Spring 2011

Metamorphic Detection via Emulation

Sushant Priyadarshi
San Jose State University

Metamorphic Detection via Emulation

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 Computer Science

by
Sushant Priyadarshi
May 2011

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

METAMORPHIC DETECTION VIA EMULATION

by
Sushant Priyadarshi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp	Department of Computer Science	Date
Dr. Chris Pollett	Department of Computer Science	Date
Dr. Johnny Martin	Department of Computer Science	Date

APPROVED FOR THE UNIVERSITY

Associate Dean	Office of Graduate Studies and Research	Date

ABSTRACT

Metamorphic Detection via Emulation

by Sushant Priyadarshi

In parallel with improvements in anti-virus technologies, computer virus writers have developed innovative viruses that are challenging to detect. Metamorphic viruses change their appearance from one generation to another by using various code obfuscation techniques. Today, signature detection is the most common method used in anti-virus products, but well designed metamorphic viruses cannot be detected using signatures. Hence, there is a need for a more robust anti-virus technology.

To counter metamorphic virus, a very successful tool based on hidden Markov models (HMM) has been previously developed. This tool was able to detect all hacker produced metamorphic viruses on which it was tested. However, a weakness of this tool was exploited to develop an advanced metamorphic virus generator. These morphed viruses, which were not detected by the HMM based technique or standard signature-based detection, rely on carefully selected dead code insertion for their success.

In this project, we have created a code emulator designed specifically to detect dead code in any virus file. The output of this code emulator is then used to enhance HMM-based detection of metamorphic viruses. We test our emulator on the previously mentioned metamorphic generator, using the existing HMM detector to determine the quality of our results.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mark Stamp for his encouragement and guidance in carrying out this project. I also extend my heartfelt thanks to my family and friends for being a wonderful support.

Table of Contents

List of Figures

List of Tables

1 Introduction

A computer virus is a computer program that can copy itself and infect another program [7]. A virus in an executable code form can spread from one network/system to another [12]. Once a virus attaches itself to a program, each time the program runs, the virus file is triggered and is executed on the host machine. This process can result in additional infections.

In general, viruses can be classified based on target and concealment strategies [9]. Viruses based on target can be Boot-Sector Infectors, File Infectors and Macro Infectors. And different strategies on which viruses are based upon are encryption, stealth, oligomorphism, polymorphism and metamorphism. Virus is typically used to describe other type of malwares such as Trojan horses, worms, etc [31].

Anti-virus techniques include both static and dynamic approaches [9]. These techniques have relative weaknesses and strengths and the effective combination of these techniques can yield stronger detection. Scanners, Static Heuristics and Integrity Checkers form the static approach whereas Behavior Monitors and Emulation form the dynamic approach in anti-virus techniques.

Signature detection is the most common method implemented in anti-virus products [32]. A signature is essentially a “bit pattern” which is characteristic of a given virus family [33]. Ideally, the signature is not common in other software. Signature detection is relatively fast and effective, but it cannot detect new and unknown viruses, since signatures must be available prior to the detection. Since signature detection is the most popular technique, virus writers have developed many innovative techniques to evade signature detection. The most advanced such technique is the use of metamorphic code that has the ability to morph its internal structure (but retain its function) at each infection. Well designed metamorphic viruses cannot be detected using signatures, since there is no common signature available.

The aim of this project is to develop an anti-virus mechanism based on code emulation, and specifically aimed at improved metamorphic detection. The advanced metamorphic virus generator in [16] injects dead/junk code from non-virus files into its morphed copies, which makes signature detection fail. This code injection also causes the HMM-based detection in [8]

to fail, which is noteworthy since the technique in [8] was able to successfully detect all hacker produced metamorphic viruses on which it was tested.

The emulator developed for this project will implement a virtual machine that will be used to record the execution of a virus file in a simulated environment and thereby remove the dead code. To test the effectiveness of our emulator, the output of this virtual machine will then be used as input to the HMM tool developed in [8].

This paper contains the following section:

- Section 2 contains the evolution of computer viruses and their types.
- Section 3 discusses over the various anti-virus techniques.
- Section 4 shows various code obfuscation techniques.
- Section 5 deals with the HMM about its overview, example and how HMM is used as anti-virus.
- Section 6 and 7 discusses about the metamorphic engines developed in [21] and [16].
- Section 8 gives the details of code emulator like architecture, algorithm and implementation.
- Section 9 shows all the experiments and their respective analysis.
- Section 10 discusses few weaknesses of our code emulator
- Section 11 draws conclusions and also discusses future enhancements

2 Virus Evolution

The evolution of virus started with an academic project done by Fred Cohen in 1983 after which Len Andleman came up with the term "virus" [9]. Cohen is also considered as the "father of computer viruses" though there were viruses before this period. One of the first successful viruses was the "Creeper Virus" which was written by Bob Thomas in 1971. Creeper was able to make copy of itself and propagate through ARPANET [12].

As the internet usage increased, more and more viruses started pouring into the network and infecting computers all over the world at very high rate. According to network security experts,

2003 was the “year of worm” [13]. There has been surge in number of viruses and also in research of the anti-virus development.

2.1 Stealth Virus

Stealth viruses use a smart approach to defeat anti-virus products. It basically intercepts all the calls made by the anti-virus programs to the host machine’s operating system and then returns back the instance of a “clean” file. Frodo, Whale and Brain are some of the more popular stealth viruses [9].

2.2 Encrypted Virus

One of the advanced methods that the virus writers use to hide their viruses is by encrypting the virus body with different keys. So, a virus file will have two parts in it – the encrypted body and the decrypting module [28]. Since the virus is being encrypted with different encrypting keys each time, a virus scanner based on signature detection cannot detect it. The only way out is to do an indirect detection by detecting the decrypting module which will always remain constant. For example, a simple XOR operation of each byte of a virus file with a key will encrypt the virus file. And again applying XOR operation on the encrypted virus file will decrypt it [8].

2.3 Polymorphic Virus

Polymorphic virus is just like an encrypted virus with the difference being in the decryption module. The decryption module also gets changed/mutated after each infection and thus there is no common part between different copies of same virus [30]. Also, polymorphic viruses can generate many unique decryptors and can use many other encryption methods for encryption [8]. The Figure 1 illustrates various polymorphic virus variants [15].

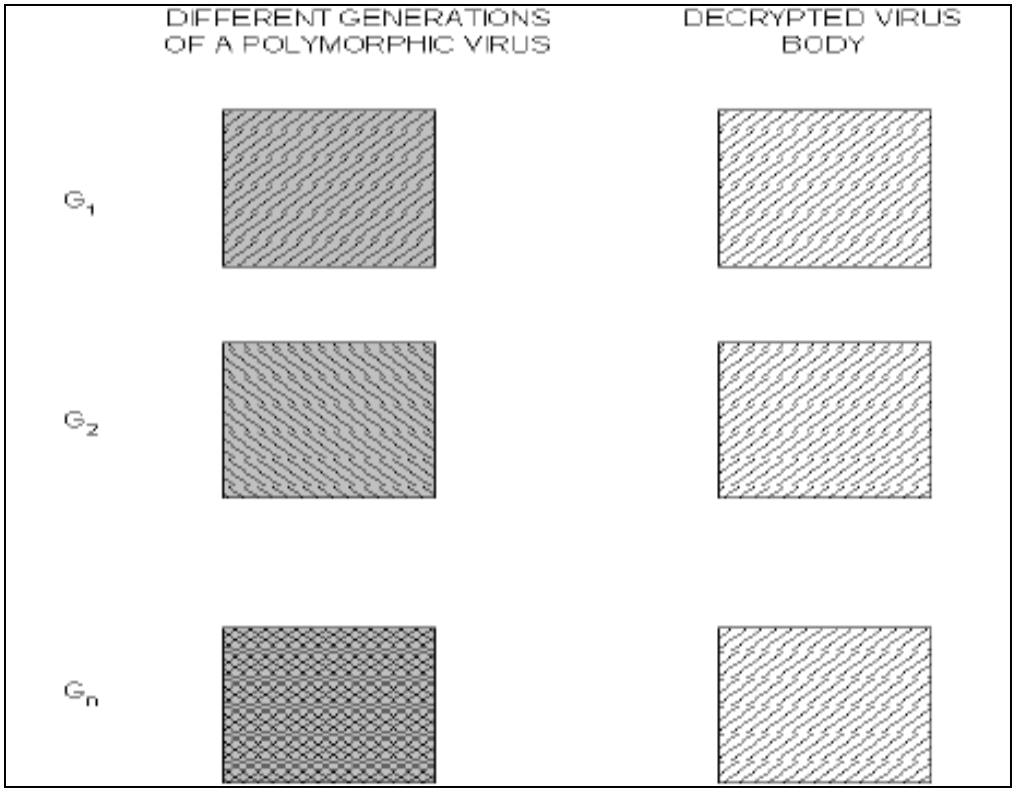


Figure 1 Polymorphic Virus Generations [15]

2.4 Metamorphic Virus

As opposed to a polymorphic virus where virus writers were trying to hide the decrypting module, more advanced techniques were developed enabling the virus writers to change the code of one virus file and create multiple morphed copies but preserving its functionalities [6][29]. These are the type of viruses which have the ability to mutate itself with the code changed but without changing its functionalities. Metamorphic virus can become a serious threat considering the fact that there can be thousands of variants of one virus file with their signature being totally different. Metamorphic viruses uses different kind of code obfuscation techniques like inserting dead code, register swapping, equivalent code instruction insertion, etc to create morphed copies of any base virus file [15]. These obfuscation techniques helps in changing the virus signature to avoid signature based detection. Figure 2 shows the generations of metamorphic virus [15].

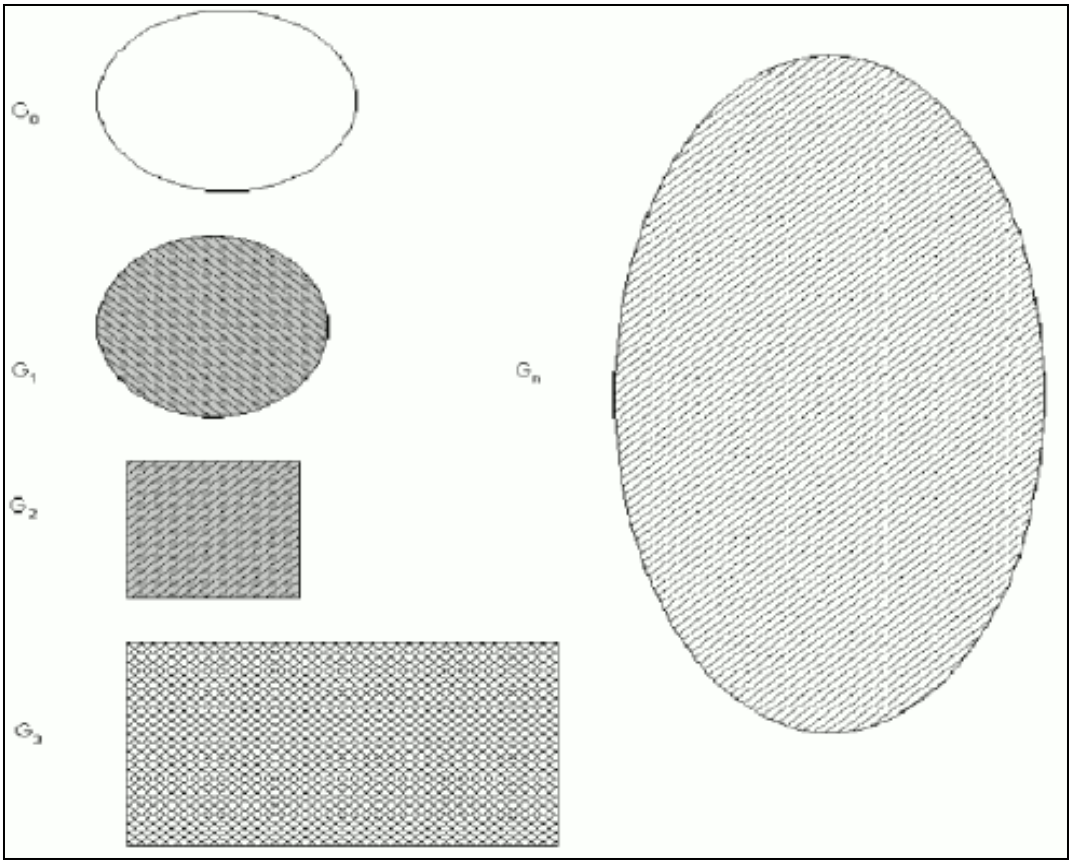


Figure 2 : Metamorphic Virus Generations [15]

3 Anti-Virus Methods

The anti-virus mechanism used today has to fulfill three functionalities so that they can locate any virus. The three parameters are detection, identification and disinfection. Detection part mainly deals in checking whether a given code is malicious in nature or not, based on the virus's behavior or appearance. The second parameter is identification, which identifies a detected virus based on the virus family it belongs to. The third and the last parameter is disinfection or cleaning which is removal of the detected and/or identified virus. This project deals with the detection phase.

Detection methods can be divided into two sub-parts which are dynamic and static detection. This categorization is based on the fact whether the virus file is being executed on the infected machine or not.

3.1 Signature Based Detection

Each virus is represented by a pattern which is a sequence of bytes. Most of the viruses are uniquely characterized by these bytes of patterns. The anti-virus software scans the part of file or the whole file or the boot sector in search of this pre-determined signature of any known virus. Considering the fact that the false alarms in this type of detection will be less, this method is most commonly used in the anti-virus products available in the industry. The downside of this method is that it cannot detect a new virus, since its signature will not be saved in the database.

For example, if the signature of an input file is 83EB 0274 EB0E 740A 81EB 0301 0000, then the scanner will search in the database for this value and will show that it's a W32/Beast virus [2]. Similar to this, a Stoned virus can be detected as shown in Figure 3.

```
seg000:7C40 BE 04 00                    mov     si, 4           ; Try it 4 times
seg000:7C40                                                     ;
seg000:7C43
seg000:7C43                 next:                               ; CODE XREF: sub_7C3A+27↓j
seg000:7C43 B8 01 02                    mov     ax, 201h        ; read one sector
seg000:7C46 0E                          push    cs
seg000:7C47 07                          pop     es
seg000:7C48                             assume es:seg000
seg000:7C48 BB 00 02                    mov     bx, 200h        ; to here
seg000:7C4B 33 C9                       xor     cx, cx
seg000:7C4D 8B D1                       mov     dx, cx
seg000:7C4F 41                          inc     cx
seg000:7C50 9C                          pushf
seg000:7C51 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C56 73 0E                       jnb     short fine
seg000:7C58 33 C0                       xor     ax, ax
seg000:7C5A 9C                          pushf
seg000:7C5B 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C60 4E                          dec     si
seg000:7C61 75 E0                       jnz     short next
seg000:7C63 EB 35                       jmp     short giveup
```

Figure 3 : Stoned Virus Search Pattern [2]

3.2 Heuristics

This method looks for code having “virus-like” behavior (abnormal activity) and can easily find known or even unknown viruses [9]. It is a static analysis, which means that the code being looked for “threat” is not being executed on an infected machine. Heuristics analysis is done in two steps [9] – Data Gathering in which the data is collected using many heuristics and Analysis in which the techniques like data mining, expert systems or neural networks can be used to analyze. Heuristics method may give false alarms but it is effective in finding new viruses.

3.3 Code Emulation

Code emulation is a technique in which a virus is allowed to execute in a simulated environment without actually impacting the host machine. This is a dynamic analysis method as the code of the virus is run to see its behavior. A good emulator comprises of five functionalities [9], which are CPU emulation, Memory emulation, Hardware and Operating System emulation, Emulation controller and Analyzer. Code emulation is a good method to find new viruses including the metamorphic virus. Table 1 lists the various weaknesses and strengths of various detection methods.

Detection technique	Strength	Weakness
Signature based	Efficient	New malware
Anomaly based	New malware	Costly to implement, False Positives, unproven
Emulation based	Encrypted viruses	Costly to implement

Table 1 : Strength and Weakness of Detection Techniques [10]

4 Techniques Used for Code Obfuscation

Code obfuscation techniques can be used by programmers to conceal any logic or purpose by making the code difficult to understand. In the world of viruses, use of these techniques is a boon for any virus writer to make the viruses hidden from the anti-virus software. Metamorphic

engines execute many code obfuscation techniques which allow them to evade signature based detection. These techniques help metamorphic engines to create many morphed copies of a single base virus file.

For assembly programs, code obfuscation basically works over the data section and the control flow [1]. Insertion of jump statements to change the flow of execution is involved in Control Flow obfuscation whereas, dealing with register renaming, subroutine permutation, insertion of dead code constitutes code obfuscation techniques related to the data section. Table 2 shows the code obfuscation techniques used by the well known metamorphic viruses [1].

	EVOL (2000)	ZMIST (2001)	ZPERM (2000)	REGSWAP (2000)	METAPHOR (2001)
Instruction Substitution				✓	
Instruction Permutation	✓	✓			✓
Dead code Insertion	✓	✓			✓
Variable Substitution	✓	✓		✓	✓
Changing the Control Flow		✓	✓		✓

Table 2 : Code Obfuscation Techniques [1]

4.1 Subroutine Permutation

This is a very basic technique used for code obfuscation wherein, the subroutines are reordered /shuffled around using instructions such as jump and label without impacting the subroutine's functionality (Figure 4 shows one such scenario). So, if any program is having n number of subroutines, then all the subroutines can be reordered in n! (n factorial) different ways. W32/Ghost virus [1] had in total 10 subroutines which gave it the capacity to reorder its subroutine in 3,628,800 ways.

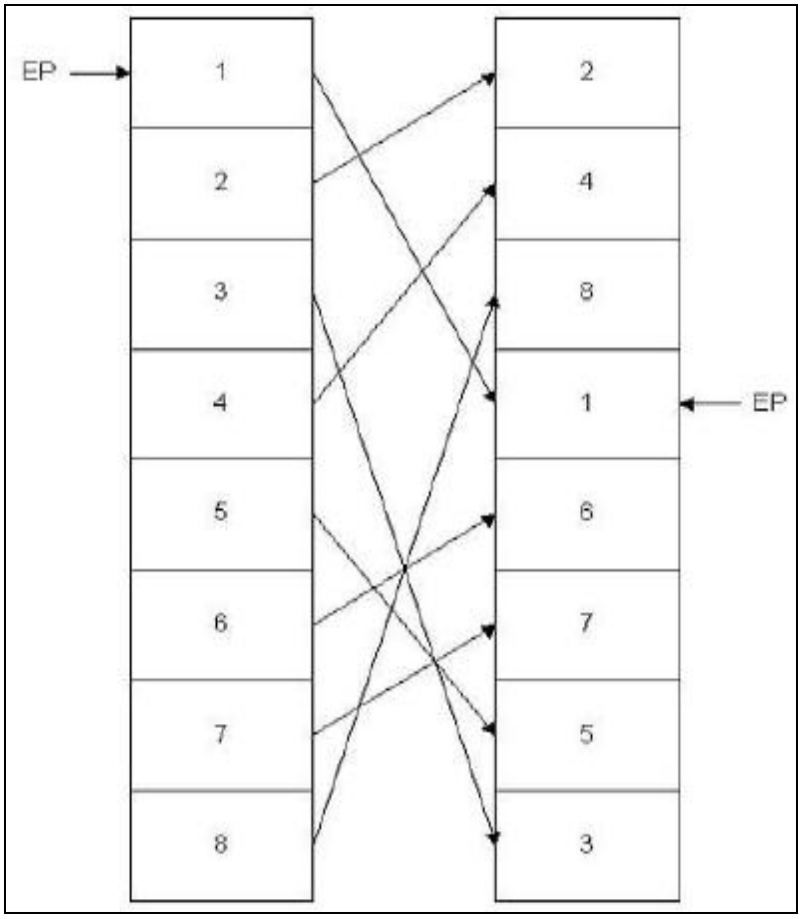


Figure 4 : Subroutine Permutation Example [21]

4.2 Transposition

Modifying the order of execution of instructions in any program is called Transposition. This method can be only applied to a set of instructions which do not have any mutual dependencies. If the output of first instruction is not taken into account by the second instruction, then these two instructions can be swapped as their order of execution will not impact the program's function.

In order to swap two instructions, say instruction one is ADD R1, R2 and instruction two is MOV R3, R4, one needs to make sure that the following rules are satisfied [3]:

1. R3 != R2
2. R3 != R1
3. R4 != R1

For example, instructions MOV A, B and ADD C, D can be swapped based on the above given rule as it would not impact the outcome of the program.

4.3 Register Swapping

This method modifies the current registers used in a particular instruction by swapping it with another equivalent register, which is helpful in evading signature detection as this changes the opcode pattern. W95/RegSwap virus [4] used this technique extensively. An example of two generations of RegSwap appears in Figure 5.

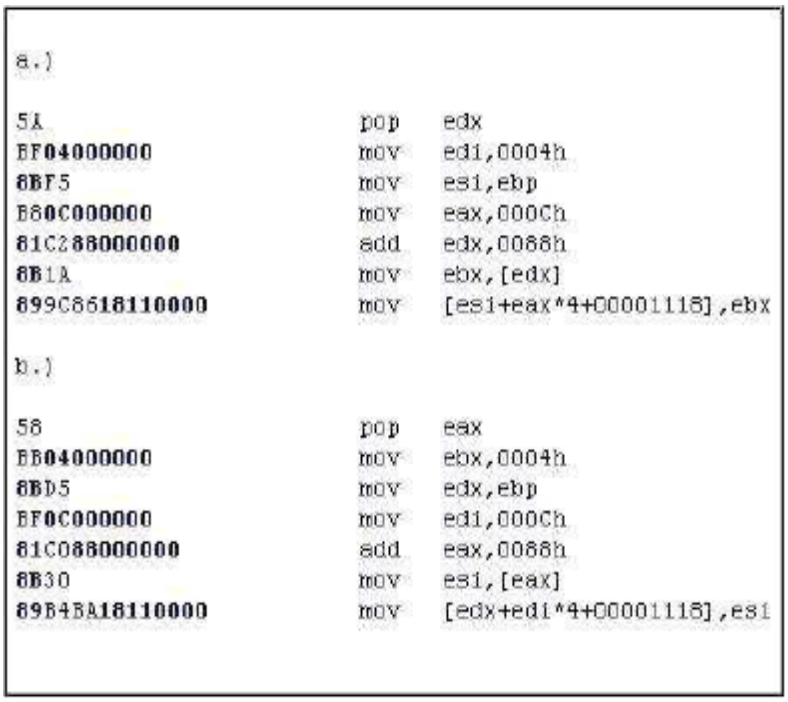


Figure 5 : RegSwap Example [21]

4.4 Instruction Substitution

Metamorphic engines use this technique very commonly for generating highly morphed virus copies. The idea of this method is to replace instruction (even group of instructions) with an equivalent instruction [6]. In assembly language, instruction “add eax, 1” can be replaced with “inc eax”. A few examples used by W32/MetaPhor [1] are shown in Table 3.

Single Instruction	Instruction block
XOR Reg,Reg	MOV Reg,0
MOV Reg,Imm	PUSH Imm POP Reg

OP Reg,Reg2	MOV Mem,Reg
	OP Mem,Reg2
	MOV Reg,Mem

Table 3 : W32 Example of Instruction Replacement [1]

4.5 Insertion of Junk/Dead Code

Most of the metamorphic engines insert junk or dead code in the virus file to vary the signatures of individual virus files morphed from a base virus file. This technique is very effective if used within a certain limit. Inserting dead code beyond a particular point triggers an abnormality which can be easily detected by intrusion detection systems. If an instruction or group of instructions has been inserted, which might be executed but does not alter the functionality of the program, it can be termed as "do nothing code". Instructions like "push eax" followed by "pop eax", if executed, will not affect the program's normal functionality. And if an instruction or block of instructions which has been inserted after a unconditional "jmp" instruction to the next authentic/actual instruction, then this inserted code is called "dead code" as these instructions will never be executed.

The Win95/Zperm is one of the virus which has used this technique in order to create metamorphic copies [1]. Figure 6 illustrates an example of instruction reordering.

