	0
JE	-6.26031473983025	-7.36534335228878	-5.21327582050379	-11.0593748307406	-11.6936889037666	-7.00380325562496	-14.1504323074871
JNZ	-6.69119809406928	-8.64221864603631	-6.35883457761307	-12.2533082686933	-12.4764527738696	-7.24129517251662	-14.5558976540599
ADD	-6.9123791049777	-6.82750033244031	-5.97064880492973	-9.08772495128403	-9.92598644598958	-4.620951645574	-7.85170769764496
RET	-8.94996633158059	-9.42591881181064	-7.88475544231717	-12.2533082686933	-13.0518185880298	-8.45536644599337	-14.5558976540599

Table 8: Log Odds for Pair of Instructions (Part 2)

11.3. Appendix C: Chi-square Distribution Table



Chi Square Distribution Table

d.f.	$\chi^2_{.25}$	$\chi^2_{.10}$	$\chi^2_{.05}$	$\chi^2_{.025}$	$\chi^2_{.010}$	$\chi^2_{.005}$	$\chi^2_{.001}$
1	1.32	2.71	3.84	5.02	6.63	7.88	10.8
2	2.77	4.61	5.99	7.38	9.21	10.6	13.8
3	4.11	6.25	7.81	9.35	11.3	12.8	16.3
4	5.39	7.78	9.49	11.1	13.3	14.9	18.5
5	6.63	9.24	11.1	12.8	15.1	16.7	20.5
6	7.84	10.6	12.6	14.4	16.8	18.5	22.5
7	9.04	12.0	14.1	16.0	18.5	20.3	24.3
8	10.2	13.4	15.5	17.5	20.1	22.0	26.1
9	11.4	14.7	16.9	19.0	21.7	23.6	27.9
10	12.5	16.0	18.3	20.5	23.2	25.2	29.6
11	13.7	17.3	19.7	21.9	24.7	26.8	31.3
12	14.8	18.5	21.0	23.3	26.2	28.3	32.9
13	16.0	19.8	22.4	24.7	27.7	29.8	34.5
14	17.1	21.1	23.7	26.1	29.1	31.3	36.1
15	18.2	22.3	25.0	27.5	30.6	32.8	37.7
16	19.4	23.5	26.3	28.8	32.0	34.3	39.3
17	20.5	24.8	27.6	30.2	33.4	35.7	40.8
18	21.6	26.0	28.9	31.5	34.8	37.2	42.3
19	22.7	27.2	30.1	32.9	36.2	38.6	32.8
20	23.8	28.4	31.4	34.2	37.6	40.0	45.3
21	24.9	29.6	32.7	35.5	38.9	41.4	46.8
22	26.0	30.8	33.9	36.8	40.3	42.8	48.3
23	27.1	32.0	35.2	38.1	41.6	44.2	49.7
24	28.2	33.2	36.4	39.4	32.0	45.6	51.2
25	29.3	34.4	37.7	40.6	44.3	46.9	52.6
26	30.4	35.6	38.9	41.9	45.6	48.3	54.1
27	31.5	36.7	40.1	43.2	47.0	49.6	55.5
28	32.6	37.9	41.3	44.5	48.3	51.0	56.9
29	33.7	39.1	42.6	45.7	49.6	52.3	58.3
30	34.8	40.3	43.8	47.0	50.9	53.7	59.7
40	45.6	51.8	55.8	59.3	63.7	66.8	73.4
50	56.3	63.2	67.5	71.4	76.2	79.5	86.7
60	67.0	74.4	79.1	83.3	88.4	92.0	99.6
70	77.6	85.5	90.5	95.0	100	104	112
80	88.1	96.6	102	107	112	116	125
90	98.6	108	113	118	124	128	137
100	109	118	124	130	136	140	149

Table from Ronald J. Wonnacott and Thomas H. Wonnacott,
Statistics: Discovering Its Power, New York: John Wiley and Sons, 1982, p.352.

Table 9: Chi-square Distribution Table