

2009

Hunting for Undetectable Metamorphic Viruses

Da Lin
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects



Part of the [Computer Sciences Commons](#)

Recommended Citation

Lin, Da, "Hunting for Undetectable Metamorphic Viruses" (2009). *Master's Projects*. 18.
DOI: <https://doi.org/10.31979/etd.8eyg-zxga>
https://scholarworks.sjsu.edu/etd_projects/18

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Hunting for Undetectable Metamorphic Viruses

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

Da Lin

December 2009

Abstract

Hunting for Undetectable Metamorphic Viruses

by Da Lin

Commercial anti-virus scanners are generally signature based, that is, they scan for known patterns to determine whether a file is infected by a virus or not. To evade signature-based detection, virus writers have adopted code obfuscation techniques to create highly metamorphic computer viruses. Since metamorphic viruses change their appearance from generation to generation, signature-based scanners cannot detect all instances of such viruses.

To combat metamorphic viruses, detection tools based on statistical analysis have been studied. A tool based on hidden Markov models (HMMs) was previously developed and the results are encouraging—it has been shown that metamorphic viruses created by a well-designed metamorphic engine can be detected using an HMM.

In this project, we explore whether there are any exploitable weaknesses in this HMM-based detection approach. We create a highly metamorphic virus generating tool designed specifically to evade HMM-based detection. We then test our engine, showing that we can generate viral copies that cannot be detected using previously-developed HMM-based detection techniques. Finally, we consider possible defenses against our approach.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mark Stamp for trusting me with his idea. A special thanks is also owed to Dr. Stamp for his guidance, encouragement, and support throughout the duration of the project.

TABLE OF CONTENTS

1. Introduction.....	8
2. Computer Virus Evolution and Detection.....	9
2.1 Antivirus Defense Techniques.....	10
2.1.1 Signature Detection.....	10
2.1.2 Heuristic Analysis.....	11
2.2 Virus Evolution.....	11
2.2.1 Virus Obfuscation Techniques.....	11
2.2.2 Encrypted Viruses.....	12
2.2.3 Polymorphic Viruses.....	12
2.2.4 Metamorphic Viruses.....	12
2.2.5 Formal Grammar Mutation.....	17
3. Similarity and the HMM.....	18
3.1 Similarity Test.....	18
3.1.1 Similarity Test Method.....	18
3.1.2 Similarity Test Results.....	19
3.2 HMM.....	20
3.2.1 HMM Example.....	21
3.2.2 HMM as a Virus Detection Tool.....	27
4. Implementation.....	31
4.1 Introduction.....	31
4.2 Goal.....	31
4.3 Code Obfuscation Techniques Used.....	32
4.3.1 Dynamic Scoring Algorithm.....	32
4.3.2 Dead Code Insertion.....	35
4.3.3 Equivalent Instruction Substitution.....	37
4.3.4 Transposition.....	38
4.4 Metamorphic Engine Algorithm.....	39
5. Experiments.....	40
5.1 Base Virus.....	40

5.2 Similarity Test.....	42
5.3 HMM.....	44
5.3.1 Zero Percent Dead Code	44
5.3.2 Copying Blocks of Dead Code from Normal File	45
5.3.3 Copying Subroutines and Blocks of Dead Code from Normal File	47
5.3.4 Copying Subroutines Only from Normal File	49
6. Detection Technique for Our Engine	51
7. Conclusions.....	52
8. Future Works	53
Appendix A: Additional HMM results	57
Appendix B: Selected HMM models	68
Appendix C: Built-in Dead code instructions	75
Appendix D: Equivalent instruction substitution.....	76

LIST OF FIGURES

Figure 1. A virus can spell doom for your computer [24]	10
Figure 2. Multiple shapes of a metamorphic virus body [20].....	13
Figure 3. Two different generations of RegSwap [9]	14
Figure 4. Subroutine permutation	15
Figure 5. Zperm virus [19].....	16
Figure 6. Simple polymorphic decryptor	17
Figure 7. Generated polymorphic decryptor	17
Figure 8. Process of finding the similarity between two assembly programs [2].....	19
Figure 9. Similarity graph [2]	20
Figure 10. Generic Hidden Markov Model [14].....	21
Figure 11. Temperature transition probability	22
Figure 12. Tree size probability	22
Figure 13. HMM model [14]	23
Figure 14. Training data example [5]	28
Figure 15. HMM model example [5].....	29
Figure 16. HMM result example [5].....	30
Figure 17. Base virus opcodes and their frequencies [5].....	35
Figure 18. Opcodes of normal files and their frequencies [5]	36
Figure 19. Dead code insertion algorithm.....	37
Figure 20. Equivalent instruction substitution algorithm	38
Figure 21. Transposition algorithm.....	39
Figure 22. Metamorphic engine algorithm	40
Figure 23. HMM results for base viruses generated by NGVCK.....	41
Figure 24. Similarity score for morphed virus against normal files	42
Figure 25. Similarity score of virus and normal file.....	43
Figure 26. HMM result with 0% dead code copied.....	45
Figure 27. Maximum normal file scores vs. percentage increase.....	46
Figure 28. HMM results with 35% dead code copied	47
Figure 29. HMM result with 35% dead code blocks and 5% subroutine copied.....	49
Figure 30. HMM results with 5% subroutine copied.....	50
Figure 31. Ave. scores vs. subroutines copied.....	52
Figure 32. Average scores vs. percent of dead code copied	53
Figure 33. HMM results with 10% dead code copied	58
Figure 34. HMM results with 25% dead code copied	59
Figure 35. HMM results with 35% dead code copied	60
Figure 36. HMM results with 35% dead code blocks and 15% subroutines copied	62
Figure 37. HMM results with 35% dead code blocks and 20% subroutines copied	63
Figure 38. HMM results with 35% dead code blocks and 30% subroutines copied	65
Figure 39. HMM results with 15% subroutines copied.....	66
Figure 40. HMM results with 30% subroutines copied	67