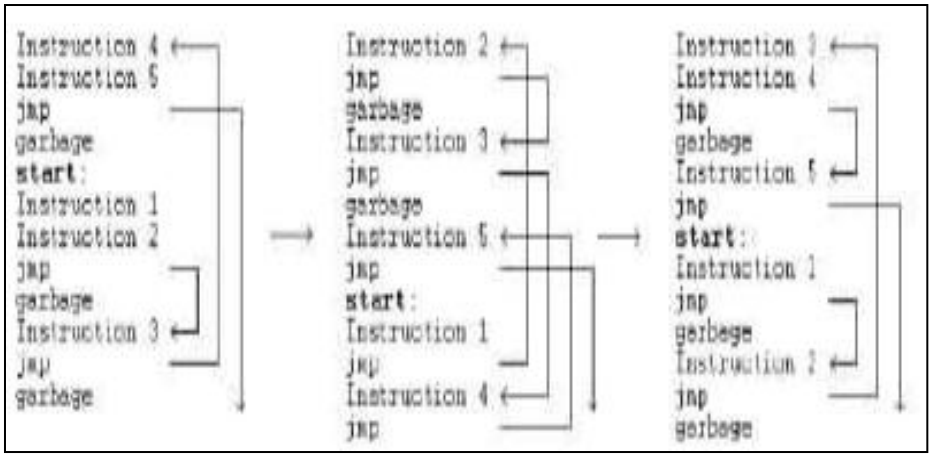


Figure 6 : Win85 Instruction Reordering [1]

5 Hidden Markov Model

5.1 Introduction

A Markov process in probability and statistics is a random phenomenon dependent upon time for which the phenomenon holds a specific property [19]. Hidden Markov Model (HMM) is a tool based on pattern analysis. In this analysis, the system which is being modeled is nothing but a Markov process. A few areas where HMM is used are bioinformatics, protein modeling, gesture recognition and speech recognition applications [10].

First, HMM is fed with an input/training data. HMM then tries to extract a list of unique symbols from the training data. In addition, it also identifies their respective positions in the training data. The data obtained by these extractions and identifications is treated as a model with which HMM will determine whether there is similarity of pattern between the model and a new set of input.

The HMM makes use of the following notations [20]:

T = Length of the observed sequence

N = Number of states in the model

M = number of distinct observation symbols

O = Observation sequence

A = State transition probability matrix

B = Observation probability distribution matrix

π = Initial state distribution matrix

Figure 7 depicts the HMM in generic form [20]. The state at time t is represented by X_t and O_t represents the observation at time t. The dashed line shows the Markov process which is calculated based on State transition probability matrix and the initial state X_0. For every state, we have an Observation sequence representing the Markov process' actual states by the matrices - Observation probability distribution matrix (B) and State transition probability matrix (A).

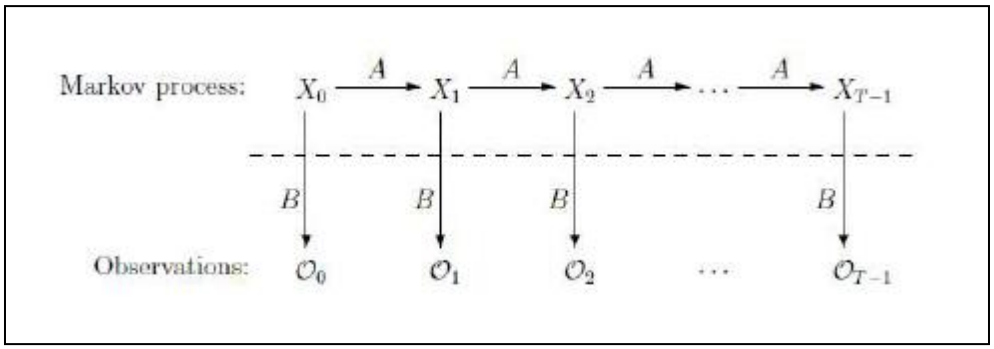


Figure 7 : Generic HMM [20]

5.2 HMM Example

The inner working of HMM is illustrated through an example in [8]. Lets assume about the annual temperature of any given place. It can be either cold (C) or hot (H). One can determine the annual temperature of any year in the future by observing the various size of the trees (size can be Large-L, Medium-M or Small-S). To solve this problem, we have the following information :

a. The probability of a hot year occurring before a cold year is 0.4 or the probability of two consecutive hot years is 0.7. The probability of a cold year occurring before a hot year is 0.3 or the probability of two consecutive cold years is 0.6. Figure 8 shows the probabilities' matrix.

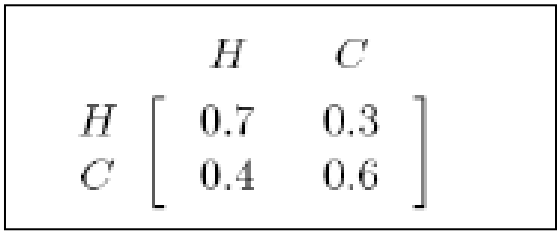


Figure 8 : Probability Based on Temperature Transition [8]

b. This information deals with the temperature and tree size (Large-L, Medium-M or Small-S). The probability of tree being small in a hot year is 0.1 and small in a cold year is 0.7. The probability of tree being medium in a hot year is 0.4 and medium in a cold year is 0.2. And the probability of tree being large in a hot year is 0.5 and large in a cold year is 0.1. The matrix representation is shown in Figure 9.

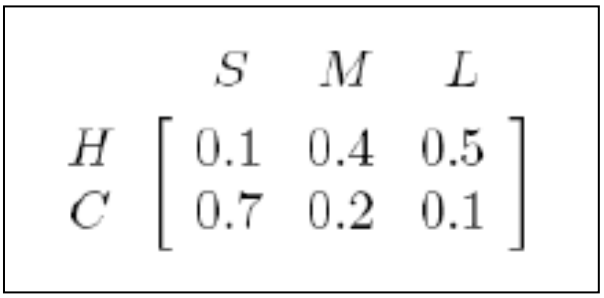


Figure 9 : Probability Based on Tree Size [8]

Now correlating the above information with the HMM notations here, its states are represented by the annual temperatures. The observable symbols are identified as tree sizes. In each state, the probability of observation symbols are represented by tree sizes at each temperature. Figure 10 shows the resulting HMM model [20].

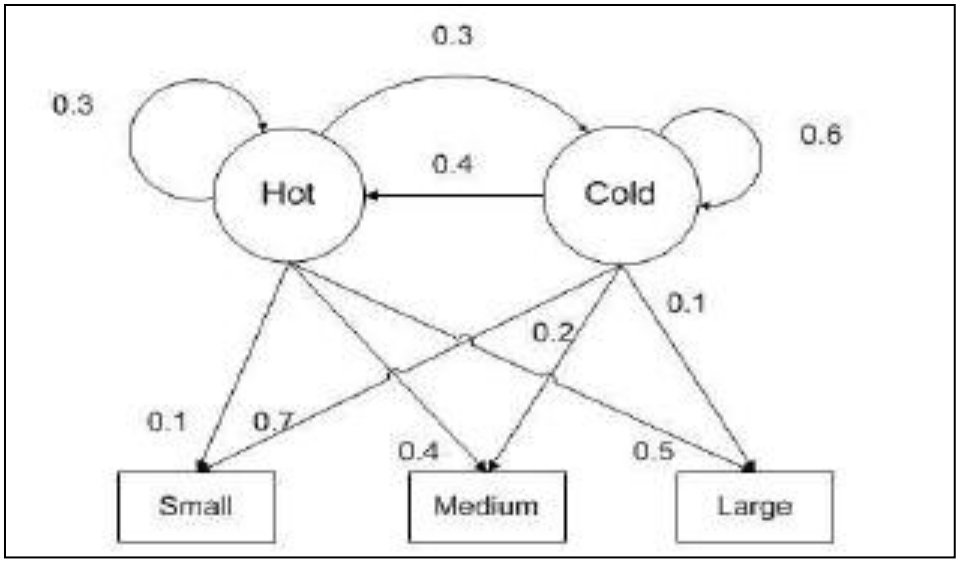


Figure 10 : Resulting HMM Model [20]

For a given observation like (S,M,S,L) having length T=4, to determine the state transition, the HMM would perform the following steps :

1. Determine N^T, which are the state transitions.
2. N ow for each state transition (4 in this example), calculate observations sequence's probability [8].

$$P(HHCC) = \pi_H * b_H(S) * a_{H,H} * b_H(M) * a_{H,C} * b_C(S) * a_{C,C} * b_C(L)$$
$$= (0.6) * (0.1) * (0.7) * (0.4) * (0.3) * (0.7) * (0.6) * (0.1)$$
$$= 0.000212$$

Table 4 shows all the probabilities.

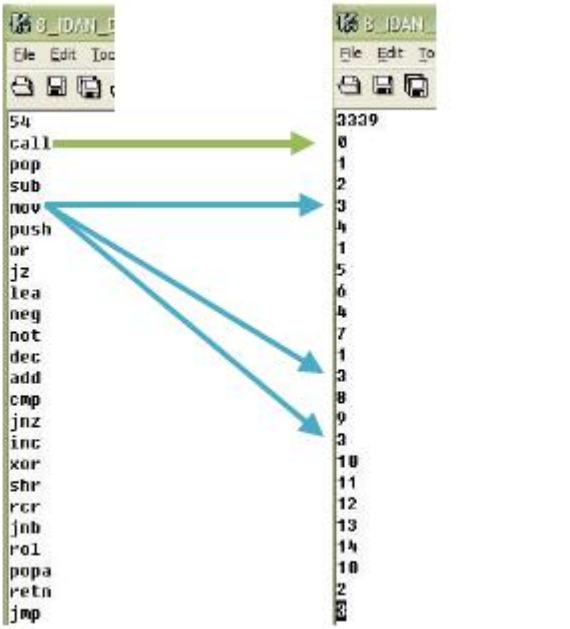
state sequence	probability
HHHH	0.000412
HHHC	0.000035
HHCH	0.000706
HHCC	0.000212
HCHH	0.000050
HCHC	0.000004
HCCH	0.000302
HCCC	0.000091
CHHH	0.001098
CHHC	0.000094
CHCH	0.001882
CHCC	0.000564
CCHH	0.000470
CCHC	0.000040
CCCH	0.002822
CCCC	0.000847
Σ probability	0.009629
max probability	0.002822

Table 4 : Probabilities of all the State Sequences [20]

3. From the Table 4, we can see that the maximum probability is 0.002822. This corresponds to "CCCH" which is the most probable annual temperature sequence.

5.3 Detecting Metamorphic Virus using HMM

To detect a metamorphic virus using HMM, we need training data. This training data is nothing but virus files generated from same virus generator, and converted to .asm file (assembly files) using IDA Pro [22]. HMM needs a unique observation sequence and observation symbols to train a model. Concatenating the opcodes of viruses will give the unique observation sequence and unique assembly opcodes forms the observation symbols. For example, considering the training data in Figure 11, HMM model can be constructed as shown in Figure 12.



(a) Unique Symbols (b) Observation sequence

Figure 11 : Training Data [21]

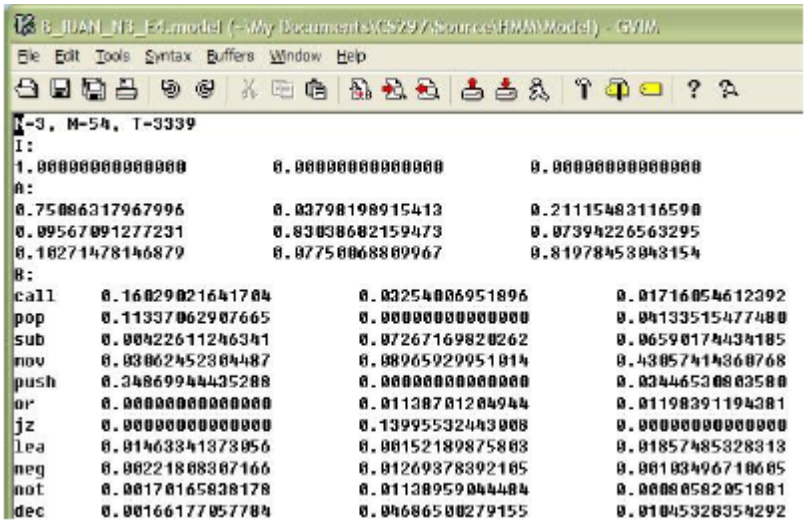


Figure 12 : HMM Model [21]

After constructing the model for a particular virus family, now HMM is used to check whether a particular virus belongs to that family or not. The HMM would produce the result as shown in Figure 13.

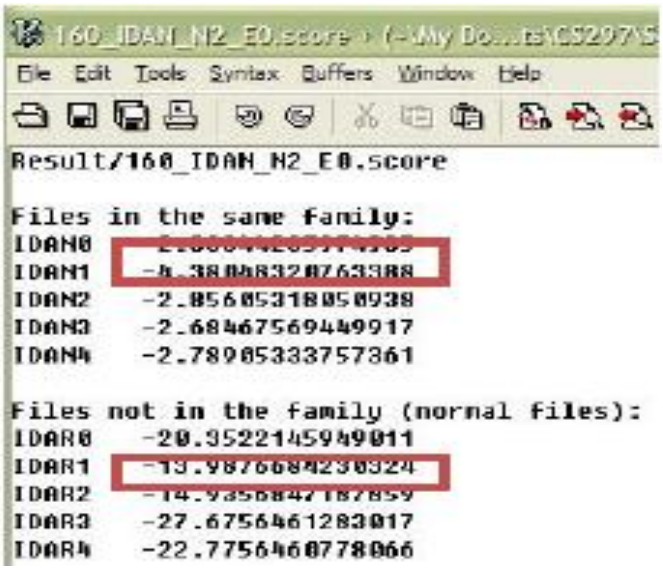


Figure 13 : HMM Output [21]

Considering a threshold value of -4.38, the virus files IDAN0, IDAN1, IDAN2, IDAN3 and IDAN4 belongs to same virus family as their scores are greater than the threshold. The other files have scores less than the threshold, so they cannot be considered as belonging to the same virus family.

5.4 HMM Results Observation

200 viruses generated by Next Generation Virus Creation Kit in [8] were tested with HMM. In total, 25 models were trained and used to differentiate non-virus (normal files) from that of 200 virus files. Out of 25 models, 23 were able to identify normal programs depending upon their scores, which meant NGVCK viruses were easily detected. Figure 14 shows an example of a result which shows the difference of scores between the normal files and the virus files [16].

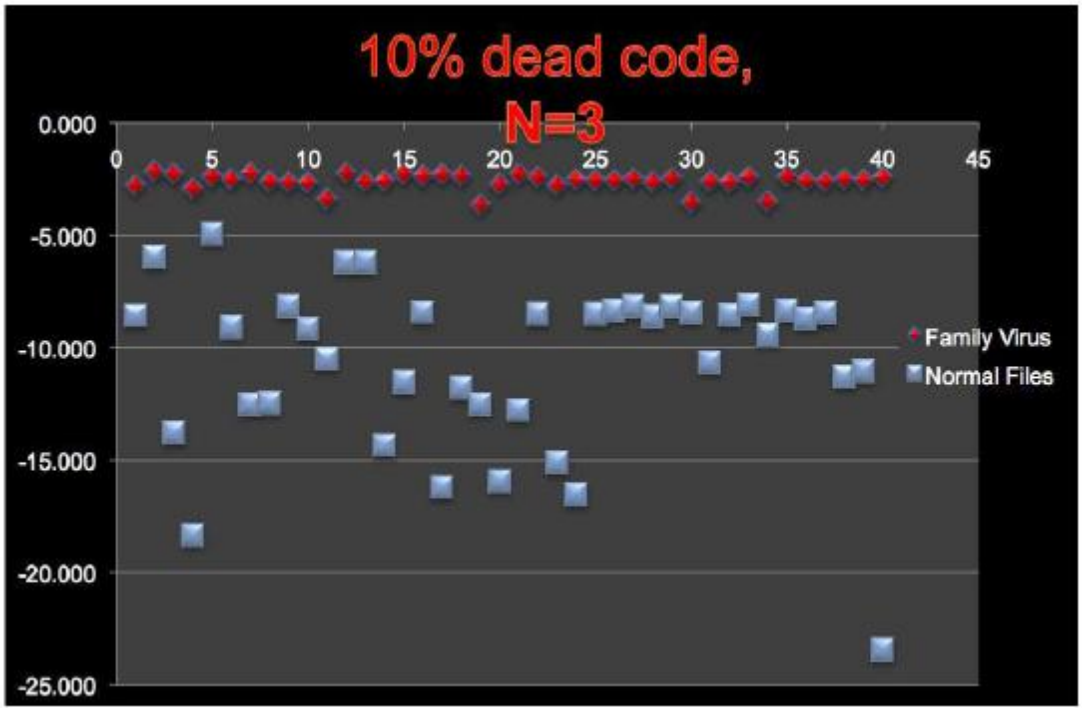


Figure 14 : Sample HMM Result [16]

6 Metamorphic Engine

A metamorphic engine was developed in [21], which used many code obfuscation techniques to produce highly morphed copies of any base virus file. These morphed copies were made by copying codes from normal files which were Cygwin utility files. The metamorphic generator used code obfuscation techniques such as dead code insertion, NOP sequence insertion, equivalent instruction substitution and transposition. Special algorithms were developed to incorporate the above discussed code obfuscation techniques. The morphed virus copies were then tested against the commercial virus scanners and later with the Hidden Markov Model developed in [8].

The experiments conducted with the commercial available anti-virus scanners were very successful. The tests showed that the base virus file was detected by the anti-virus products and thus quarantined. But when the anti-virus scanners were tested against the morphed copies, it failed. The scanners were not able to detect the morphed copies of the same base virus file which

was detectable and thus showing the high level of metamorphism created by the metamorphic generator.

Then the morphed copies were tested against the virus detection tool based on HMM. For one of the test case, 90 virus files were used to make HMM model and then 30 virus files were tested against this generated model. Even with high degree of metamorphism involved, HMM was successful in differentiating between the normal files and the virus files as shown in Figure 15 [21].

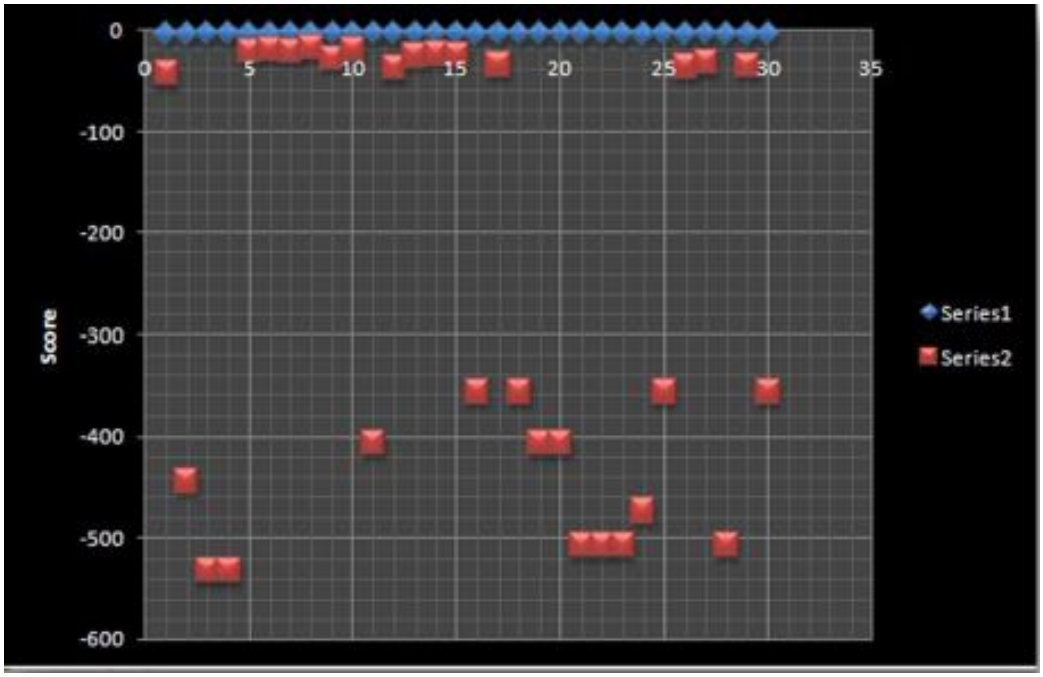


Figure 15 : HMM Results [21]

7 Improved Metamorphic Engine

Even though the metamorphic engine developed in [21] as discussed in previous section was able to develop highly metamorphic virus files, HMM developed in [8] was able to classify the virus files into the same virus family. This drawback of the metamorphic engine developed in [21] was because the engine was randomly applying code obfuscation techniques. Therefore, an improved version of metamorphic engine was developed in [16] to remove this randomness feature. A scoring algorithm known as Dynamic Scoring Algorithm was developed [16], which basically made sure that the code obfuscation techniques are applied only if they make the virus file look like a normal file/program.

7.1 Dynamic Scoring Algorithm

The Dynamic Scoring Algorithm developed in [16] has been mainly divided into three steps :

1. Algorithm Initialization - After passing a virus file and a normal file as parameters, four master lists are created. These lists maintains the information which are individual opcode count and opcode-pair counts of both the normal file and the virus file. Consider the opcodes as shown in Table 5 as present in the normal and virus files.

Virus opcode	Normal file opcode
Mov	Mov
Add	Mov
Mov	Sub
Pop	Pop
Retn	Retn

Table 5 : Opcodes in Virus and Normal Files [16]

Then the four lists generated by the algorithm will have the following contents as shown in Table 6. The algorithm also computes the difference between each opcode-pair and opcode count and adds them.

Virus opcode count list	Normal file opcode count list	difference	Virus opcode-pair count list	Normal file opcode-pair count list	difference
Mov (2) Add (1) Pop (1) Retn(1) Sub (0)	Mov (2) Add (0) Pop(1) Retn(1) Sub (1)	0 1 0 0 1	Mov_add (1) Add_mov(1) Mov_pop(1) Pop_retn(1) Mov_mov(0) Mov_sub(0) Sub_pop(0)	Mov_add (0) Add_mov(0) Mov_pop(0) Pop_retn(1) Mov_mov(1) Mov_sub(1) Sub_pop(1)	1 1 1 0 1 1 1

Table 6 : List Maintained by the Algorithm [16]

2. Score the Changes - Before making any change permanently, a new score is calculated to see whether the new change will bring the virus file closer to the normal file or not. A score less than 0 make the virus file closer to the normal file. An exact score of 0 means there is no change. A score more than 0 mean that the virus file and the normal file is less similar to each other. For example, if "add mov" is changed to "mov add" after transposition, the two opcode sequences passed will be "mov add mov pop" (which is original subsequence) and "mov mov add pop" (which is the new subsequence).

   A change in score is computed as following [16]:

   a. Calculate and save the to-be-affected-counts. Table 7 shows this calculation. The to-be-affected score in this case will be 5.

To-be-affected Virus opcode count list	Normal file opcode count list	Difference before changes	To-be-affected Virus opcode-pair count list	Normal file opcode-pair count list	Difference before changes
Mov (2) Add (1) Pop (1)	Mov (2) Add (0) Pop(1)	0 1 0	Mov_add (1) Add_mov(1) Mov_pop(1) Mov_mov(0)	Mov_add (0) Add_mov(0) Mov_pop(0) Mov_mov(1)	1 1 1 1

Table 7 : Original Subsequence Score [16]

   b. From the master list, subtract the original subsequence's respective counts.

c. Counts of the new subsequence should be added to the master lists. Table 8 shows the steps b and c. Notice that the "Add_Pop" is the new counter in the table.

Subtract original subsequence	Add new subsequence		Subtract original opcode-pair count list	Add new subsequence opcode-pair count
Mov (2-2=0) Add (1-1=0) Pop (1-1=0)	Mov (0+2=2) Add (0+1=1) Pop(0+1=1)		Mov_add (1-1=0) Add_mov(1-1=0) Mov_pop(1-1=0) Mov_mov(0)	Mov_add (0+1=1) Add_mov(0+0=0) Mov_pop(0+0=0) Mov_mov(0+1=1) Add_pop(1)

Table 8 : Subtraction and Addition of New Count [16]

d. Now compute the affected counts. Table 9 shows that the new score will be 3 and the original score was 5, which indicates that if the transposition is done, then the virus file will become closer to the normal file by 2 points.

New Virus opcode count list	Normal file opcode count list	Difference after changes	new Virus opcode sequence count list	Normal file opcode sequence count list	Difference after changes
Mov (2) Add (1) Pop(1)	Mov (2) Add (0) Pop(1)	0 1 0	Mov_add (1) Add_mov(0) Mov_pop(0) Mov_mov(1) Add_pop(1)	Mov_add (0) Add_mov(0) Mov_pop(0) Mov_mov(1) Add_pop(0)	1 0 0 0 1

Table 9 : New Score Calculation [16]

3. Updating the changes - This step deals with making the changes in the master list permanently. The master score now will decrease from 8 to 6 as the score was improved by 2. Table 10 shows the updated master lists .

Virus opcode count list	Normal file opcode count list	difference	Virus opcode-pair count list	Normal file opcode-pair count list	difference
Mov (2)	Mov (2)	0	Mov_add (1)	Mov_add (0)	1
Add (1)	Add (0)	1	Add_mov(0)	Add_mov(0)	0
Pop (1)	Pop(1)	0	Mov_pop(0)	Mov_pop(0)	0
Retn(1)	Retn(1)	0	Pop_retn(1)	Pop_retn(1)	0
Sub (0)	Sub (1)	1	Mov_mov(1)	Mov_mov(1)	0
			Mov_sub(0)	Mov_sub(1)	1
			Sub_pop(0)	Sub_pop(1)	1
			Add_pop(1)	Add_pop(0)	1

Table 10 : Updated Master Lists [16]

7.2 Experimental Results

The improved metamorphic generator was successful in evading HMM detection. It was possible only by generating highly morphed viruses and also maintaining the similarity between the virus file and the normal file, based on the Dynamic Scoring Algorithm. Figure 16 shows one of the test case result, which depicts the failure of HMM to classify correctly between the virus and normal files..

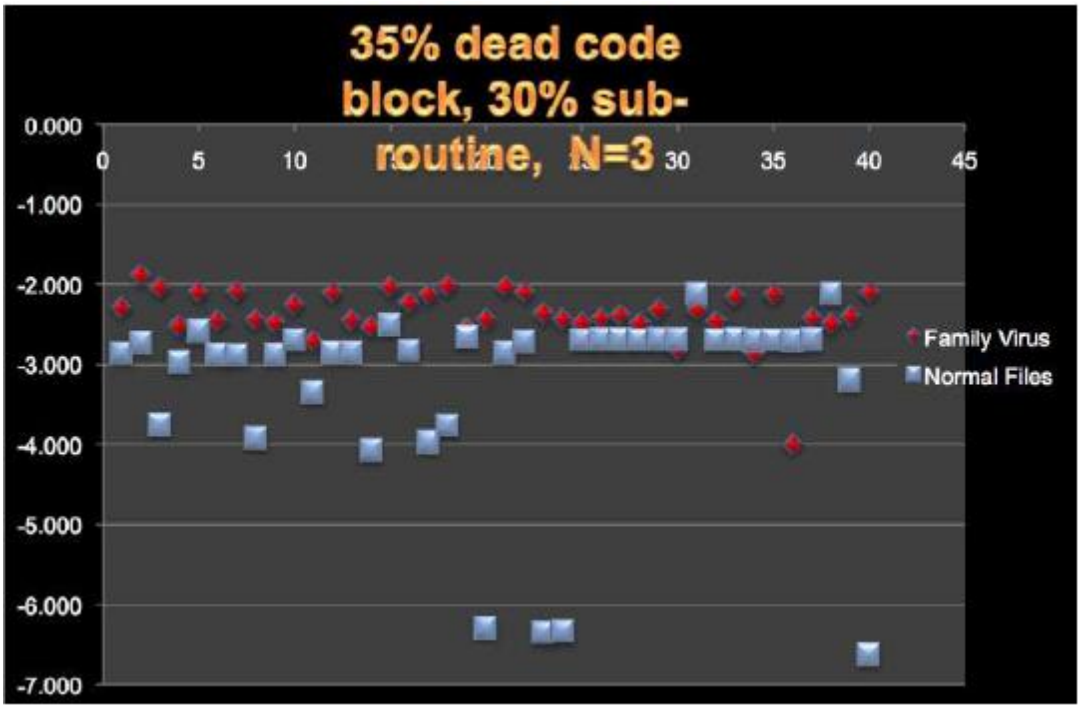


Figure 16 : HMM Results with 30% Subroutines and 35% Dead Code [16]

8 Code Emulator for Metamorphic Code Detection

8.1 Introduction

In general, the code emulator should have the ability to run the virus code being analyzed in an emulated environment. In this way, there is a very high chance that the virus will expose itself about its functionalities. Using virtual flags and registers, the code emulator will run the instruction set of the CPU. Even though code emulation may be a costly solution, but given the task at hand to detect the metamorphic virus, it can be a very effective solution in the long run.

In order to implement a metamorphic virus detector though code emulation, we had to make sure that most of the code obfuscation techniques were taken care of. Code obfuscation techniques such as equivalent code substitution, dead code insertion, junk block insertion and dead subroutine insertion were the primary targets of our code emulator. The aim of our code emulator is to bring the morphed copies of virus file as close (statistically) as possible to the base virus file. By doing this we can make sure that when these un-morphed copies are given as an input to the HMM, it will detect them with ease.

8.2 Goals

The main goals that we wanted to achieve through the implementation of code emulator are:

1. The code emulator should implement as many assembly level language instructions as possible.
2. The code emulator should have the capability to emulate all the important CPU registers.
3. The emulator should be able to filter out or change the instructions/subroutines, which are because of code obfuscation techniques such as: equivalent code substitution, dead code insertion, junk block insertion and dead subroutine insertion.
4. The emulator should also preserve the basic functionality of the virus program.
5. The code emulator should try to bring the un-morphed copies closer to the base virus file “statistically”.

8.3 Experimental Process

In an effort to detect the metamorphic virus or to generate the metamorphic virus, significant background research and work has been done previously. A logical gap was developed in the continued research between the developments of HMM [8] and the metamorphic code generator [16]. So where does exactly our code emulator will fit in? To get the complete picture, Figure 17 shows the entire flow of actions that will be taken to test and validate the results.

For our research, we need two types of data which are the virus files and normal files. For virus files, we used the Next Generation Virus Creation Kit (NGVCK – Version 0.3 stable released on June 2001) to create 200 virus files [25]. These generated virus files were named from "IDAN0" to "IDAN199". For normal files, we chose Cygwin utility files [25] which were randomly chosen. These utility files have pretty much same low level system functionalities as the virus files and thus are ideal candidates for comparison and morphing. These normal files were named from "IDAR0" to "IDAR39".

1. We collected 200 virus files belonging to the same family generated by the NGVCK. These virus files are the base virus files which will be used in our project.
2. IDA Pro [22] is used to dissemble the files into .asm virus files.
3. Out of those 200, 160 virus files are used to make models for the HMM, which will be used later for detection.
4. Remaining 40 virus files and 40 normal files are taken as an input to the metamorphic code generator developed in [16], which are used to create highly morphed copies of all the virus files with different morphing percentage.
5. Once we have a collection of morphed virus files, we feed those files into our code emulator.
6. The output of the code emulator will be un-morphed virus files which will be served as an input to the HMM.
7. The HMM on its behalf will now try to distinguish these virus files based on the model which we had constructed in step 3.
8. The last step will be to analyze the different scores given by the HMM.

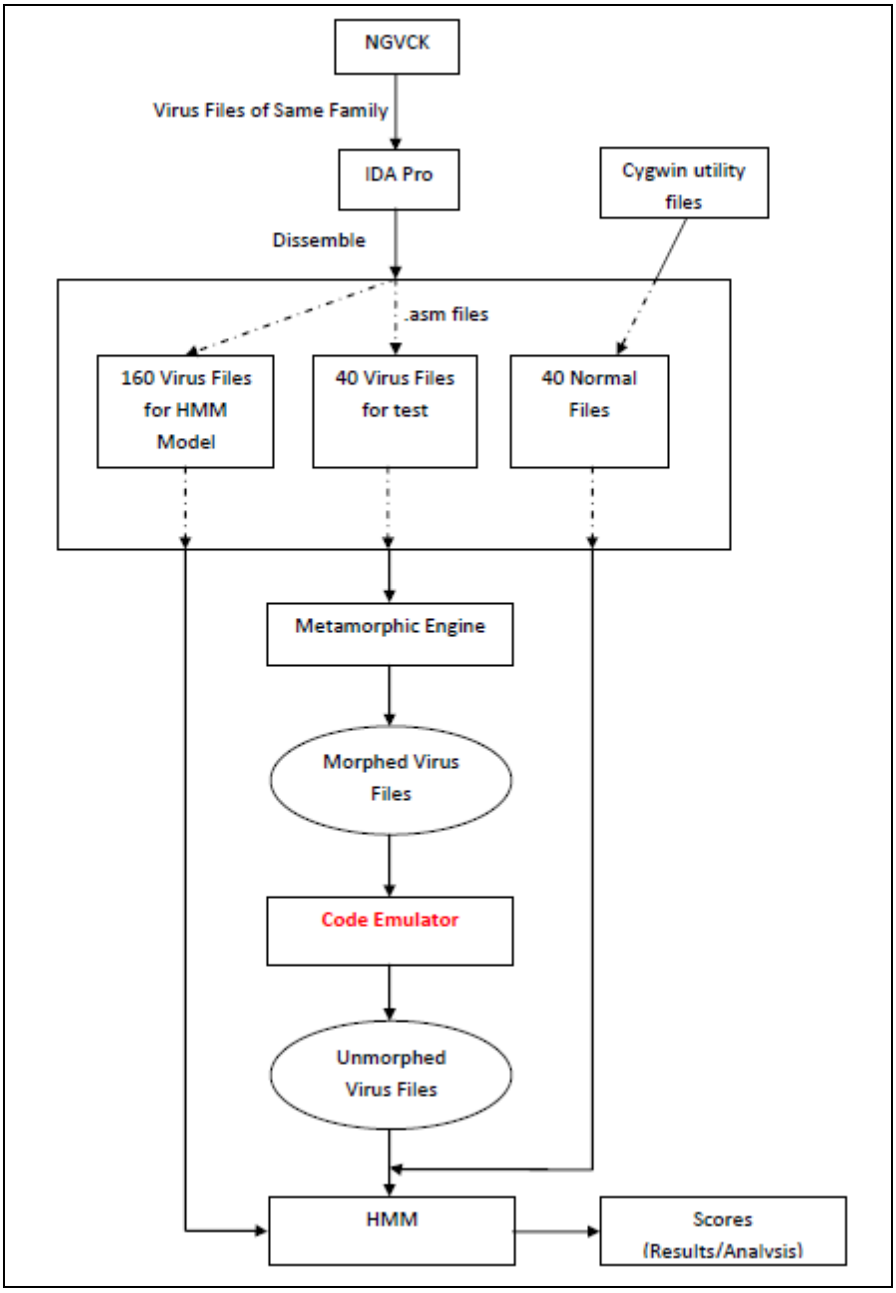


Figure 17 : Code Emulator Process Flow

8.4 Architecture

8.4.1 Introduction

One of the main goal for the development of code emulator was to have a robust architecture, where proper subsystems were identified. We tried to ensure that though implementing the code

emulator is complex, each layer or subsystem is built over relatively clean and simple concepts. Since the code emulator will be having lot of interaction with the files system and database, we chose to implement the emulator in JAVA because we needed a better hold and greater flexibility over the program and the data. The code emulator has been basically divided into seven main components like Execution Path Recorder, Equivalent Instruction Substitution Finder, etc. Figure 18 shows the overall architecture of our code emulator displaying the various components involved.

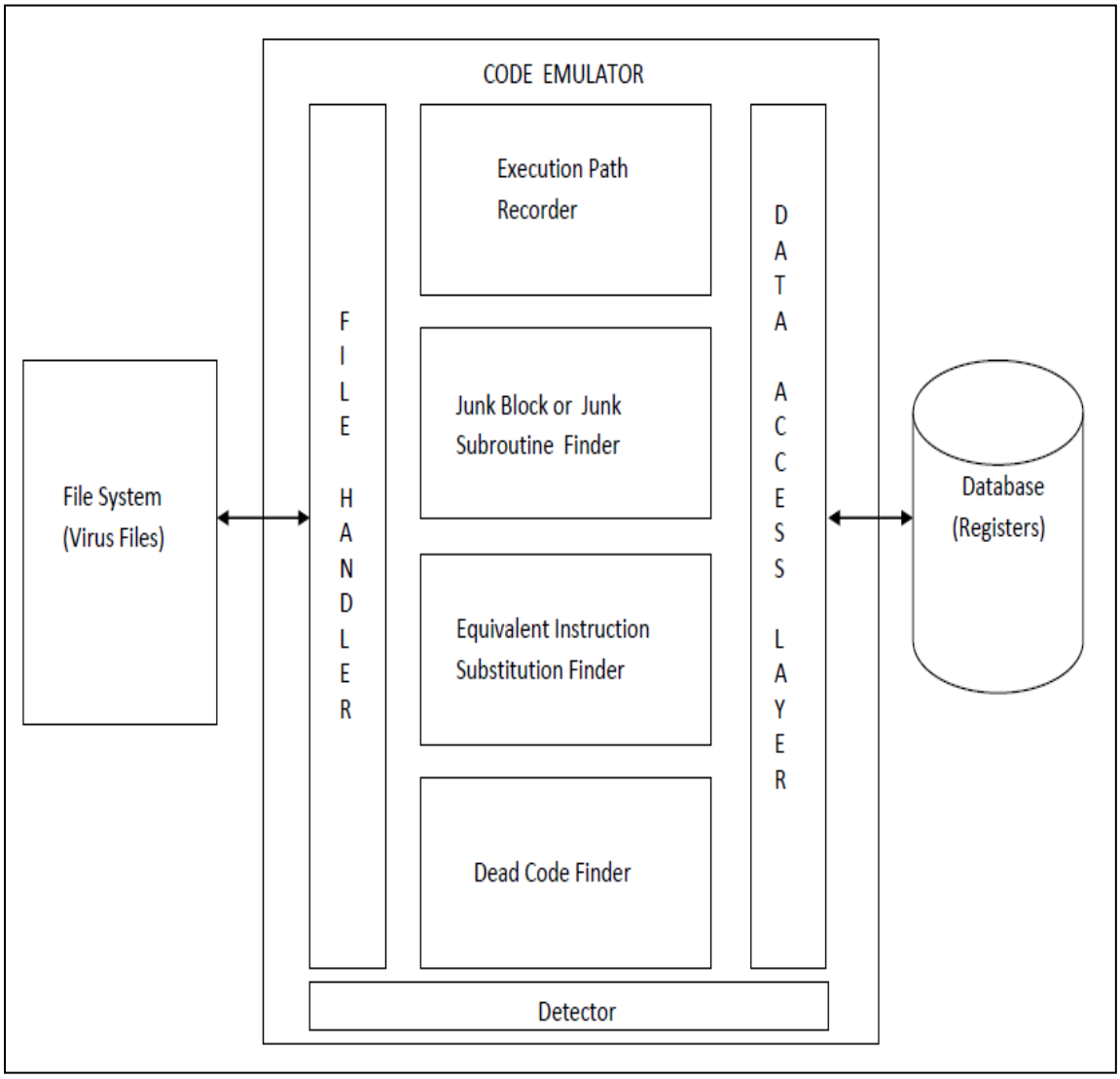


Figure 18 : Code Emulator Architecture

8.4.2 Components

The various components of our code emulator has the following functions:

1. Database Access Layer – This layer has been implemented based on the Singleton Pattern [27] to have more efficiency. The data access layer provides a database connection to all the other requesting components. The singleton pattern makes sure that the only one instance of the class is created, and thus providing global point of access to the database's object.
2. File Handler – This component deals with the writing and reading of various virus files. These operations of accessing file system has been given exclusively to this component.
3. Detector – This module is the main component where the instructions read from the file are passed. This component has been designed as per the Code Emulation Algorithm. The main task of this component is to act as a controller, which decides over which component will be executed next.
4. Dead Code Finder – This module is responsible for finding the dead code as per the Code Emulation Algorithm. This module maintains a list of already known series of dead code instructions through which it finds the equivalent dead codes in the virus file.
5. Equivalent Instruction Substitution Finder – This module is responsible for finding the equivalent instructions based on pattern matching.
6. Junk Block/Junk Subroutine Finder – This component finds all the subroutines which are not called from anywhere and marks them appropriately.
7. Execution Path Recorder – This module is the last one to be called by the Detector module. This is where the emulation takes place and along with, it also marks all the instructions which have been executed.

8.5 Code Emulation: The Algorithm

8.5.1 Introduction

To make sure that our code emulator is following a specific path or process, we came up with an algorithm known as the Code Emulation Algorithm. This algorithm consists of steps specific for

a certain types of code obfuscation techniques. Keeping in mind the various code obfuscation techniques that needs to be handled, the algorithm is designed to make couple of parsing before the actual emulation of registers take place. The sections below explain the steps in detail.

8.5.2 Initializing the Data Structure

As a first logical step, virus file will be read into a particular data structure. So it was important to have a data structure defined for our emulator, which should be easy to handle and maintain. One observation which was very much evident from the disassembled virus files was the way the instructions were laid out. Every location/subroutine individually had a different set of instructions as shown in the Figure 19.

```
loc_40105F:                             ; CODE XREF: CODE:004013E2j
                mov     ebx, 31h
                sub     ebx, 1Dh
                mov     eax, offset dword_401480
                mov     ss:dword_4014CE[ebp], ebx
                add     eax, ebp
                call    sub_401131
                inc     eax
                jz      short loc_401096


sub_401114      proc near               ; CODE XREF: CODE:0040108Cp
                jmp     loc_1019de
                lea     edi,dword_4014D2[ebp]
                push    edi
                push    0
                pop     edx
                add     edx, ss:dword_4014BA[ebp]
                push    edx
                call    ss:dword_401460[ebp]
                jmp     loc_1019fe
sub_401114      endp
```

Figure 19 : Sample Virus File

So, we maintained a separate JAVA class for each location where it was populated with their respective instructions and opcodes saved as array lists. We also maintained separate flag for each location (at class level) and also for each instruction. The respective flags were made true if a particular location/subroutine is called and/or if a particular instruction is executed. This was

the optimal way to keep track of all the instructions being executed. Figure 20 shows a representation of the class with a few methods included.

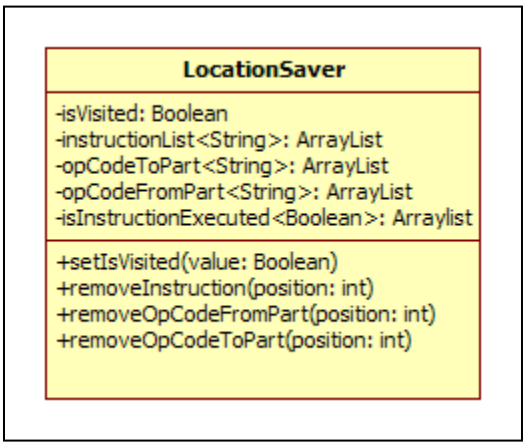


Figure 20 : Class Diagram for Data Structure Maintained

8.5.3 First Pass - Finding Junk Blocks and Junk Subroutines

This is a helper pass which is basically run to ensure that less strain is put over the execution recorder phase (where the emulation of registers take place). In this pass, the emulator will try to find any junk block or junk subroutine which has been embedded into the morphed virus file. This pass does not deal with the emulation of the registers, but it scans all the instructions looking for specific property related to junk block or junk subroutine code.

To improve efficiency, we are maintaining a list known as “CalledSubroutine”. While reading the data from the file into the data structure, this list was being populated with the names of any subroutine which has been called. So, whenever we encountered with the instruction “call”, the subroutine name or the location name was fed into the CalledSubroutine list. This step provided us with the information about the subroutines which “might” never be called for any given scenario. Once the “CalledSubroutine” list is populated, we will delete the subroutines from the data structure whose names are not included in our list.

The second part of this pass is to find the probable junk blocks of code. This part deals with the searching of unconditional “jmp” instructions. If there are any unconditional jump instructions,

then we can mark the remaining instructions in any subroutine/location as “probables” for being never executed. Note that at this stage, we do not delete these instructions from the data structure, but we just mark them so that later in execution recorder stage, we can cross check whether these instructions are executed or not through register emulation. The Figure 21 shows a similar condition.

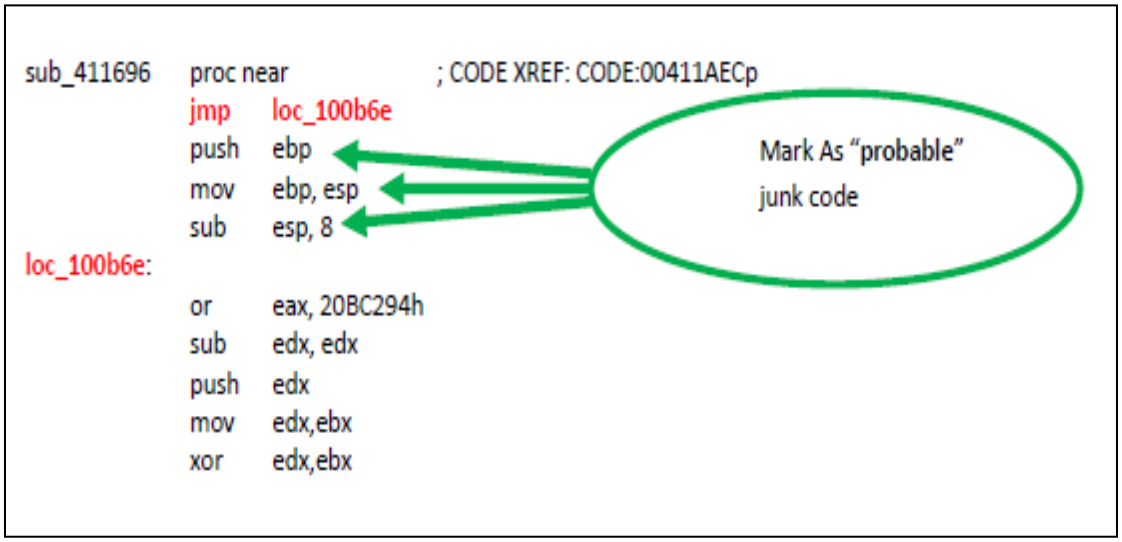


Figure 21 : Sample Junk Block

To sum up, the algorithm to be followed for this round is shown in Figure 22.

Note : From previous phase, we have the list of called subroutines in the list "CalledSubroutine"
Step 1 : For every subroutine in the data structure
Step 2 : If this subroutine exists in the list "CalledSubroutine"
then goto Step 3 else Step 6
Step 3 : For every instruction in the subroutine or location, if
it is unconditional "jmp" then goto step 4.
Step 4 : Mark the REMAINING instructions in that subroutine as
"unvisited" in the data structure
Step 5 : Repeat step 3 and 4 until end of instructions
Step 6 : Delete all the instructions and opcodes of this subroutine.
Step 7 : Repeat steps 2 to 6 until all subroutines/locations are covered

Figure 22: First Pass Algorithm

8.5.4 Second Pass: Find Equivalent Instruction Substitution

One of the steps to make morphed copies of the base virus file was to substitute an equivalent instruction [16]. The equivalent instruction substitution does make a lot of difference for scanners, which are based on signature detection and HMM, too. Since substitution of an equivalent instruction will not make any difference to the existing functionality, catching it through the emulation process solely will be very tough as we cannot impose any general logic behind it. To overcome this problem, we used the list of instructions and their equivalent instructions listed in [21] and used them in our implementation (See Appendix A for a complete list of instructions and their equivalent instructions). There are close to 50 instructions and their substitute instructions in this list.

In order to implement this scenario, we did pattern matching of various instructions and their operands to reverse it back to the original instruction. For example, consider the following instruction substitution for instruction "dec R" in the Table 11.

Instruction	Substitution
dec R	1. neg R not R
dec mem	1. neg mem not mem

Table 11 : Equivalent Substitution Example

Now in this pass, as the emulator goes through all the instructions, it will try to match all the new patterns with the patterns of the equivalent instructions already saved in the emulator. Referring from the Table 11, it can be seen that a simple instruction of "dec R" or "dec mem" can be replaced with "neg R" followed by "not R" or "neg mem" followed by "not mem". So the job of emulator at this stage will be to find the matching patterns and replacing those instructions with their original counterparts. In this case, wherever the emulator finds "neg R not R" as the two consecutive instructions for a particular location or subroutine, these instructions will be replaced with "dec R".

8.5.5 Finding Dead Code and Recording Execution Path

This is the last and the most important step in the execution of a virus file. Till this step, the virus file which has been put into the data structure, has been cleaned up of "most" of the instructions which was result of various code obfuscation techniques. But there will be still many more instructions left to be found, whether they are actually impacting over the functionality of the virus program or not.

As a first part of this step, while the code emulator goes instruction by instruction, it tries to find out the dead code (instruction which executes but will not impact over the functionality). We took the list of possible dead codes [21] (See Appendix B for a complete list) and the code emulator will keep looking for them during the execution of the virus file. If any of the sequence of instructions were found in the file, the code emulator will simply block them from being executed and mark them as unvisited.

The next phase of this step was the actual emulation of a virus file. In order to run this step, we emulated the various registers present in the 8086 architecture in our database. All the registers

were created as a new column inside a table of our database. For example, if we have 2 registers EAX and EBX to be emulated, then we will have 2 columns named EAX and EBX in our database's table. So whenever the emulator will encounter instructions handling these two registers, database table will be updated appropriately. Our emulator is supporting most of the registers (See Section 8.6 for list of Supported Registers).

Other emulation that our code emulator is dealing with is the emulation of different kind of instructions. We implemented the functionality of many instructions (See Section 8.7 for list of Supported Instructions). For example, if the emulator encounters "mov eax,ebx", then the emulator will use database query to remove the value of ebx (from the database table's column ebx) and then insert it into the column eax. Each instruction was implemented separately based on their functionality in our emulator, so that they perform the same operation with our emulated registers as it would have done with the real CPU registers.

To get a complete picture of emulation, consider an example where the emulator encounters two instructions as "mov ebp, esp" and then "dec ebx". We have a separate implementation of these instructions in our emulator. So in this case, the emulator will pick up the value from column "esp" in the database and insert it into the column "ebp". Then the code emulator will decrement the value present in the column "ebx" . Figure 23 depicts this scenario.

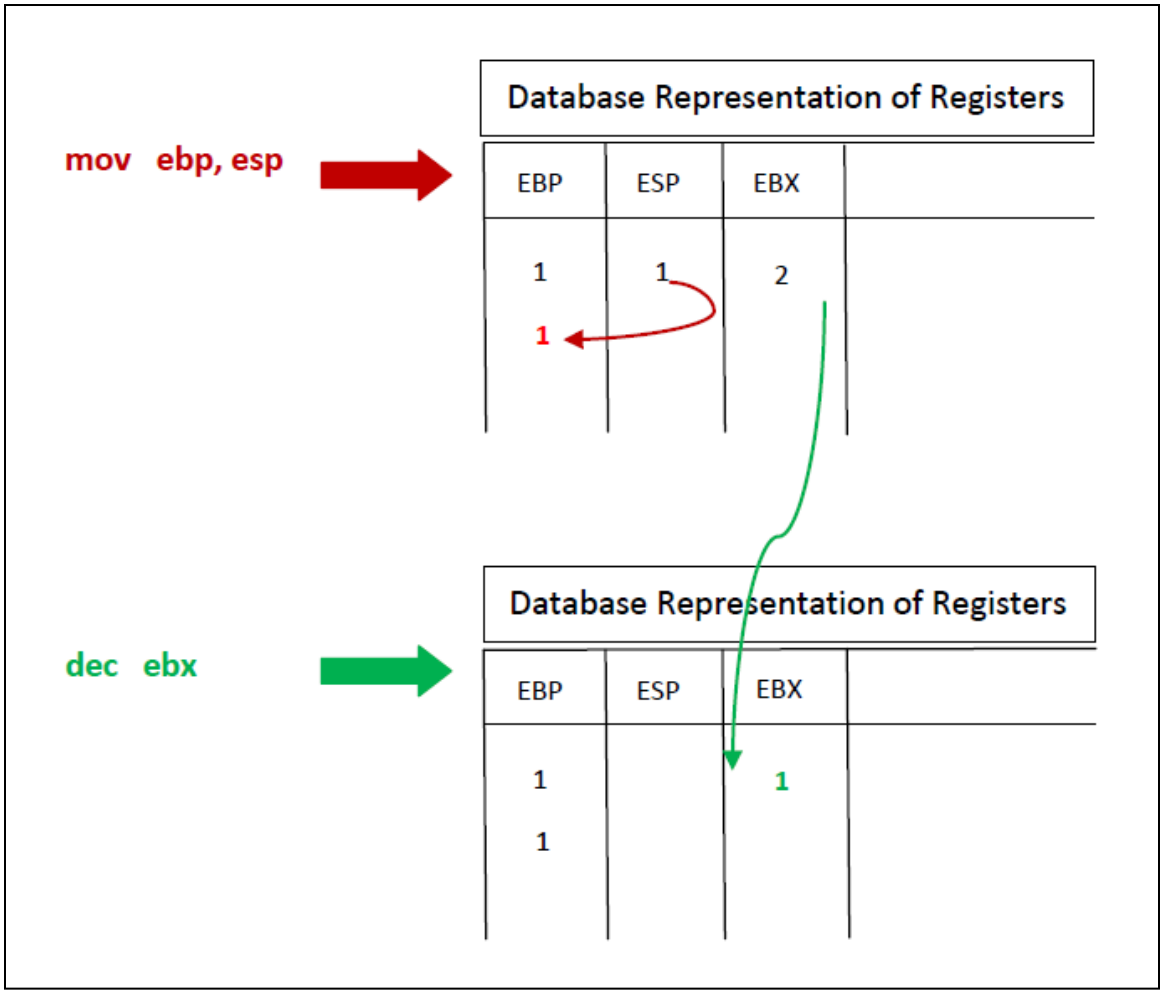


Figure 23 : Register Emulation through Database

The code emulator while executing these instructions, also keeps updating in the data structure whether or not any particular instruction has been visited/executed. In this way, when the emulation stops, the code emulator would have marked all the possible instructions which were executed for a particular path. Basically the code emulator tries to follow a particular path and record all subroutines/locations/instructions that have been executed.

At the end of this step, the code emulator will produce a .asm file which will have the instructions that were marked as visited/executed in our data structure.

8.6 List of Registers Supported

For performing an effective code emulation, the code emulator will attempt to capture as many registers as possible so that most of the .asm file of Intel 8086 could be executed. Registers are fast memory, almost always connected to circuitry that allows various arithmetic, logical, control, and other manipulations, as well as possibly setting internal flags [24]. Implementation of various registers will be based over the functionalities of individual register set. Below are the register sets which have been identified for implementation:

1. Accumulators : All the operations such as rotate, logical, arithmetic shift or similar operations are done by the registers known as Accumulators. In 8086, AX is the one word accumulator of size 16 bits. Variation is that higher order byte of AX is called AH, whereas lower order byte is called AL.
2. General Purpose Registers : General Purpose Registers in 8086 are BP, BX, AX, CX, SP and DI. To cover these, we needed to have both the lower order and the higher order bytes variations. Higher order for general purpose registers are called BH, AH, DH, and CH and the lower order bytes are named as BL, AL, DL, and CL.
3. Index Registers : In 8086, index registers are nothing but use of general purpose registers. So we have used the general purpose registers as index registers itself. A more complicated version can be made by combining the index register and the address register.
4. Base Registers : These are used to segment memory. In 8086, there are six of them : GS- data segment, SS- stack segment, , ES- extra segment, FS- data segment register, CS- code segment and DS- data segment.
5. Program Counter : We did not emulate the program counter as we had other mechanism to follow the execution path. Program Counter basically stores the next executable instruction’s address.
6. Stack Pointer : In 8086, SP- stack pointer combined with SS - stack segment pointer is used to create address of the stack.

8.7 Instructions Supported

For the implementation of our code emulator, target was to include most of the 8086/8088 instructions sets. Refer to Appendix C for a complete list of instructions [23] supported by the 8086/8088 architecture. There are close to 100 individual instructions with many instructions having different variations (which meant different approach for each variation). So to avoid unnecessary implementation of less used instructions, we wrote a utility java program, which took input as 15 of the virus files and created a list of most frequently used instructions in these virus files. So we implemented close to 30 instructions in total based on the figures thus collected. Figure 24 shows the list of those instructions and their average frequency of occurrence in those 15 virus files.

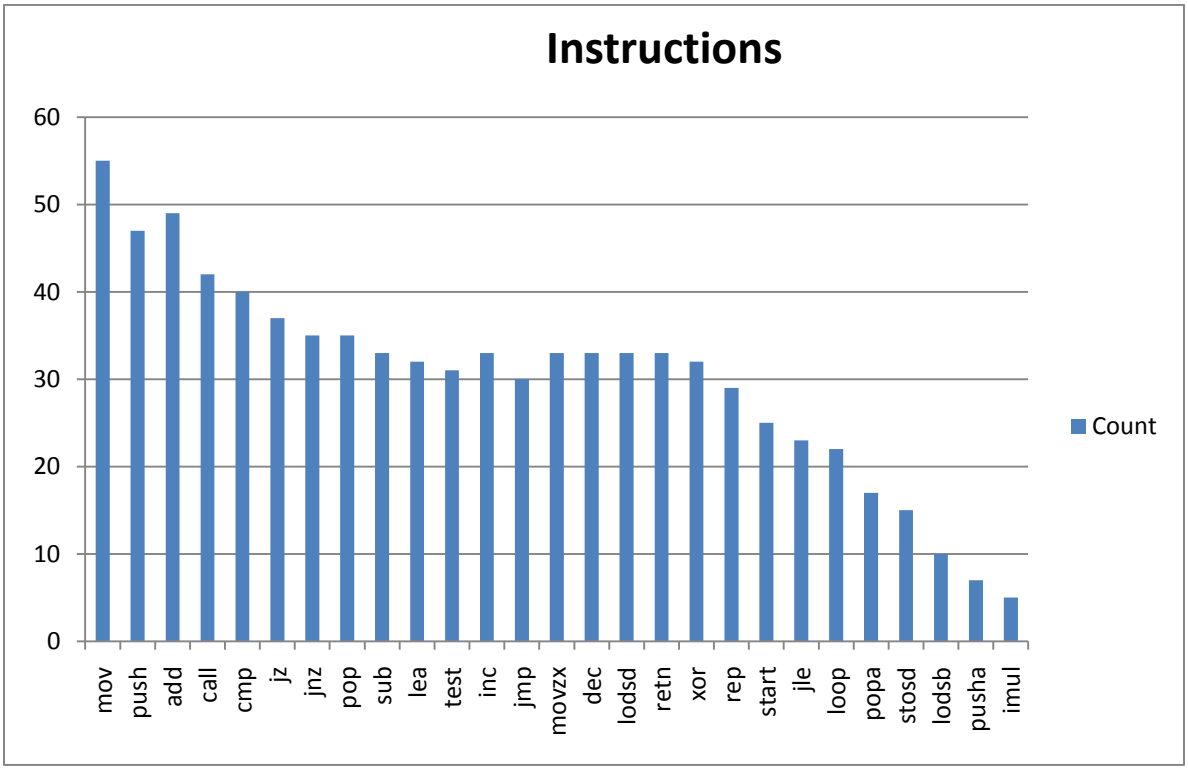


Figure 24 : Opcode Frequency of 15 Virus Files

9 Experiments and Analysis

9.1 HMM Test for Base Virus Files

To make sure that HMM is detecting our 40 base virus files; we ran a test for HMM detection. If the scores obtained by HMM for the virus files are lower than the scores obtained for the normal files, then the HMM will be able to distinguish between them. This is because, HMM maintains a threshold value. Score of any file lower than the threshold is considered as a normal file and score of file higher than the threshold is considered as a virus file.

Figure 25 shows the HMM result for our 40 base virus files. The HMM was successfully able to differentiate between the normal files from the virus files.

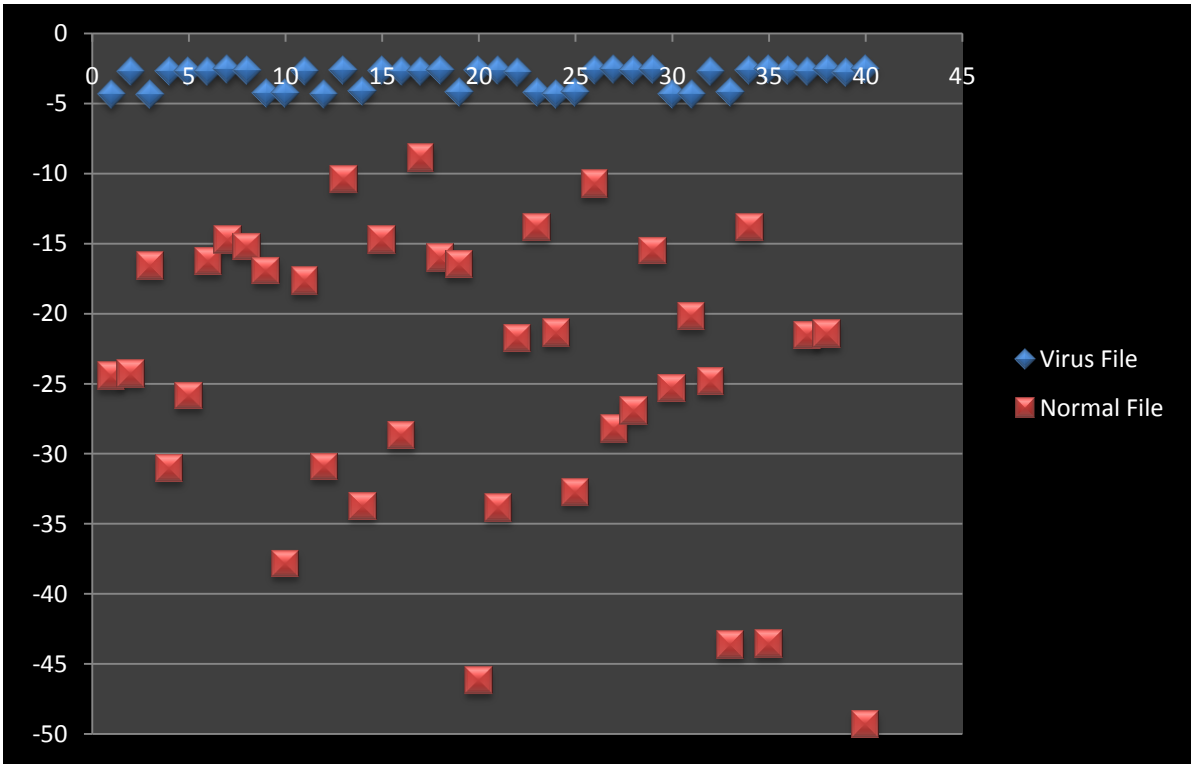


Figure 25 : HMM Results for 40 Base Virus Files

It can be observed from the above figure that the minimum score of virus family is -4.38473 and the maximum score of normal file is -8.90711, so the HMM was able to make a clear distinction of both type of files (Refer to Appendix F for the complete list of HMM scores).

9.2 HMM Test without Code Emulation

To conduct this step in the experiment process, we took the 40 base virus files and morphed them using Metamorphic Virus Generator Engine [16]. The engine will take one normal file and one virus file as input and apply various code obfuscation techniques in an effort to make the base virus file closer to the normal file. For our experiment, we have 40 base virus files and 40 normal files. So, we took the 1st virus file with 1st normal file, 2nd virus file with 2nd normal file and so on. At this stage we expect that there will be many morphed virus files which would not be detected by the HMM. We morphed the base virus files with different settings (different percentage of morphing).

9.2.1 HMM Test with 15% Morphing

We started our experiments by morphing the base virus files by 15%, which was having 5% subroutine copied into from the normal file. Then we ran the HMM test again for these set of morphed virus files. The HMM was not able to detect all the morphed virus files as it did before the morphing had happened. Figure 26 shows the result of our HMM test. With the maximum score of normal files being -8.90711, we found that there were 20 virus files whose score was less than the maximum score of the normal file (Refer to Appendix G for a complete list of HMM scores).

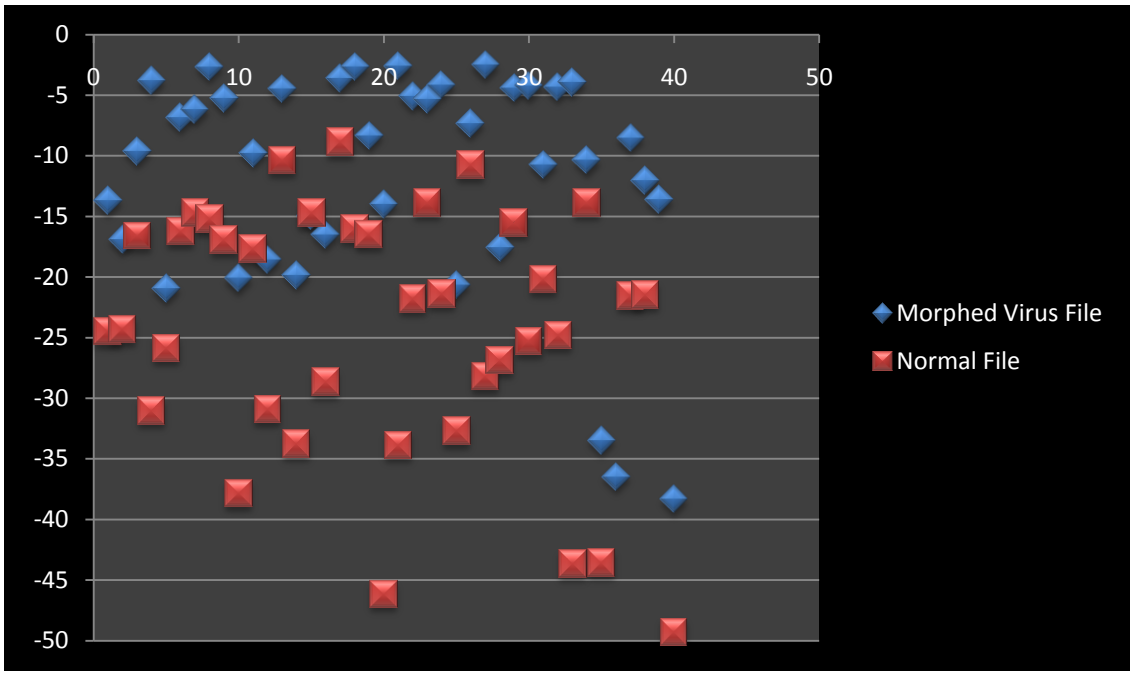


Figure 26: HMM Test with 15% Morphing

9.2.2 HMM Test with 25% Morphing

For this round, we started our experiments by morphing the base virus files by 25%, which was having 15% subroutine copied into from the normal file. Then we ran the HMM test again for these set of morphed virus files. The HMM was not able to detect the entire morphed viruses. Figure 27 shows the result of our HMM test. With the maximum score of normal files being -8.90711, we found that there were 20 virus files whose score was less than the maximum score of the normal file (Refer to Appendix H for a complete list of HMM scores).

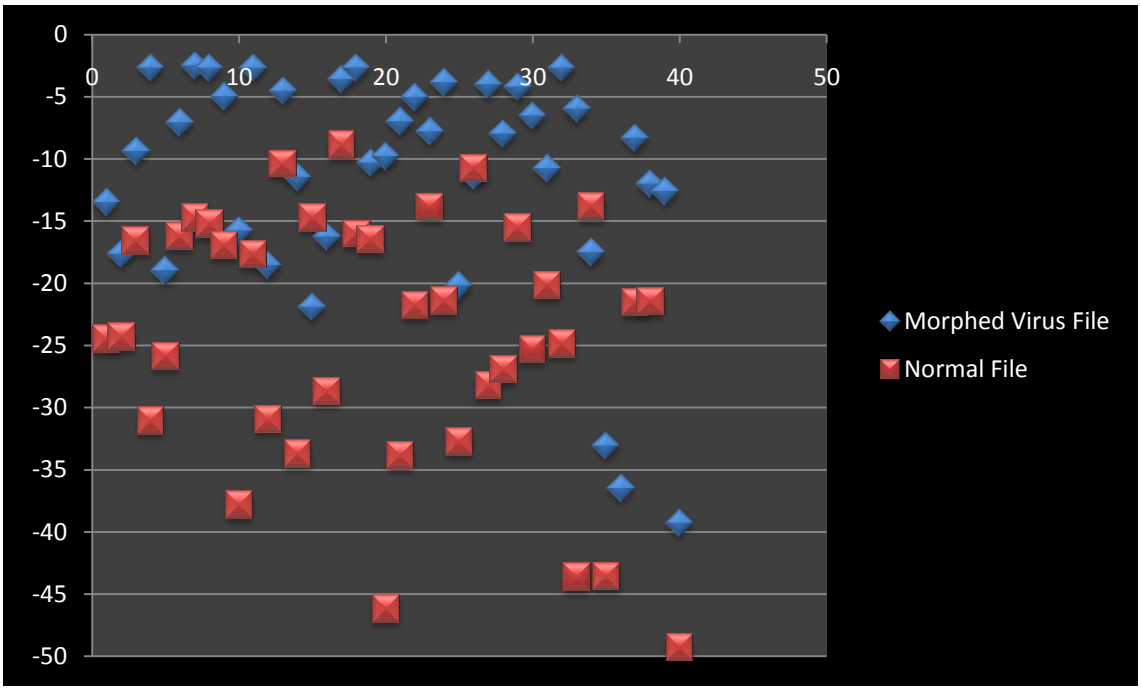


Figure 27 : HMM Test with 25% Morphing

9.2.3 HMM Test with 35% Morphing

We started our experiments by morphing the base virus files by 35%, which was having 25% subroutine copied into from the normal file. Then we ran the HMM test again for these set of morphed virus files. The HMM was not able to detect all the virus files as it did before the morphing had happened. Figure 28 shows the result of our HMM test. With the maximum score of Normal Files being -8.90711, we found that there were 16 virus files whose score was less than the maximum score of the normal file (Refer to Appendix I for a complete list of HMM scores).

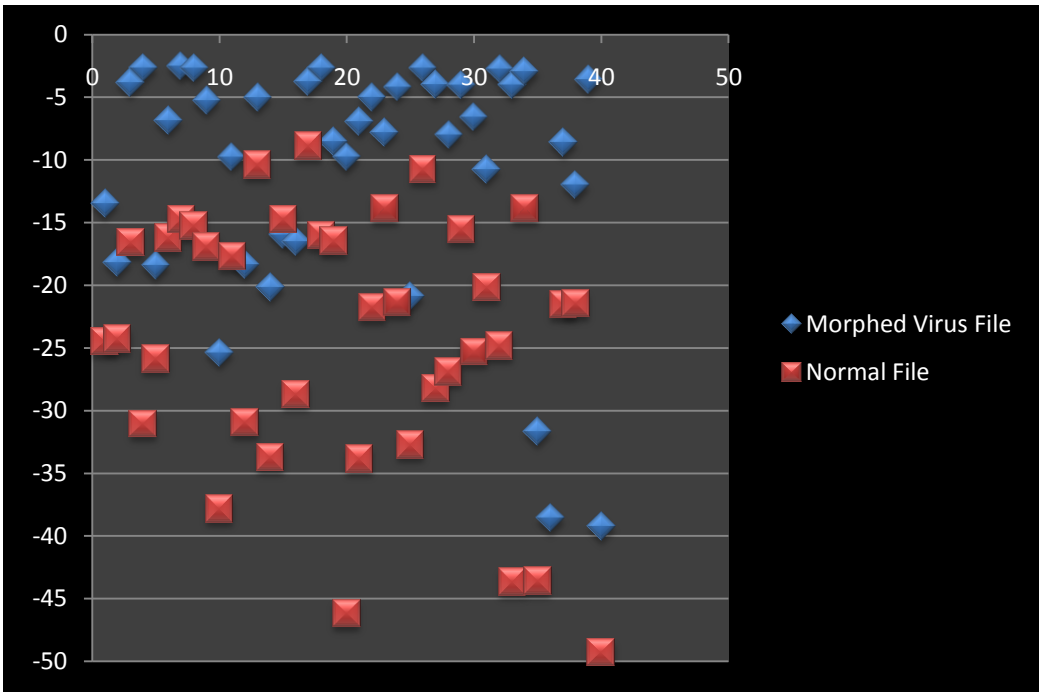


Figure 28 : HMM Test with 35% Morphing

9.3 HMM Tests with Code Emulation

From the tests conducted in the previous section, we got sure that HMM was not able to detect all the virus files after morphing. Now we took the 40 morphed virus files generated from the above tests (for various morphing percentage) to run with our Code Emulator. The code emulator will try to remove as much as code obfuscation techniques applied to the virus files and create "Un-Morphed" virus copies. We will test these un-morphed virus copies with the HMM tool. The expectation was that as the HMM was able to detect base virus files, it will also detect the corresponding un-morphed virus files.

9.3.1 HMM Test with 15% Morphing

We took the 40 morphed virus files (having 15% morphing and 5% subroutine copying) and run them in our code emulator, whose output was un-morphed virus files. We then tested the 40 un-morphed virus files to see whether HMM can now detect these or not. Figure 29 shows that the HMM was able to distinguish between the un-morphed virus files and the normal files. The

minimum score of un-morphed virus files is -6.39854 and the maximum score for the normal file was -8.90711. Thus from the HMM scores generated, we can show that the code emulator was successful in detecting the code obfuscation techniques (Refer to Appendix J for a complete list of HMM scores).

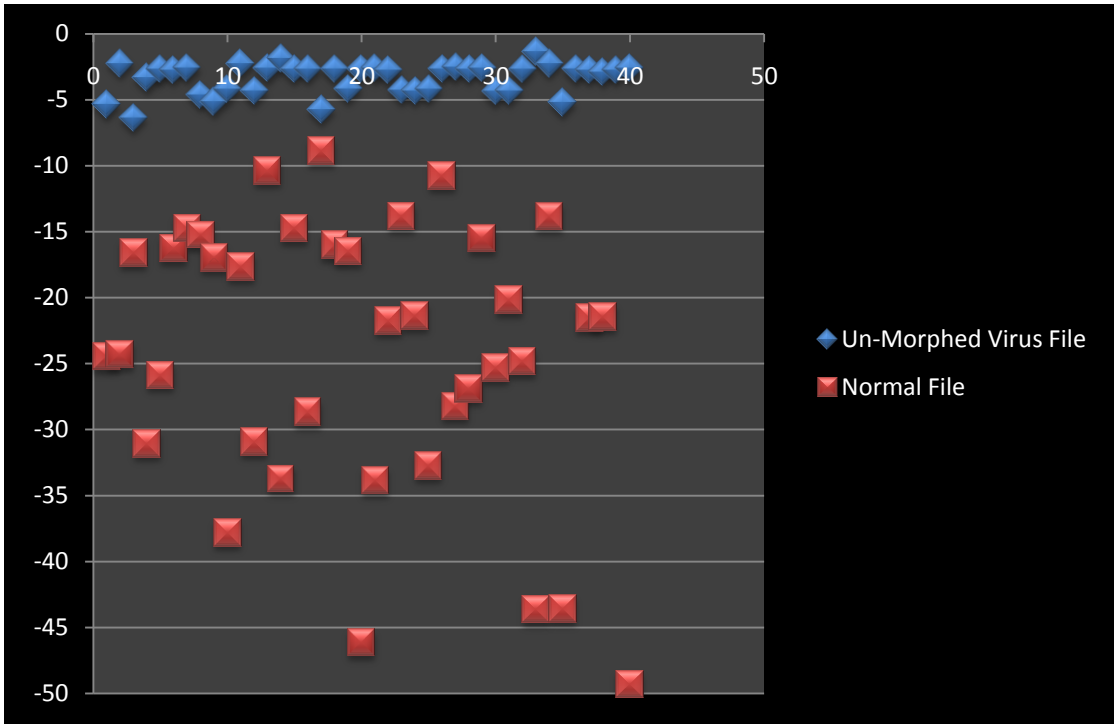


Figure 29 : HMM Test with 15% Morphing

9.3.2 HMM Test with 25% Morphing

In this step, we took the 40 morphed virus files (having 25% morphing and 15% subroutine copying) and run them in our code emulator, whose output was un-morphed virus files. We now tested these 40 un-morphed virus files to see whether HMM can now detect these or not. Figure 30 shows that the HMM was able to distinguish between the un-morphed virus files and the normal files. The minimum score of un-morphed virus files is -6.26291 and the maximum score for the normal file was -8.90711 (Refer to Appendix K for a complete list of HMM scores).

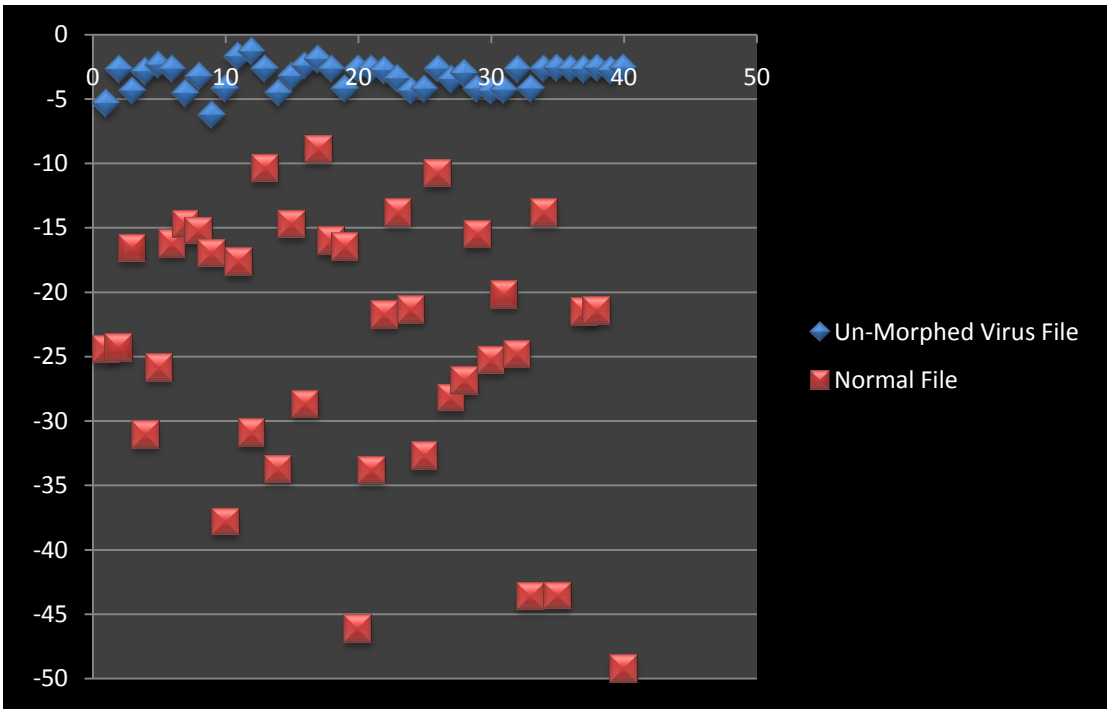


Figure 30 : HMM Test with 25% Morphing

9.3.3 HMM Test with 35% Morphing

The Figure 31 shows that the HMM was able to distinguish between the un-morphed virus files and the normal files. This test was aimed to see whether the code emulator can remove code obfuscation techniques from virus files, which have been morphed as high as 35% with 25% subroutine copying. The minimum score of un-morphed virus files is -6.73408 and the maximum score for the normal file was -8.90711 (Refer to Appendix L for a complete list of HMM scores). Thus from the HMM scores generated, we can show that the code emulator was successful in detecting the code obfuscation techniques.

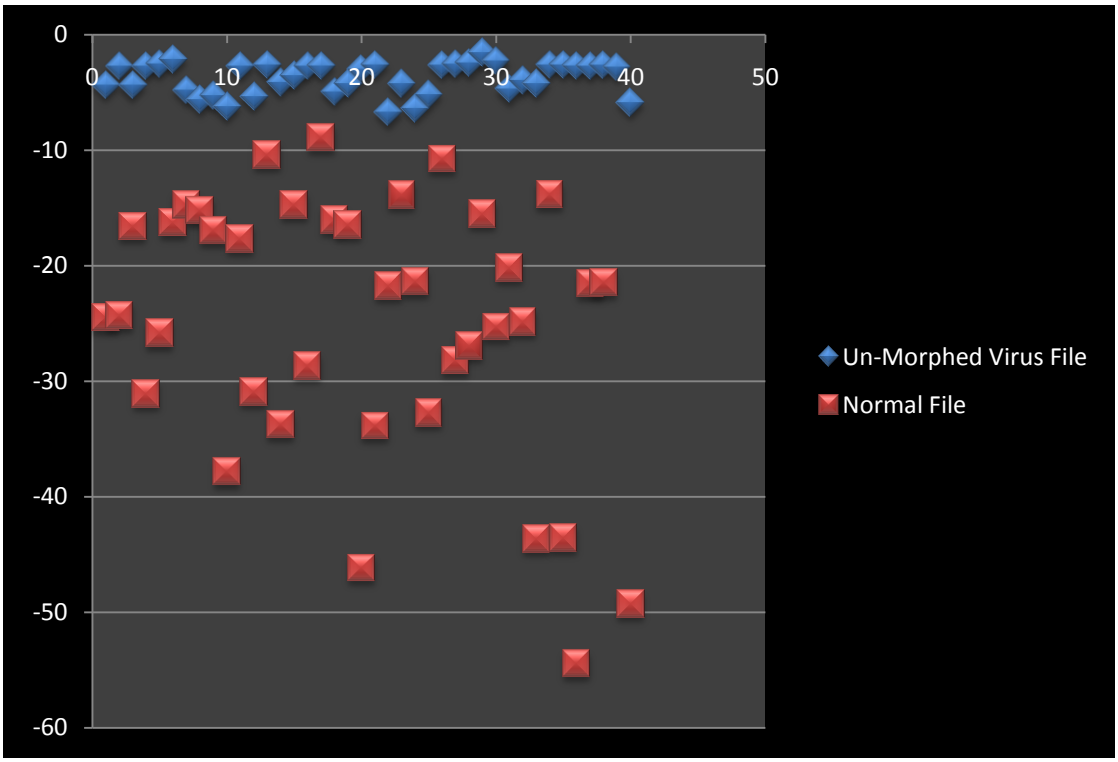


Figure 31 : HMM scores with 35% Morphing

9.4 Performance Analysis of Code Emulator

To analyze the performance of our code emulator, we conducted two tests which have been discussed in this section. The first analysis deals with the execution time of the virus file by our code emulator. In the second analysis, we tried to ascertain the percentage of actual code (undead), which our code emulator missed during emulation of a virus file.

9.4.1 Execution Time Analysis

We wanted to get an idea about the performance of our code emulator. Figure 32 shows the time analysis graph where the x-axis represents the virus file size in KB and the y-axis represents their execution time in milliseconds (Refer to Appendix M to see the time as per virus file name and their size). As the virus file size increased, the code emulator took more time to finish its operation.

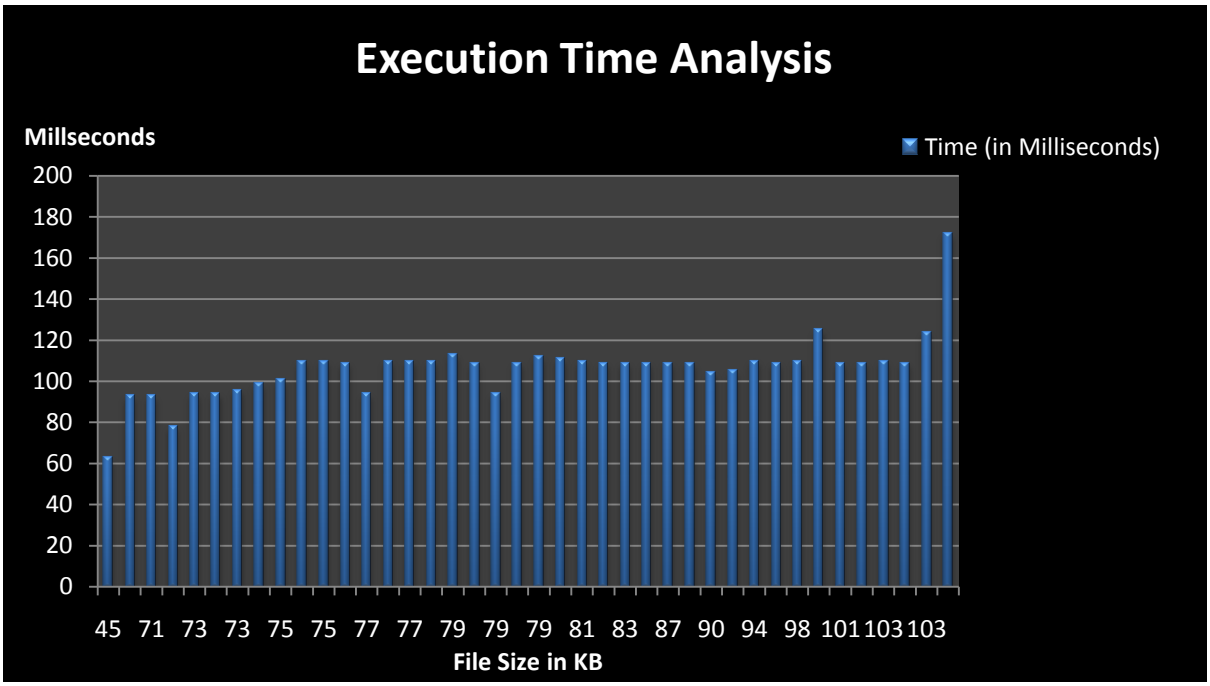


Figure 32 : Execution Time Analysis

9.4.2 Instruction Count Comparison

The code emulator tries to remove those instructions from the virus file which can be present due to various morphing techniques. While performing emulation, there can be instructions which are legitimate (undead), but are still removed by the code emulator. So, we compared the number of instructions in the base virus file to the number of instructions left in the virus files after emulation. According to Figure 33, we lost an average of 25 instructions per virus file after emulation (Refer to Appendix N for the exact values) i.e. around average 3.35% of original instructions. There were cases where no difference in the instruction count was found (like the virus files IDAN127 and IDAN139), but at the same time there were cases where the number of instructions lost due to emulation was 116 like the virus file IDAN125.

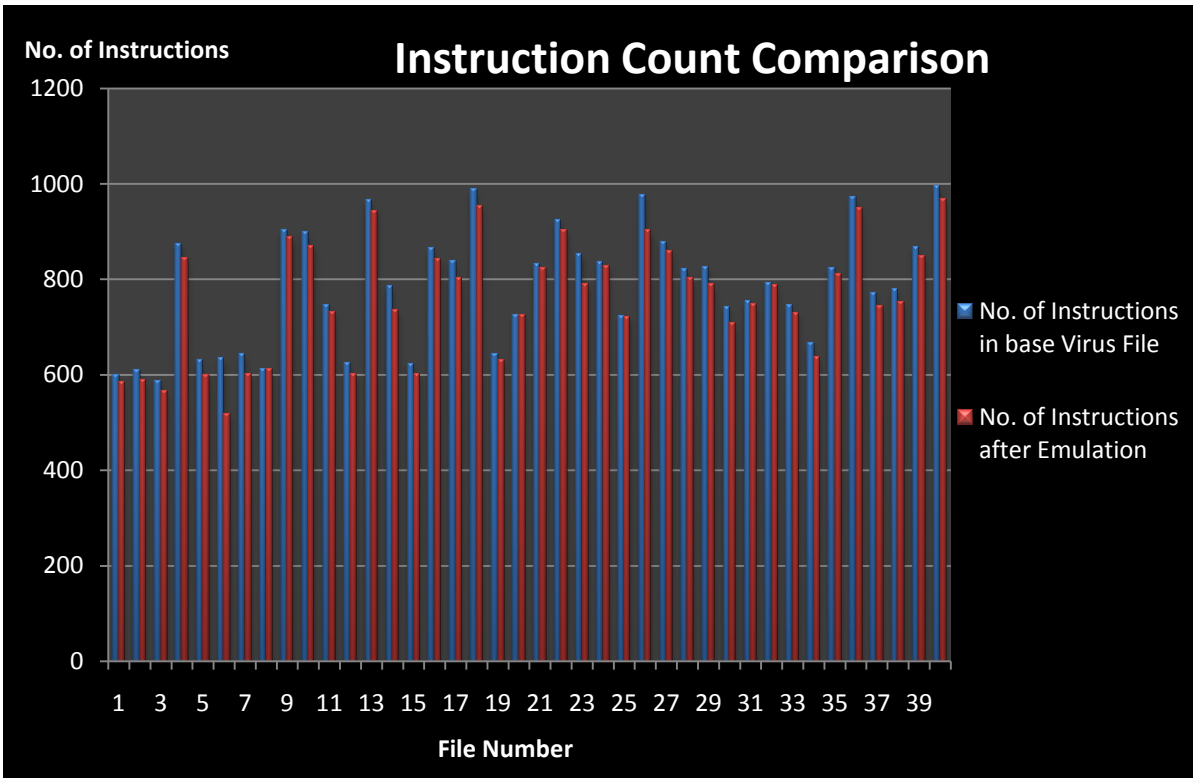


Figure 33 : Instruction Count Comparison

10 Attacks on Code Emulator

The implemented code emulator does have certain limitations which a virus writer can exploit to make the virus more complicated for any code emulator. The NGVCK generated viruses had one entry point from where our code emulator starts its process. Virus writers can introduce multiple entry points for any virus for which an advanced emulator will have to perform its operations from all the respective entry points. In our code emulator, we took the exception of Dummy Loops Detection. Based on some conditions, these loops are inserted to make the emulator run thousands of instruction unnecessarily thus preventing it from rebuilding the original base virus file.

In the Code Execution Recording phase, we followed the path of the instructions being executed. But there can be viruses where the instructions are based over the CPU properties. The overhead will be that the emulator will have to run over different kind of CPU to detect the virus's actual behaviors.

11 Conclusions and Future Work

The emulator we developed was able to emulate the execution of virus files and remove the unexecuted instructions/subroutines successfully. The code emulator was also able to remove or change the instructions which were result of various code obfuscation techniques such as equivalent instruction substitution, junk code/block insertion and dead code insertion. Once the virus files were un-morphed by our code emulator, the HMM tool which was not able to classify the virus files from the normal files (after the virus files were morphed by the metamorphic engine) are now able to classify them.

The virus files which were morphed up to as high as 35% (with 15% to 25% subroutine copying) also exposed themselves in our code emulator. We also showed that though code emulator is complex to implement, but with a good design and algorithm it can be a very powerful tool to detect not only metamorphic viruses but also any kind of virus.

The code emulator can be made a real powerful tool once many new techniques have been incorporated to the existing one. We listed few of our weaknesses with which our code emulator can be attacked. Handling these issues could be the next logical step in an attempt to improve the code emulator. One very challenging task what we anticipate is to make the code emulator very efficient. Our code emulator had a few steps which was a kind of "add on" to help the actual emulation. To remove these steps will be a beneficial step towards increasing its efficiency.

Other very interesting work which can be done is to combine the HMM and the code emulator in one package. The automation of processes like disassembling .exe files and making HMM models would be very beneficial. This will be full of new challenges, but end product, if achieved, can be a wonderful tool to find metamorphic viruses. The present code emulator did not take care of I/O devices emulation. Even though special treatments are required for each I/O device, which will be very comprehensive to implement, few common features like managing interrupts (both hardware and software) and physical memory access can be implemented.

References

[1] J. Borello and L. Me, “Code Obfuscation Techniques for Metamorphic Viruses”, Feb 2008, http://www.springerlink.com/content/233883w3r2652537

[2] P. Szor, “The Art of Computer Virus Defense and Research,” Symantec Press 2005.

[3] HowStuffWorks, “Computer & Internet Security,” May 2008, http://computer.howstuffworks.com/virus.htm

[4] Orr, “The Molecular Virology of Lexotan32: Metamorphism Illustrated,” 2007. http://www.antilife.org/files/Lexo32.pdf

[5] A. Venkatesan, “Code Obfuscation and Metamorphic Virus Detection,” Master’s thesis, San Jose State University, 2008.

[6] Walenstein, R. Mathur, M. Chouchane, R. Chouchane, and A. Lakhotia, “The Design Space of Metamorphic Malware,” In Proceedings of the 2nd International Conference on Information Warfare, March 2007.

[7] Dr. Solomon's Virus Encyclopedia, 1995, ISBN 1897661002

[8] W. Wong, “Analysis and Detection of Metamorphic Computer Viruses,” Master’s thesis, San Jose State University, 2006. http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf

[9] John Aycock, “Computer Viruses and Malware”, Springer Publications 2006

[10] S. Attaluri, “Profile hidden Markov models for metamorphic virus analysis,” Master’s thesis, San Jose State University, 2007. http://www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pdf

[11] Von Neumann, John (1966). "Theory of Self-Reproducing Automata". Essays on Cellular Automata (University of Illinois Press): 66–87. Retrieved June 10., 2010

[12] http://en.wikipedia.org/wiki/Computer_virus

[13] PCWorld. (2008). “Security Worries for 2004”, Retrieved August 10, 2010,from http://www.pcworld.com/article/114058/security_worries_for_2004.html 32

[14] VX Heavens, http://vx.netlux.org/

[15] P. Szor, P. Ferrie, “Hunting for Metamorphic”, Symantec Security Response. http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

[16] Da Lin, “Hunting for Undetectable Metamorphic Viruses”, Master’s thesis, San Jose State University, 2010

[17] The Ollydbg Debugger, http://webster.cs.ucr.edu/AsmTools/OllyDbg/

[18] ProgrammersHeaven, “Inject Code To Portable Executable File,” 2010, http://www.programmersheaven.com/2/Inject-code-to-Portable-Executable-file

[19] http://en.wikipedia.org/wiki/Markov_process

[20] M. Stamp, “A Revealing Introduction to Hidden Markov Models,” January 2004. http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf

[21] P Desai, “Towards an Undetectable Computer Virus”, Master’s thesis, San Jose State University, 2008

[22] IDA Pro, http://www.hex-rays.com/idapro/

[23] Wikipedia, “8086 Instructions,” April 2010, http://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086.2F8088_instructions

[24] http://www.osdata.com/topic/language/asm/register.htm

[25] VX Heavens, “Next Generation Virus Construction Kit,” June 2001, http://vx.netlux.org/vx.php?id=tn02

[26] Cygwin, “Cygwin Utility Files,” http://www.cygwin.com/

[27] Oodesign, “Singleton Pattern,” http://www.oodesign.com/singleton-pattern.html

[28] Wikipedia, “Encryption with a Variable Key,” April 2010, http://en.wikipedia.org/wiki/Computer_virus#Encryption_with_a_variable_key

[29] “Benny/29A,” Theme: metamorphism, http://www.vx.netlux.org/lib/static/vdat/epmetam2.htm

[30] Wikipedia, “Polymorphic Code,” April 2010, http://en.wikipedia.org/wiki/Computer_virus#Polymorphic_code

[31] Wisegeek, “What is Malware,” May 2010, http://www.wisegeek.com/what-is-malware.htm

[32] Wikipedia, “Signature Based Detection,” April 2011, http://en.wikipedia.org/wiki/Antivirus_software#Signature-based_detection

[33] About, “What is a Virus Signature,” May 2010, http://antivirus.about.com/od/whatisavirus/a/virussignature.htm

Appendix A: Equivalent instruction substitution [21]

Notations:
R – Register (eax, ax, ah, al)
RR – Random register
mem, [mem] – Memory address ([esi])
imm – Immediate value (12h)
op1 – To-operand with length more than 1 including R and mem
op2 – From-operand with length more than 1 including R, mem, and imm
loc – any location or label

add R, imm	3. sub R, new_imm where new_imm = imm x (- 1) 4. lea R, [R + imm]
add R, 1	3. not R neg R
mov R, imm	1. mov R, random_imm add R, new_imm where new_imm = imm – random_imm 2. mov R, random_imm sub R, new_imm where new_imm = (random_imm - imm) mov R, random_imm xor R, new_imm
mov R1, R2 (no 8 bit R)	1. push R2 pop R1
mov R, mem (no 8 bit R)	1. push mem pop R
mov R, imm (no 8 bit R)	1. push imm pop R 2. lea R, [imm]
mov mem, R (no 8 bit R)	1. push R pop mem
mov mem, imm	1. push imm pop mem
cmp R, 0	1. or R, R 2. and R, R 3. test R, R
cmp R1, R2	1. sub R1, R2
cmp R, mem	1. sub R, mem
cmp R, imm	1. sub R, imm
cmp mem, R	1. sub mem, R
cmp mem, imm	1. sub mem, imm
and R1, R2	1. push RR mov R, R1

	or R, R2 xor R1, R2 xor R1, R pop RR 2. not R1 not R2 or R1, R2 not R1
dec R	1. neg R not R
dec mem	1. neg mem not mem
inc R	1. add R, 1 2. not R neg R
inc mem	1. add mem, 1 2. not mem neg mem
invoke op1, op2	1. stdcall [op1], op2
jmp loc	1. cmp RR, RR jz loc
jmp R	1. push R ret
lea R, [R1 + R2]	1. mov R, R1 add R, R2
lea R, [R + R1 + imm]	1. add R, imm add R, R1
lea R, [R1 + R2 + imm]	1. lea R, [R1 + imm] add R, R2
lodsb	1. mov al, [esi] add esi, 1
lodsd	1. mov eax, [esi] add esi, 4

movsb	1. push eax
	mov al, [esi]
	add esi, 1
	mov [edi], al
	add edi, 1
	pop eax

movsd	1. push eax
	mov [eax], esi
	add esi, 4
	mov [edi], eax

	add edi, 4 pop eax
neg R	1. not R add R, 1
neg mem	1. not mem add mem, 1

not R	1. neg R
	sub R, 1
	2. neg R
	dec R
	3. neg R
	add R, -1
	4. xor R, -1

not mem	1. neg mem
	sub mem, 1
	2. neg mem
	dec mem
	3. neg mem
	add mem, -1

or R1, R2	1. push RR mov RR, R1 xor RR, R2 and R1, R2 xor R1, RR pop RR
or R1, mem	1. push RR mov RR, R1 xor RR, mem and R1, mem xor R1, RR pop RR
or R1, imm	1. push RR mov RR, R1 xor RR, imm and R1, imm xor R1, RR pop RR
or mem, R	1. push RR mov RR, mem xor RR, R and mem, R xor mem, RR pop RR

Table 12 : Equivalent Instruction Substitution [21]

Appendix B: Dead Code Instructions [21]

Appendix A: Dead code instructions

Transfer Dead Code

1. mov R, R
2. push R followed by pop R

Arithmetic Dead Code

1. add R, 0
2. sub R, 0
3. adc bx, 0
4. sbb bx, 0
5. inc R followed by dec R

Logical Dead Code

1. shl R, 0
2. shr R, 0
3. and R, 1
4. test R, 1
5. or R ,0
6. xor R, 0

Floating Point Dead Code

1. fadd st2, st0
2. fmul st2, st0
3. fld st2
4. fsub st2, st0
5. fdiv st2, st0
6. fst st3

Miscellaneous Dead Code

1. nop
2. neg R, not R, dec R

Figure 34 : Dead Code Instructions [21]

Appendix C: List of 8086 Instructions [23]

Instruction	Meaning	Notes
AAA	ASCII adjust AL after addition	used with unpacked binary coded decimal
AAD	ASCII adjust AX before division	8086/8088 datasheet documents only base 10 version of the AAD instruction (opcode 0xD5 0x0A), but any other base will work. Later Intel's documentation has the generic form too. NEC V20 and V30 (and possibly other NEC V-series CPUs) always use base 10, and ignore the argument, causing a number of incompatibilities
AAM	ASCII adjust AX aftermultiplication	Only base 10 version is documented, see notes for AAD
AAS	ASCII adjust AL after subtraction	
ADC	Add with carry	destination := destination + source + carry_flag
ADD	Add	
AND	Logical AND	
CALL	Call procedure	
CBW	Convert byte to word	
CLC	Clear carry flag	
CLD	Clear direction flag	
CLI	Clear interrupt flag	
CMC	Complement carry flag	
CMP	Compare operands	
CMPSB	Compare bytes in memory	
CMPSW	Compare words	
CWD	Convert word to doubleword	
DAA	Decimal adjust AL after addition	(used with packed binary coded decimal)
DAS	Decimal adjust AL aftersubtraction	
DEC	Decrement by 1	
DIV	Unsigned divide	
ESC	Used with floating-point unit	
HLT	Enter halt state	
IDIV	Signed divide	
IMUL	Signed multiply	
IN	Input from port	
INC	Increment by 1	
INT	Call to interrupt	
INTO	Call to interrupt if overflow	
IRET	Return from interrupt	
Jxx	Jump if condition	(*JA, JAE, JB, JBE, JC, JCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ*)
JMP	Jump	

LAHF	Load flags into AH register	
LDS	Load pointer using DS	
LEA	Load Effective Address	
LES	Load ES with pointer	
LOCK	Assert BUS LOCK# signal	(for multiprocessing)
LODSB	Load signed byte	
LODSW	Load signed word	
LOOP/LOOPx	Loop control	(*LOOPE, LOOPNE, LOOPNZ, LOOPZ*)
MOV	Move	
MOVSB	Move byte from string to string	
MOVSW	Move word from string to string	
MUL	Unsigned multiply	
NEG	Two's complement negation	
NOP	No operation	opcode (0x90) equivalent to XCHG EAX, EAX
NOT	Negate the operand, logical NOT	
OR	Logical OR	
OUT	Output to port	
POP	Pop data from stack	POP CS (opcode 0x0F) works only on 8086/8088. Later CPUs use 0x0F as a prefix for newer instructions.
POPF	Pop data into flags register	
PUSH	Push data onto stack	
PUSHF	Push flags onto stack	
RCL	Rotate left (with carry)	
RCR	Rotate right (with carry)	
REPxx	Repeat MOVS/STOS/CMPS/LODS/SCAS	(*REP, REPE, REPNE, REPNZ, REPZ*)
RET	Return from procedure	
RETN	Return from near procedure	
RETF	Return from far procedure	
ROL	Rotate left	
ROR	Rotate right	
SAHF	Store AH into flags	
SAL	Shift Arithmetically left (signed shift left)	
SAR	Shift Arithmetically right (signed shift right)	
SBB	Subtraction with borrow	
SCASB	Compare byte string	
SCASW	Compare word string	
SHL	Shift left (unsigned shift left)	
SHR	Shift right (unsigned shift right)	

STC	Set carry flag	
STD	Set direction flag	
STI	Set interrupt flag	
STOSB	Store byte in string	
STOSW	Store word in string	
SUB	Subtraction	
TEST	Logical compare (AND)	
WAIT	Wait until not busy	Waits until BUSY# pin is inactive (used withfloating-point unit)
XCHG	Exchange data	
XLAT	Table look-up translation	
XOR	Exclusive OR	

Table 13 : List of 8086 Instructions [23]

Appendix D: HMM Model Trained N=2

Model
N=2, M=76, T=67032
I:
1.00000000000000 0.00000000000000
A:
0.31213745192201 0.68786254807796
0.99999374304194 0.00000625695803
B:

call	0.08218503496863	0.03371900087267
sub	0.06417040292496	0.02231340405298
pop	0.02997515656273	0.09260052958416
mov	0.19554566095211	0.25090974189873
or	0.01377346911843	0.00000000000000
jz	0.00000000000000	0.08610033301483
push	0.14347589399134	0.07921793237216
lea	0.01597360820614	0.01796039785637
xor	0.02014892699289	0.01342772980439
rol	0.00158144088397	0.00472946366722
add	0.16496314877007	0.10651431749596
cmp	0.06473782285829	0.00000000000000
jnz	0.00000000000000	0.06058505575661
test	0.00982020650126	0.00000000000000
jmp	0.03524565445683	0.00074140491387
sar	0.00130327978519	0.00077759378900
dec	0.03017438600252	0.00159803441059
pusha	0.01946415288583	0.00000000000000
popa	0.02102429536802	0.00406488936664
jb	0.00000000000000	0.01797417666253
movzx	0.00633346740711	0.00602089047992
imul	0.00000000000000	0.00585716551121
shl	0.00468050134375	0.01245081165768
movsb	0.00000000000000	0.00391697943562
lodsw	0.00056746394289	0.00104197944089
ror	0.00406322284121	0.00170710881670
stosw	0.00000000000000	0.00175714965336
clc	0.01198568793487	0.00000000000000
retn	0.00017857316316	0.07980051733038
stc	0.00808278535104	0.00000000000000
ja	0.00000000000000	0.00285536818672
and	0.00982310629799	0.01811639000946
jnb	0.00000000000000	0.00483216154675
inc	0.01471252036211	0.02030629820452
stosd	0.00000000000000	0.00248929534227
div	0.00000000000000	0.00582055822677

rcl	0.00111653269881	0.00716250747895
adc	0.00453676494134	0.00251956113741
cld	0.00449021197316	0.00152562965895
shr	0.00183230442523	0.00440135979471
rcr	0.00151080100019	0.00000000000000
not	0.00397610573962	0.00366412587481
neg	0.00337886115374	0.00233597665418
loop	0.00023144348474	0.00035906057008
start	0.00237618465993	0.00145082448129
jbe	0.00000000000000	0.00545448538232
xchg	0.00000000000000	0.00424644499563
lodsb	0.00059653230453	0.00045060992486
stosb	0.00000000000000	0.00135446952447
rep	0.00000000000000	0.00219643706670
sbb	0.00129044575376	0.00083285948202
lodsd	0.00021804863447	0.00122050185215
popf	0.00000000000000	0.00003660728445
bound	0.00000000000000	0.00003660728445
in	0.00000000000000	0.00010982185334
jnp	0.00005036003334	0.00000000000000
ins	0.00002397489038	0.00007496660907
fnstenv	0.00002518001667	0.00000000000000
scasb	0.00000000000000	0.00003660728445
retf	0.00004811051560	0.00003987768482
cmc	0.00000000000000	0.00003660728445
aad	0.00002518001667	0.00000000000000
enter	0.00002518001667	0.00000000000000
movsd	0.00005036003334	0.00000000000000
jp	0.00000000000000	0.00003660728445
repe	0.00000000000000	0.00010982185334
jns	0.00002518001667	0.00000000000000
fild	0.00002518001667	0.00000000000000
icebp	0.00002518001667	0.00000000000000
jecxz	0.00002518001667	0.00000000000000
std	0.00003128771775	0.00002772776885
jle	0.00002518001667	0.00000000000000
out	0.00002518001667	0.00000000000000
hlt	0.00002518001667	0.00000000000000
cmpsb	0.00000000000000	0.00003660728445
fidiv	0.00000000000000	0.00003660728445

Table 14 : HMM Model Trained N=2

Appendix E: HMM Model Trained N=3

N=3, M=76, T=67032

I:

1.00000000000000	0.00000000000000	0.00000000000000

A:

0.05276957768954	0.32624506516877	0.62098535714169
0.99351380535297	0.00648619464703	0.00000000000000
0.00000000000000	0.19527911680493	0.80472088319506

B:

call	0.10758113770840	0.08648240197820	0.04102623878677
sub	0.00000000000000	0.03581588477658	0.06531482231911
pop	0.18166133767637	0.00000000000000	0.03246089973430
mov	0.00000000000000	0.00214018531199	0.35144683990257
or	0.00012871267202	0.02145703596697	0.00669954940899
jz	0.18012165403566	0.00000000000000	0.00000000000000
push	0.12363627514111	0.38829768090538	0.03403992656142
lea	0.00587430282473	0.00000000000000	0.02524571594103
xor	0.00000000000000	0.00758730122965	0.02582966139870
rol	0.00015627484036	0.00000000000000	0.00457472755101
add	0.00012882190291	0.01315385159534	0.22386179377281
cmp	0.00000000000000	0.20651418412296	0.00000000000000
jnz	0.12674376591370	0.00000000000000	0.00000000000000
test	0.00008500730552	0.03123737790837	0.00000000000000
jmp	0.01849599396451	0.00227467727887	0.02769900085102
sar	0.00012832115939	0.00055517619690	0.00155122999419
dec	0.00000000000000	0.04816595322001	0.01546967979789
pusha	0.00000000000000	0.00000000000000	0.01861589698618
popa	0.00994806263832	0.06471620675649	0.00025081700451
jb	0.03760192692666	0.00000000000000	0.00000000000000
movzx	0.00000000000000	0.00000000000000	0.01001838699385
imul	0.00000000000000	0.00000000000000	0.00385322576687
shl	0.00082072400712	0.00000000000000	0.01240938862476
movsb	0.00000000000000	0.00000000000000	0.00257684473159
lodsw	0.00000000000000	0.00000000000000	0.00122821571319
ror	0.00063405649312	0.00000000000000	0.00480980329256
stosw	0.00000000000000	0.00000000000000	0.00115596773006
clc	0.00000000000000	0.03823444249029	0.00000000000000
retn	0.15195076882412	0.00000000057476	0.00488519056093
stc	0.00000000000000	0.02578415134324	0.00000000000000
ja	0.00597342220016	0.00000000000000	0.00000000000000
and	0.00000000000078	0.00257744170986	0.02054039345274
jnb	0.01010886833874	0.00000000000000	0.00000000000000
inc	0.00016599922965	0.01407682673458	0.02315747361172
stosd	0.00000000000000	0.00000000000000	0.00163762095092
div	0.00558348927739	0.00000000000000	0.00207331680300

rcl	0.01434107987383	0.00017317118835	0.00121811440210
adc	0.00219209464094	0.00181431389863	0.00476327726245
cld	0.00306595152299	0.00000000000000	0.00433404310556
shr	0.00035094525423	0.00010394120584	0.00450642931023
rcr	0.00000000000000	0.00481946754079	0.00000000000000
not	0.00000000000000	0.00000000000000	0.00621332654907
neg	0.00000000000000	0.00000000000000	0.00476836688650
loop	0.00000000000000	0.00000000000000	0.00045757055982
start	0.00000000000000	0.00349429548412	0.00217942640244
jbe	0.01141076804903	0.00000000000000	0.00000000000000
xchg	0.00000000000000	0.00000000000000	0.00279358868098
lodsb	0.00000000000000	0.00000000000000	0.00086697579755
stosb	0.00000000000000	0.00000000000000	0.00089105845859
rep	0.00000000000000	0.00000000000000	0.00144495966258
sbb	0.00057683585006	0.00000000000000	0.00160072074936
lodsd	0.00000000000000	0.00000000000000	0.00101147176380
popf	0.00000000000000	0.00000000000000	0.00002408266104
bound	0.00007658233590	0.00000000000000	0.00000000000000
in	0.00000000000000	0.00000000000000	0.00007224798313
jnp	0.00015250381015	0.00000069315403	0.00000000000000
ins	0.00000000000000	0.00000000000000	0.00007224798313
fnstenv	0.00000000000000	0.00000000000000	0.00002408266104
scasb	0.00000000000000	0.00000000000000	0.00002408266104
retf	0.00007456857443	0.00005653792738	0.00003184753700
cmc	0.00000000000000	0.00000000000000	0.00002408266104
aad	0.00000000000000	0.00008032445901	0.00000000000000
enter	0.00000000000000	0.00000000000000	0.00002408266104
movsd	0.00000000000000	0.00006517720536	0.00002862406947
jp	0.00007658233590	0.00000000000000	0.00000000000000
repe	0.00000000000000	0.00000000000000	0.00007224798313
jns	0.00000000000000	0.00000000000000	0.00002408266104
fild	0.00000000000000	0.00008032445901	0.00000000000000
icebp	0.00000000000000	0.00000000000000	0.00002408266104
jecxz	0.00000000000000	0.00008032445901	0.00000000000000
std	0.00000000000000	0.00000000000000	0.00004816532209
jle	0.00000000000000	0.00000000000000	0.00002408266104
out	0.00000000000000	0.00008032445901	0.00000000000000
hlt	0.00000000000000	0.00008032445901	0.00000000000000
cmpsb	0.00007658233590	0.00000000000000	0.00000000000000
fidiv	0.00007658233590	0.00000000000000	0.00000000000000

Table 15 : HMM Model Trained N=3

Appendix F: Scores of Base Virus Files vs Normal Files

Virus File	Score	Normal File	Score	Virus File	Score	Normal File	Score
IDAN120	-4.35317	IDAR0	-24.47949	IDAN140	-2.6305	IDAR20	-33.87
IDAN121	-2.72685	IDAR1	-24.31159	IDAN141	-2.775077	IDAR21	-21.767
IDAN122	-4.38473	IDAR2	-16.59831	IDAN142	-4.2499	IDAR22	-13.866
IDAN123	-2.71454	IDAR3	-31.06836	IDAN143	-4.33749	IDAR23	-21.388
IDAN124	-2.70386	IDAR4	-25.87769	IDAN144	-4.19953	IDAR24	-32.738
IDAN125	-2.71754	IDAR5	-16.272108	IDAN145	-2.63856	IDAR25	-10.778
IDAN126	-2.59433	IDAR6	-14.742251	IDAN146	-2.54809	IDAR26	-28.223
IDAN127	-2.68013	IDAR7	-15.2443731	IDAN147	-2.655287	IDAR27	-26.911
IDAN128	-4.26291	IDAR8	-16.95945	IDAN148	-2.58213	IDAR28	-15.533
IDAN129	-4.26499	IDAR9	-37.84378	IDAN149	-4.37208	IDAR29	-25.32
IDAN130	-2.71394	IDAR10	-17.687719	IDAN150	-4.33251	IDAR30	-20.19
IDAN131	-4.35447	IDAR11	-30.93656	IDAN151	-2.69571	IDAR31	-24.848
IDAN132	-2.62	IDAR12	-10.403818	IDAN152	-4.219643	IDAR32	-43.644
IDAN133	-4.1517	IDAR13	-33.739803	IDAN153	-2.62501	IDAR33	-13.854
IDAN134	-2.671508	IDAR14	-14.742432	IDAN154	-2.59279022	IDAR34	-43.583
IDAN135	-2.69673	IDAR15	-28.68644	IDAN155	-2.680101	IDAR35	-54.447
IDAN136	-2.70727	IDAR16	-8.907111	IDAN156	-2.74872	IDAR36	-21.556
IDAN137	-2.64067	IDAR17	-16.035186	IDAN157	-2.61201	IDAR37	-21.467
IDAN138	-4.19188	IDAR18	-16.483198	IDAN158	-2.772162	IDAR38	-169.93
IDAN139	-2.66747	IDAR19	-46.165916	IDAN159	-2.594808	IDAR39	-49.299

Table 16 : Scores of Base Virus Files vs Normal Files

Appendix G: HMM Test with 15% Morphing

Virus File	Score	Virus File	Score
IDAN120	-13.67460344	IDAN140	-2.61428
IDAN121	-16.930178	IDAN141	-5.0936136
IDAN122	-9.6491296	IDAN142	-5.328729
IDAN123	-3.803387	IDAN143	-4.1712801
IDAN124	-20.9437299	IDAN144	-20.622002
IDAN125	-6.8493165	IDAN145	-7.338346
IDAN126	-6.178747	IDAN146	-2.52195
IDAN127	-2.64337732	IDAN147	-17.528855
IDAN128	-5.27169	IDAN148	-4.392361
IDAN129	-20.120129	IDAN149	-4.264325
IDAN130	-9.75918045	IDAN150	-10.757379
IDAN131	-18.5724536	IDAN151	-4.3097744
IDAN132	-4.5055146	IDAN152	-3.957129
IDAN133	-19.83854	IDAN153	-10.37725
IDAN134	-14.925978	IDAN154	-33.456365
IDAN135	-16.4969121	IDAN155	-36.4617486
IDAN136	-3.633918	IDAN156	-8.557395
IDAN137	-2.684068	IDAN157	-12.0109775
IDAN138	-8.3367156	IDAN158	-13.6011
IDAN139	-13.95565	IDAN159	-38.3416859

Table 17 : HMM Test with 15% Morphing

Appendix H: HMM Test with 25% Morphing

Virus File	Score	Virus File	Score
IDAN120	-13.4838	IDAN140	-6.97814
IDAN121	-17.6592	IDAN141	-5.09419
IDAN122	-9.40807	IDAN142	-7.77197
IDAN123	-2.69738	IDAN143	-3.8671
IDAN124	-19.0013	IDAN144	-20.2359
IDAN125	-7.115	IDAN145	-11.298
IDAN126	-2.53906	IDAN146	-4.01597
IDAN127	-2.66287	IDAN147	-8.00268
IDAN128	-4.99566	IDAN148	-4.2418
IDAN129	-15.7902	IDAN149	-6.56102
IDAN130	-2.68677	IDAN150	-10.7584
IDAN131	-18.5724	IDAN151	-2.68392
IDAN132	-4.5047	IDAN152	-6.01988
IDAN133	-11.4683	IDAN153	-17.5171
IDAN134	-21.9264	IDAN154	-33.0852
IDAN135	-16.2617	IDAN155	-36.4615
IDAN136	-3.6265	IDAN156	-8.34347
IDAN137	-2.68939	IDAN157	-12.0109
IDAN138	-10.2884	IDAN158	-12.6013
IDAN139	-9.79208	IDAN159	-39.2773

Table 18 : HMM Test with 25% Morphing

Appendix I: HMM Test with 35% Morphing

Virus File	Score	Virus File	Score
IDAN120	-13.4835799	IDAN140	-6.9784975
IDAN121	-18.1800181	IDAN141	-5.052987
IDAN122	-3.836687	IDAN142	-7.7716808
IDAN123	-2.67901446	IDAN143	-4.171118355
IDAN124	-18.396751	IDAN144	-20.8734075
IDAN125	-6.848575	IDAN145	-2.6385652
IDAN126	-2.5380398	IDAN146	-3.9888902
IDAN127	-2.63963077	IDAN147	-8.00298211
IDAN128	-5.270669	IDAN148	-4.0379121
IDAN129	-25.396235	IDAN149	-6.5617813
IDAN130	-9.75924171	IDAN150	-10.757389
IDAN131	-18.36376	IDAN151	-2.717030095
IDAN132	-5.006305	IDAN152	-4.032057
IDAN133	-20.185832	IDAN153	-2.946882
IDAN134	-15.9534406	IDAN154	-31.68627444
IDAN135	-16.496435	IDAN155	-38.59124105
IDAN136	-3.7544547	IDAN156	-8.6172232
IDAN137	-2.682355886	IDAN157	-12.01148972
IDAN138	-8.5888858	IDAN158	-3.600851
IDAN139	-9.7920761	IDAN159	-39.27768023

Table 19 : HMM Test with 35% Morphing

Appendix J: HMM Test with 15% Morphing after Code Emulation

Un-Morphed Virus File	Score	Un-Morphed Virus File	Score
IDAN120	-5.35317	IDAN140	-2.6305
IDAN121	-2.2342	IDAN141	-2.775077
IDAN122	-6.39854	IDAN142	-4.2499
IDAN123	-3.3542	IDAN143	-4.33749
IDAN124	-2.70386	IDAN144	-4.19953
IDAN125	-2.71754	IDAN145	-2.63856
IDAN126	-2.59433	IDAN146	-2.54809
IDAN127	-4.62352	IDAN147	-2.655287
IDAN128	-5.12531	IDAN148	-2.58213
IDAN129	-4.22352	IDAN149	-4.37208
IDAN130	-2.2345	IDAN150	-4.33251
IDAN131	-4.35447	IDAN151	-2.69571
IDAN132	-2.62432	IDAN152	-1.3425
IDAN133	-1.83747	IDAN153	-2.23521
IDAN134	-2.672345	IDAN154	-5.23543
IDAN135	-2.65454	IDAN155	-2.680101
IDAN136	-5.72727	IDAN156	-2.74872
IDAN137	-2.64067	IDAN157	-2.873245
IDAN138	-4.19188	IDAN158	-2.772162
IDAN139	-2.66747	IDAN159	-2.594808

Table 20: HMM Test with 15% Morphing after Code Emulation

Appendix K: HMM Test with 25% Morphing after Code Emulation

Un-Morphed Virus File	Score	Un-Morphed Virus File	Score
IDAN120	-5.35317	IDAN140	-2.6305
IDAN121	-2.72685	IDAN141	-2.775077
IDAN122	-4.38473	IDAN142	-3.432599
IDAN123	-2.91657	IDAN143	-4.33749
IDAN124	-2.345226	IDAN144	-4.19953
IDAN125	-2.66654	IDAN145	-2.63856
IDAN126	-4.59433	IDAN146	-3.474509
IDAN127	-3.23484	IDAN147	-2.99987
IDAN128	-6.26291	IDAN148	-4.24513
IDAN129	-4.24523	IDAN149	-4.37208
IDAN130	-1.71394	IDAN150	-4.33251
IDAN131	-1.3234447	IDAN151	-2.69571
IDAN132	-2.62434	IDAN152	-4.219643
IDAN133	-4.48754	IDAN153	-2.62501
IDAN134	-3.252348	IDAN154	-2.59279
IDAN135	-2.45213	IDAN155	-2.680101
IDAN136	-1.90747	IDAN156	-2.74872
IDAN137	-2.64067	IDAN157	-2.61201
IDAN138	-4.19188	IDAN158	-2.772162
IDAN139	-2.66747	IDAN159	-2.594808

Table 21 : HMM Test with 25% Morphing after Code Emulation

Appendix L: HMM Test with 35% Morphing after Code Emulation

Un-Morphed Virus File	Score	Un-Morphed Virus File	Score
IDAN120	-4.3532	IDAN140	-2.6305
IDAN121	-2.7269	IDAN141	-6.7341
IDAN122	-4.3847	IDAN142	-4.2499
IDAN123	-2.7145	IDAN143	-6.3375
IDAN124	-2.5383	IDAN144	-5.2133
IDAN125	-2.1238	IDAN145	-2.6386
IDAN126	-4.7645	IDAN146	-2.5481
IDAN127	-5.6541	IDAN147	-2.4857
IDAN128	-5.2629	IDAN148	-1.5821
IDAN129	-6.2388	IDAN149	-2.3237
IDAN130	-2.7139	IDAN150	-4.7356
IDAN131	-5.3545	IDAN151	-3.954
IDAN132	-2.62	IDAN152	-4.2196
IDAN133	-4.1517	IDAN153	-2.625
IDAN134	-3.5539	IDAN154	-2.5928
IDAN135	-2.6967	IDAN155	-2.6801
IDAN136	-2.7073	IDAN156	-2.7487
IDAN137	-4.9234	IDAN157	-2.612
IDAN138	-4.1919	IDAN158	-2.7722
IDAN139	-2.9985	IDAN159	-5.8754

Table 22 : HMM Test with 35% Morphing after Code Emulation

Appendix M: Code Emulator – Execution Time Analysis

Virus File	File Size (in KB)	Time (in Milliseconds)	Virus File	File Size (in KB)	Time (in Milliseconds)
IDAN158	45	63	IDAN127	79	112
IDAN148	71	93	IDAN128	80	111
IDAN151	71	93	IDAN129	81	110
IDAN142	72	78	IDAN132	82	109
IDAN137	73	94	IDAN126	83	109
IDAN138	73	94	IDAN140	84	109
IDAN130	73	96	IDAN121	87	109
IDAN146	74	99	IDAN120	89	109
IDAN135	75	101	IDAN150	90	104
IDAN149	75	110	IDAN157	93	105
IDAN141	75	110	IDAN124	94	110
IDAN123	76	109	IDAN144	95	109
IDAN152	77	94	IDAN156	98	110
IDAN139	77	110	IDAN136	99	125
IDAN147	77	110	IDAN131	101	109
IDAN125	79	110	IDAN155	102	109
IDAN133	79	113	IDAN154	103	110
IDAN143	79	109	IDAN153	103	109
IDAN122	79	94	IDAN134	103	124
IDAN145	79	109	IDAN159	107	172

Table 23: Execution Time Analysis

Appendix N: Instruction Count Comparison

Virus File Name	No. of Instructions in base Virus File	No. of Instructions after Emulation	Difference	Percentage Lost
IDAN120	599	585	14	2.337228715
IDAN121	608	588	20	3.289473684
IDAN122	586	566	20	3.412969283
IDAN123	875	844	31	3.542857143
IDAN124	630	599	31	4.920634921
IDAN125	635	519	116	18.26771654
IDAN126	643	601	42	6.531881804
IDAN127	611	611	0	0
IDAN128	902	888	14	1.55210643
IDAN129	899	869	30	3.337041157
IDAN130	745	732	13	1.744966443
IDAN131	624	601	23	3.685897436
IDAN132	964	943	21	2.178423237
IDAN133	785	735	50	6.369426752
IDAN134	623	601	22	3.531300161
IDAN135	863	841	22	2.549246813
IDAN136	838	801	37	4.415274463
IDAN137	989	953	36	3.640040445
IDAN138	642	631	11	1.713395639
IDAN139	725	725	0	0
IDAN140	831	823	8	0.962695548
IDAN141	924	903	21	2.272727273
IDAN142	852	789	63	7.394366197
IDAN143	835	828	7	0.838323353
IDAN144	724	721	3	0.414364641
IDAN145	977	902	75	7.676560901
IDAN146	878	858	20	2.277904328
IDAN147	821	801	20	2.436053593
IDAN148	826	789	37	4.479418886
IDAN149	742	708	34	4.582210243
IDAN150	753	748	5	0.664010624
IDAN151	791	788	3	0.379266751
IDAN152	745	730	15	2.013422819
IDAN153	664	637	27	4.06626506
IDAN154	824	810	14	1.699029126
IDAN155	972	949	23	2.366255144
IDAN156	772	743	29	3.756476684
IDAN157	779	751	28	3.594351733
IDAN158	867	848	19	2.191464821
IDAN159	995	969	26	2.613065327

Table 24: Instruction Count Comparison

Appendix O: HMM Tests with Models Built with x% Morphed Virus Files

Considering the fact that the base virus files may not be always available, we performed few additional tests to see whether the code emulator and HMM can distinguish between the morphed virus and normal files. Idea was to make HMM models based on the morphed virus copies rather than using base virus files. We collected 200 morphed viruses having 15% morphing. HMM model was made using 160 of these morphed copies and the remaining 40 were used for HMM scoring. We repeated this process for 35%, 55% and 75% morphing too. We then also analyzed the detection rate before and after the emulation.

HMM Tests without Code Emulation

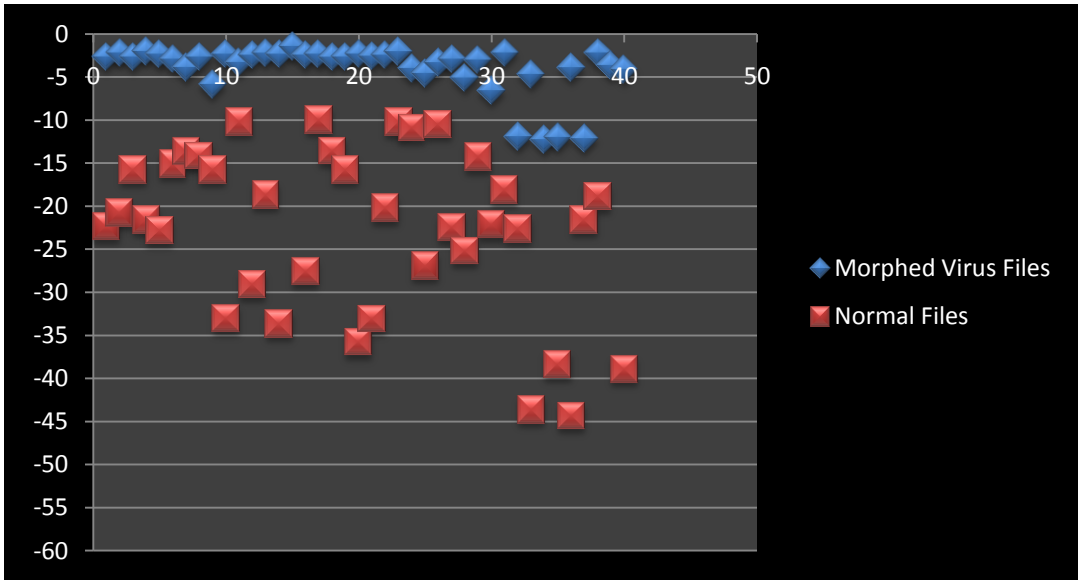


Figure 35 : HMM Test with 15% Morphing

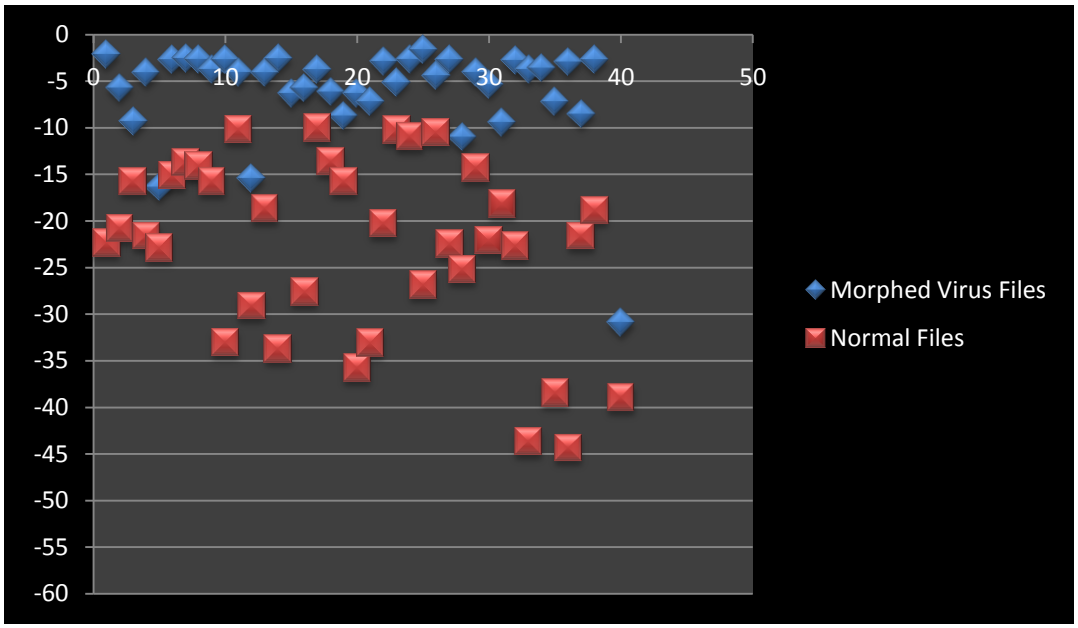


Figure 36 : HMM Test with 35% Morphing

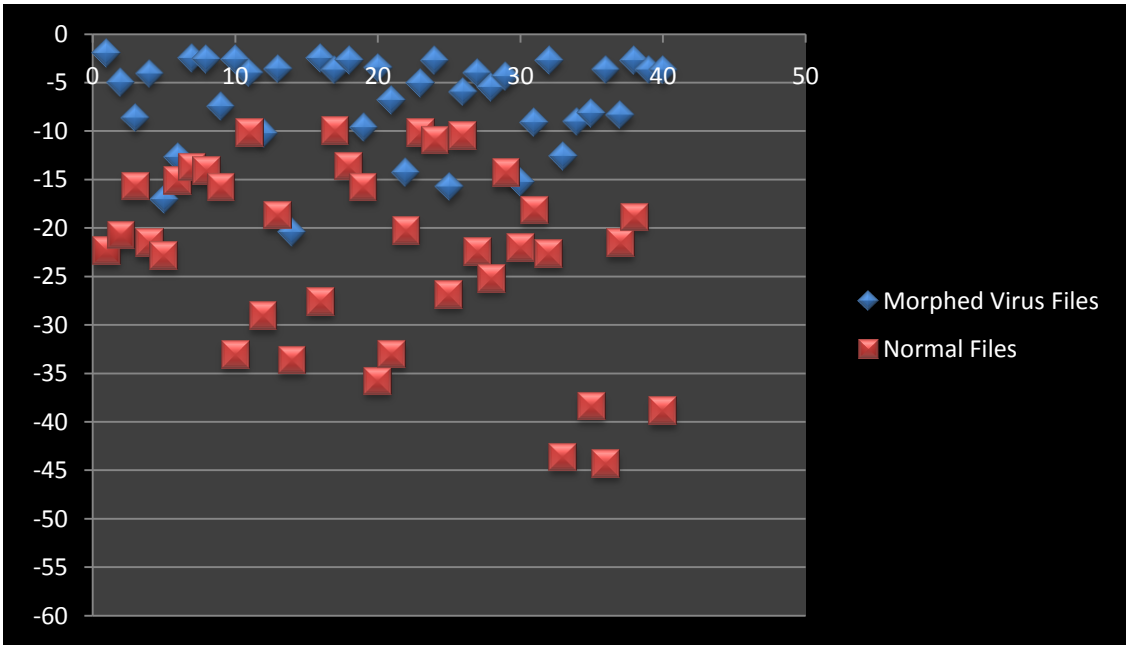


Figure 37 : HMM Test with 55% Morphing

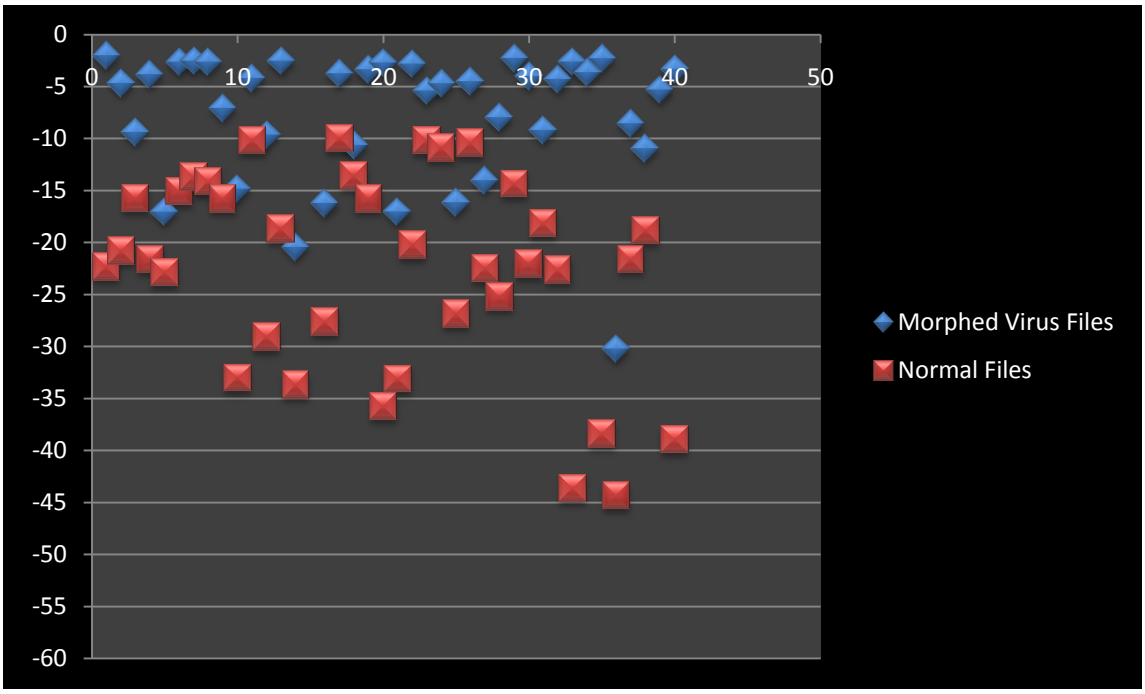


Figure 38 : HMM Test with 75% Morphing

HMM Tests with Code Emulation

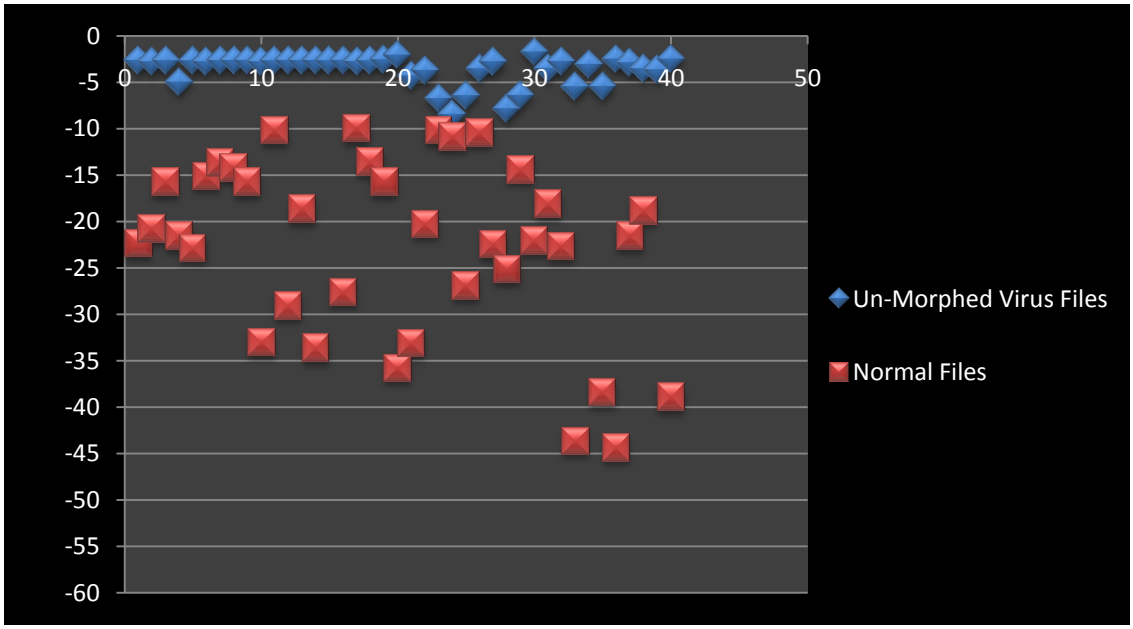


Figure 39 : HMM Test with 15% Morphing

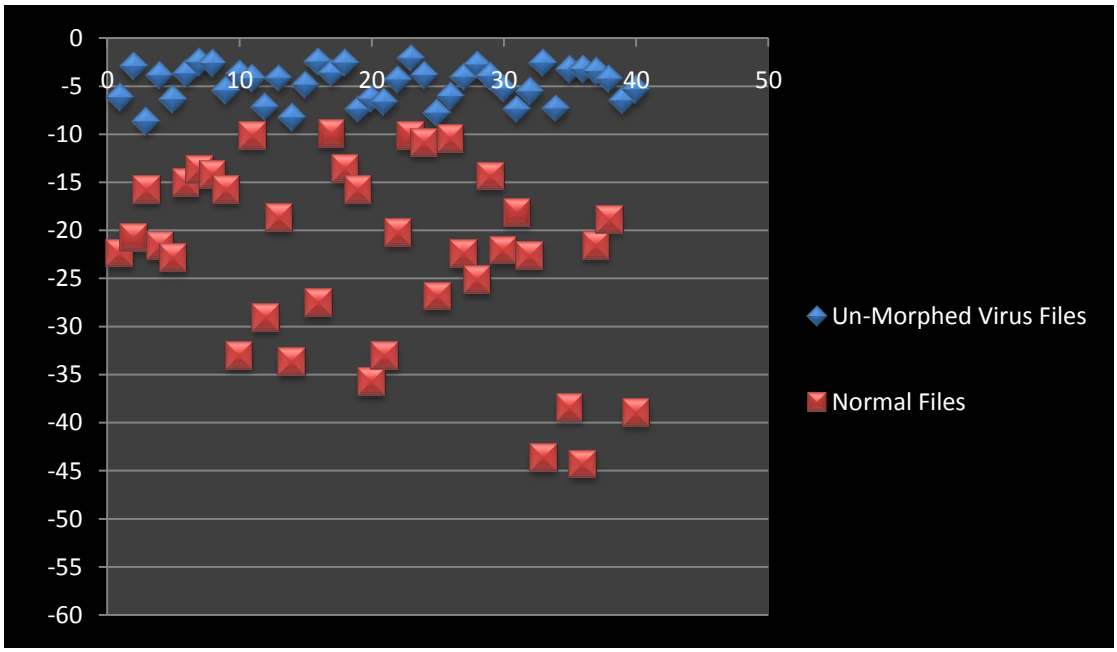


Figure 40 : HMM Test with 35% Morphing

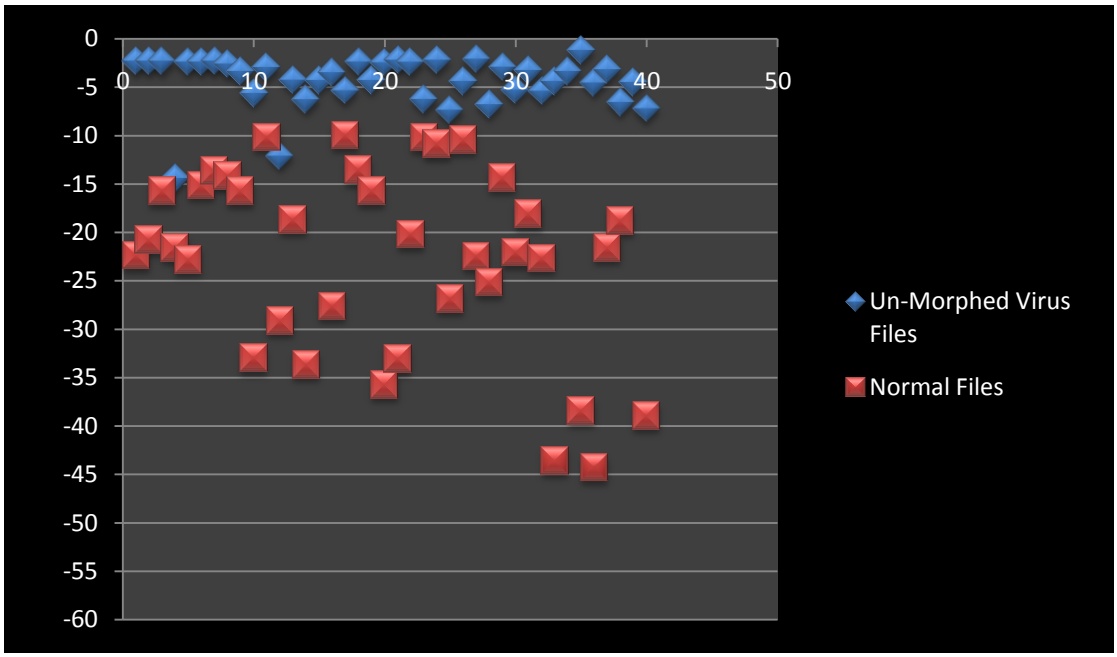


Figure 41 : HMM Test with 55% Morphing

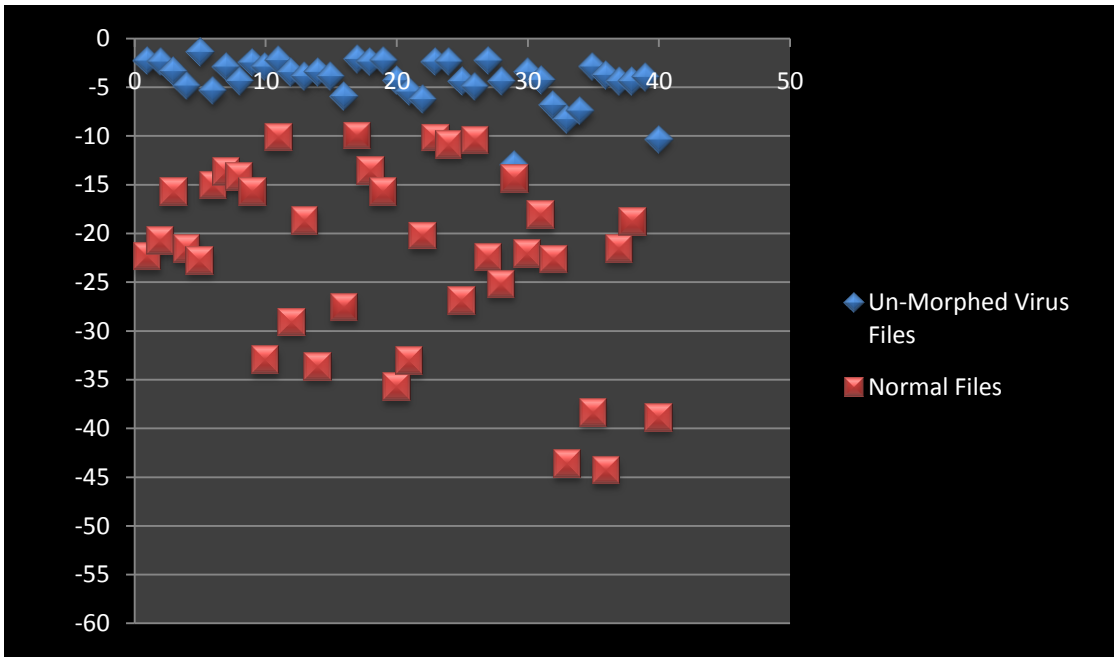


Figure 42 : HMM Test with 75% Morphing

Virus Detection Rate Comparison

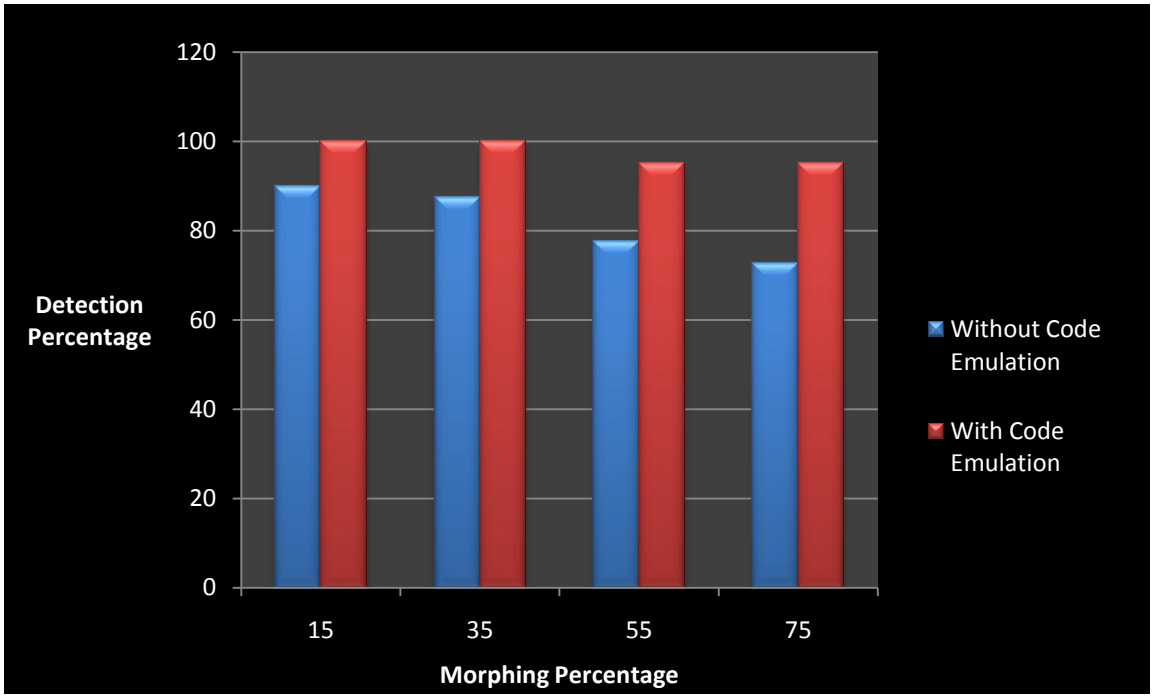


Figure 43 : Virus Detection Rate Comparison

Appendix P: HMM Tests with Training Files

HMM Tests without Code Emulation

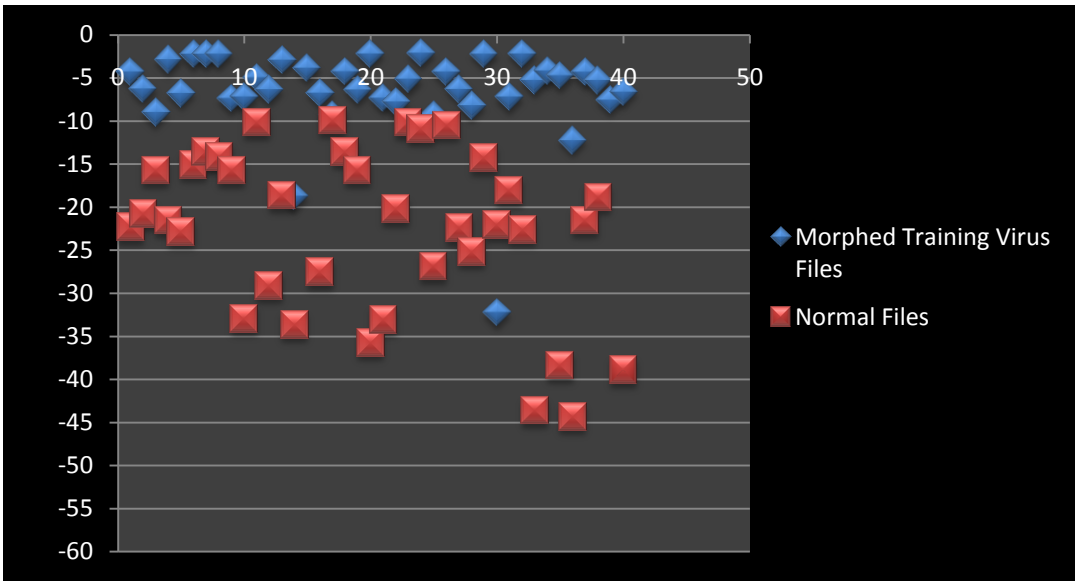


Figure 44 : HMM Test with 15% Morphing

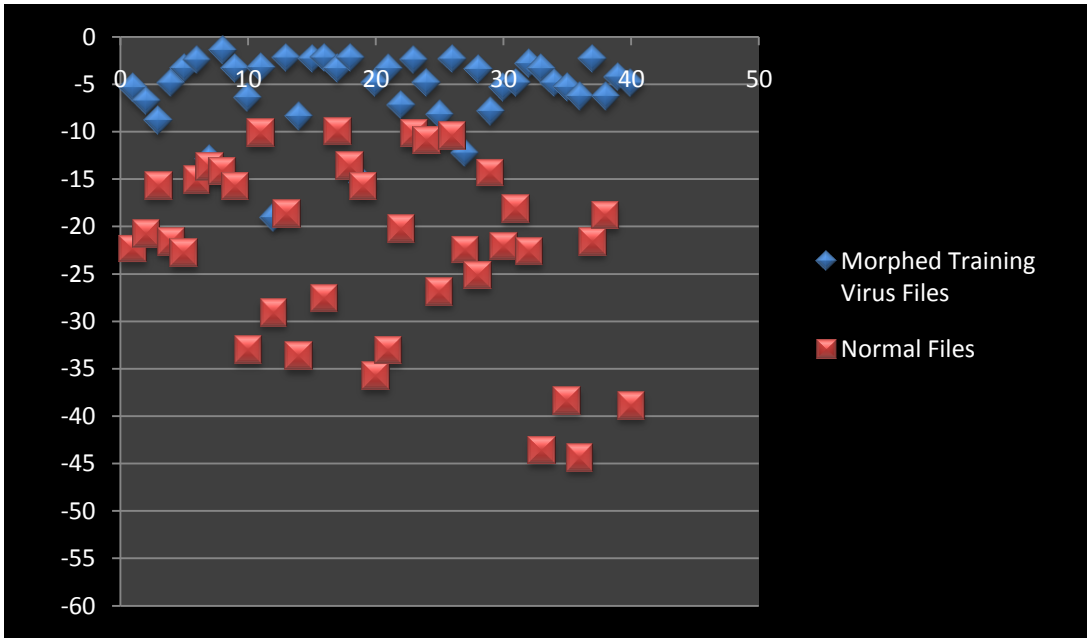


Figure 45 : HMM Test with 35% Morphing

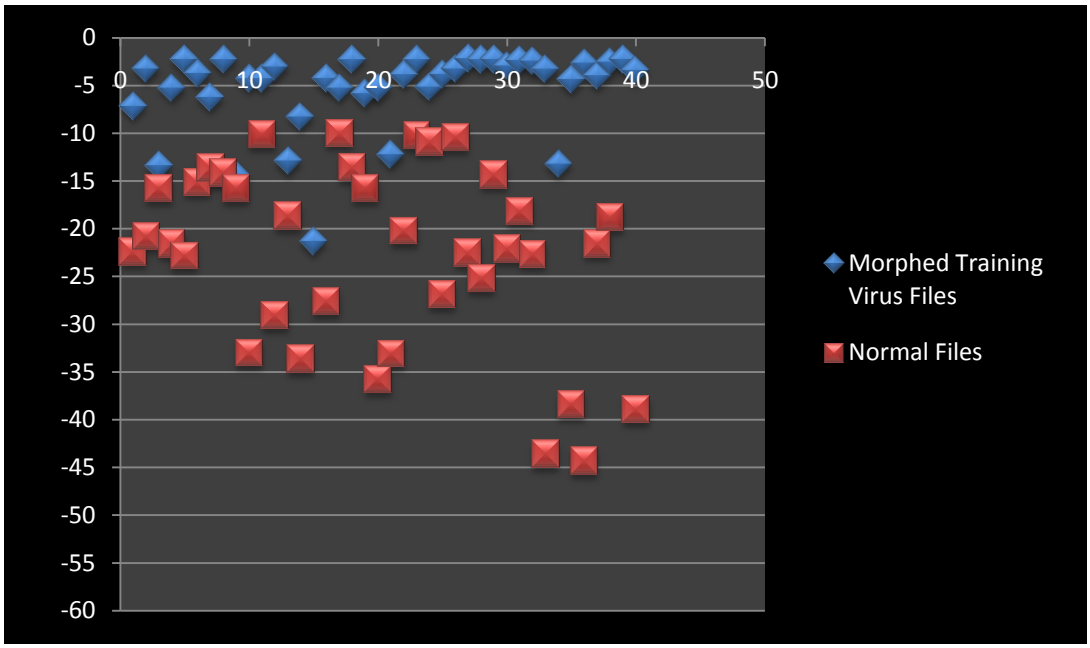


Figure 46 : HMM Test with 55% Morphing

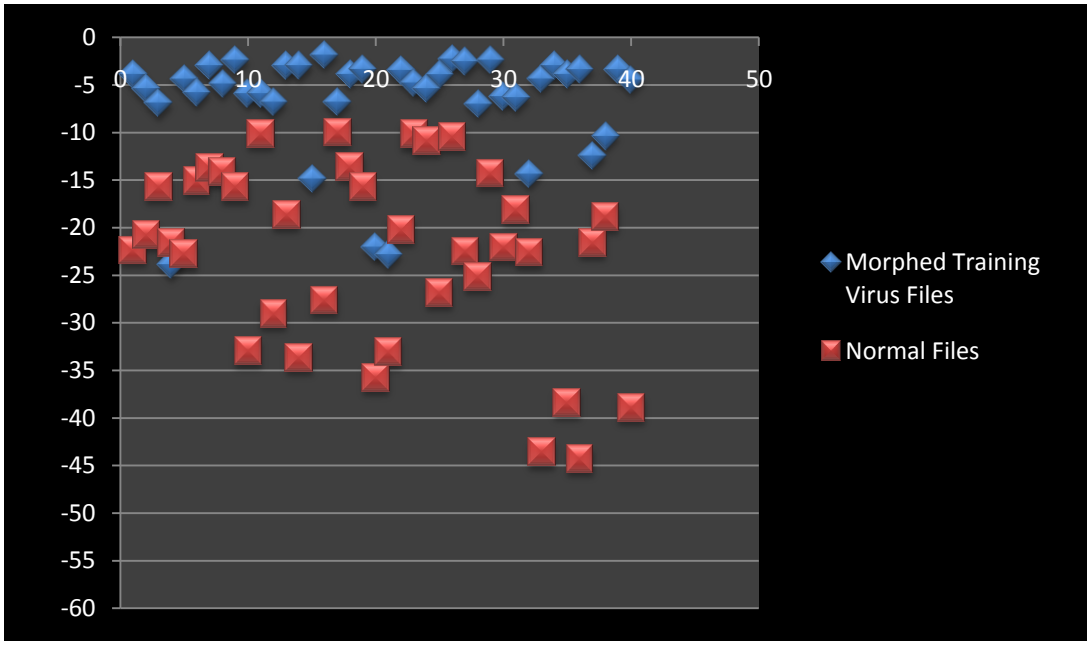


Figure 47 : HMM Test with 75% Morphing

HMM Tests with Code Emulation

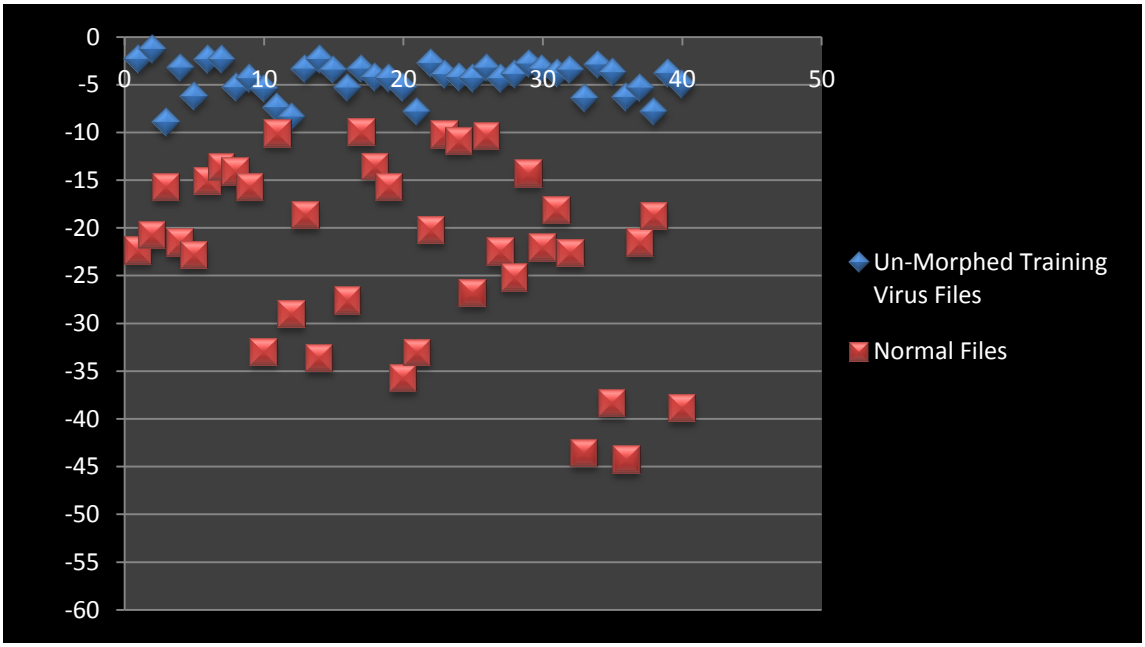


Figure 48 : HMM Test with 15% Morphing

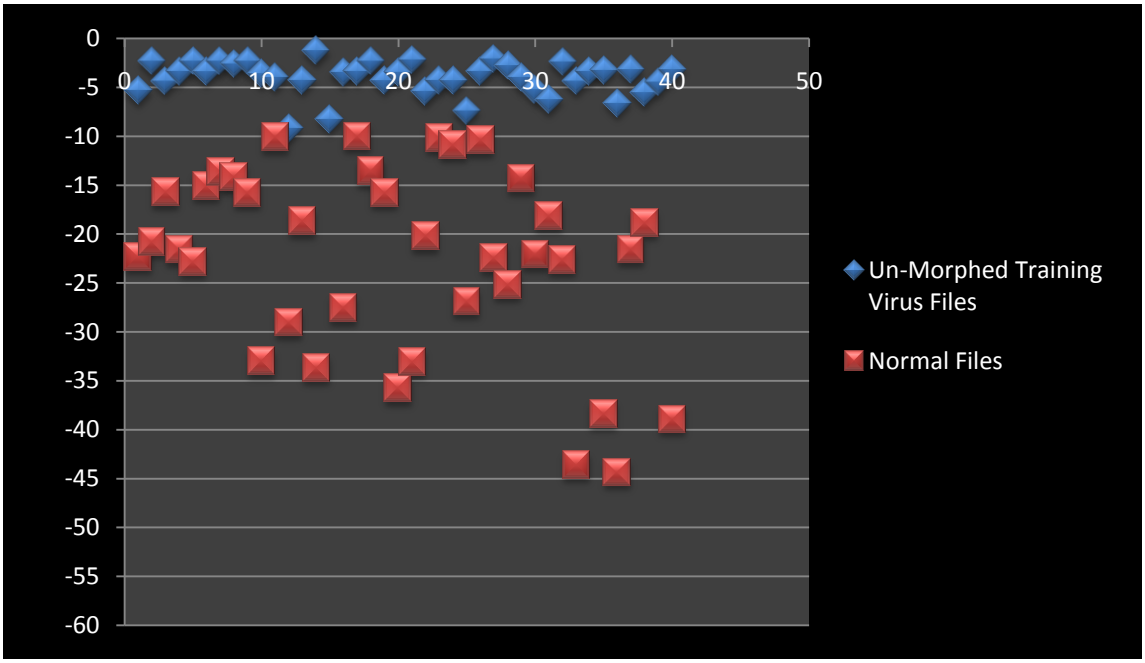


Figure 49 : HMM Test with 35% Morphing

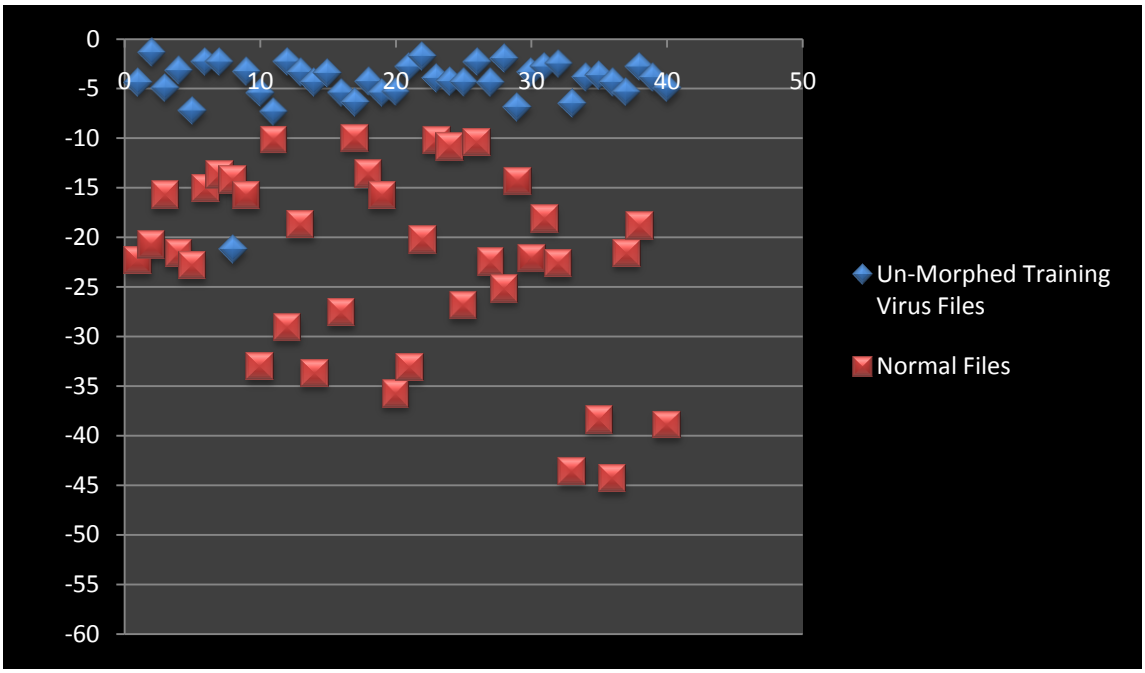


Figure 50 : HMM Test with 55% Morphing

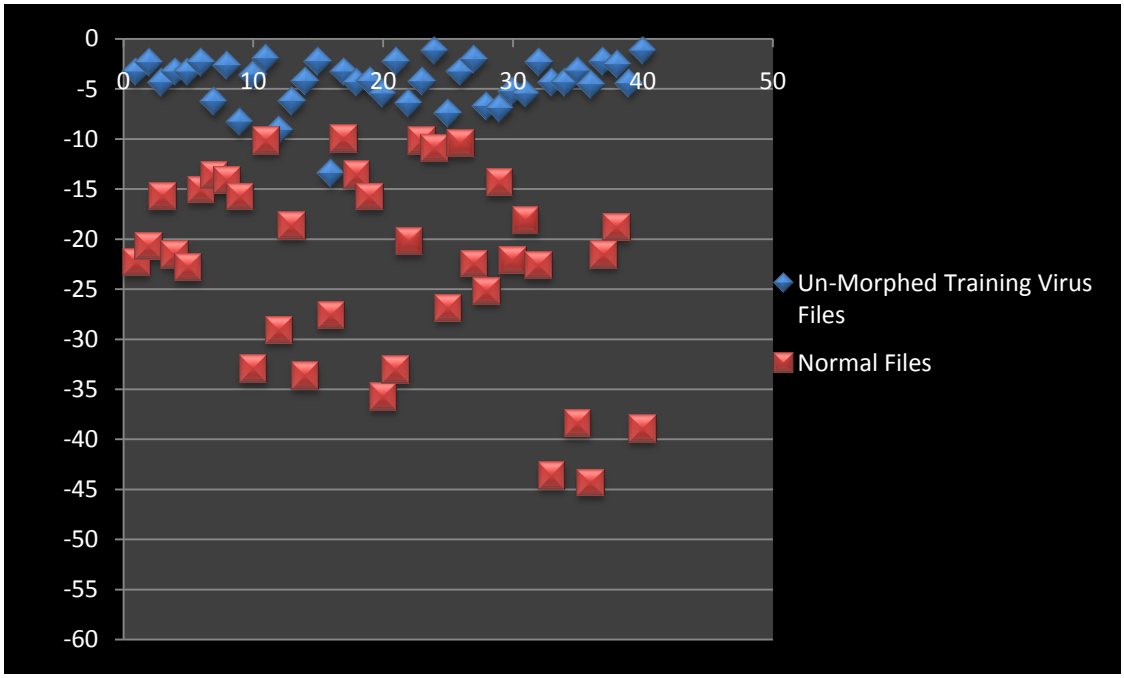


Figure 51 : HMM Test with 75% Morphing

Virus Detection Rate Comparison

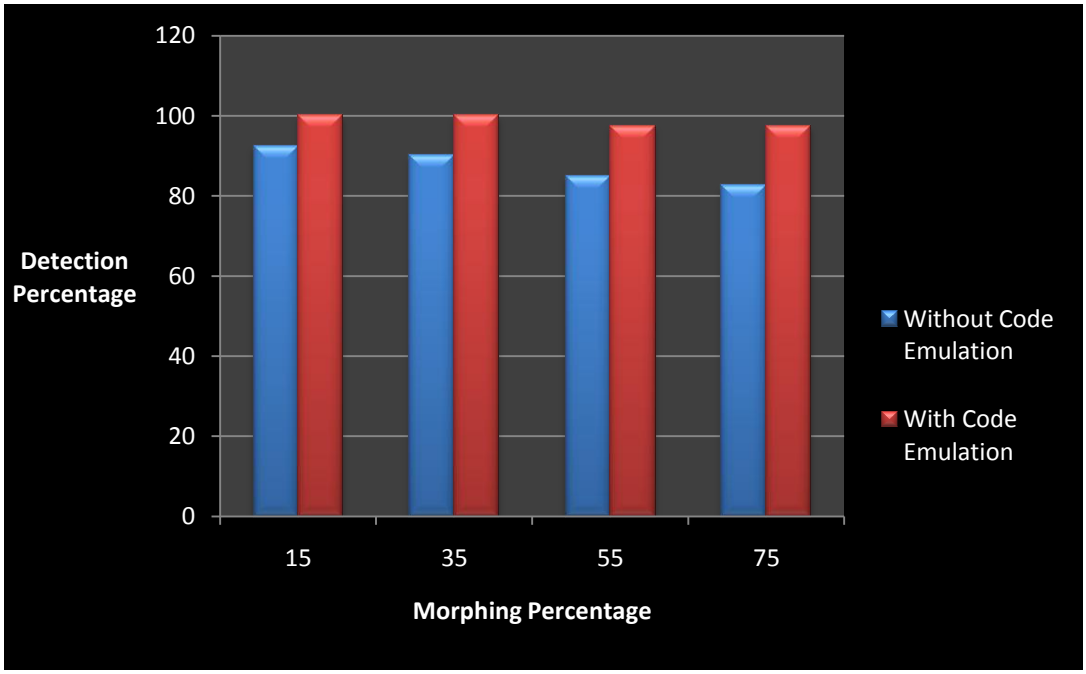


Figure 52 : Virus Detection Rate Comparison