LIST OF TABLES

Table 1. Examples of instruction substitution used by W32/MetaPhor virus [19].....	16
Table 2. Probabilities of observing (S, M, S, L) for all possible state sequences [14]	24
Table 3. Opcode sequences of virus file and normal files	33
Table 4. Opcode and opcode-pair counts lists	33
Table 5. Saved original subsequence score.....	34
Table 6. Subtract from old count and add new count	34
Table 7. New score after changes	34
Table 8. Updated opcode count lists	35
Table 9. Equivalent instructions for add [5]	38
Table 10. Similarity score of virus and its peer normal file.....	43
Table 11. HMM Results with 0% dead code copied	45
Table 12. HMM results with 35% dead code copied.....	47
Table 13. HMM results with 35% dead code blocks and 5% subroutine copied	48
Table 14. HMM results with 5% subroutine copied	50
Table 15. HMM results with 10% dead code copied.....	57
Table 16. HMM results with 25% dead code copied.....	59
Table 17. HMM results with 35% dead code copied.....	60
Table 18. HMM results with 35% dead code blocks and 15% subroutine copied	61
Table 19. HMM results with 35% dead code blocks and 20% subroutines copied.....	63
Table 20. HMM results with 35% dead code blocks and 30% subroutines copied.....	64
Table 21. HMM results with 15% subroutines copied	66
Table 22. HMM results with 30% subroutines copied	67
Table 23. HMM parameters (A, B, π) of the base virus with N = 3	68
Table 24. HMM parameters (A, B, π) of the virus without dead code copying with N = 3	70
Table 25. HMM parameters (A, B, π) of the virus with most dead code copied with N = 3	72

1. Introduction

A virus is a program designed to infect and potentially damage files on a computer [8]. To replicate itself, a virus must be permitted to execute code and write to memory. For this reason, many viruses attach themselves to executable files that are part of legitimate programs [16]. When an infected program is launched, the embedded virus is also executed and may replicate itself to infect other files and programs.

In general, a virus performs activities without permission of users. Some viruses may perform damaging activities on the host machine, such as corrupting hard disk data or crashing the computer. Other viruses are harmless and might, for example, print annoying messages on the screen, or do nothing at all. In any case, viruses are undesirable for computer users, regardless of their nature [12]. Modern viruses also take advantage of the always-connected Internet to spread on a global level. Therefore, early detection of viruses is necessary to minimize potential damage.

There are many antivirus defense mechanisms available today. The most widely used mechanism is signature detection, which detects viruses by searching the files on a computer system and looking for known binary strings—or other signatures—of viral files [1]. Another mechanism for virus detection is code emulation, which creates a virtual machine to execute suspicious programs and monitor unusual activities.

To evade signature detection, virus writers have adopted code obfuscation techniques to create highly metamorphic computer viruses. Since metamorphic viruses use various code obfuscation techniques to change their appearance from generation to generation, signature-based scanners might not be able to detect all generations of such viruses.

In order to combat metamorphic viruses, virus detection tools based on statistical analysis have been studied. A tool based on the Hidden Markov Model (HMM) was developed in [2], and the

results are encouraging. In [2], it was shown that metamorphic viruses created by a well-designed metamorphic engine could be detected using statistical analysis tools based on HMMs.

The goal of this project is to develop a standalone metamorphic engine to show that it is possible to defeat HMM-based detection tools developed in [2]. We employ code obfuscation techniques, including equivalent instruction substitution, dead code insertion, and rearrangement of instruction order. In addition, we have designed our metamorphic engine to generate highly discrete copies of the base virus. Furthermore, each discrete viral copy will randomly select a “normal” file and make itself similar to that normal file. These morphed copies have been tested against an HMM of the base virus family, normal files, and our own morphed copies. We also tested our morphed copies against commercial virus scanners.

This paper is organized as follows. In Section 2, we provide background information on computer viruses and discuss some possible defenses. Section 3 describes a similarity test that is useful for quantifying the degree of metamorphism, as well as describes HMMs and their use in detecting metamorphic viruses. Section 4 details the design and implementation of our metamorphic generator. Section 5 outlines the experimental results for our metamorphic virus engine. In Section 6, we consider a detection technique for identifying viruses generated by our engine. Section 7 presents our conclusions. Finally, we discuss possible extensions to the project and future work in Section 8.

2. Computer Virus Evolution and Detection

A computer virus is a small piece of software that piggybacks on real programs [24]. For example, a virus can insert itself into a spreadsheet program. When a user opens the spreadsheet and executes the program, the virus also runs, and it has the chance to reproduce (by attaching to other programs) and wreak havoc [24].



Figure 1. A virus can spell doom for your computer [24]

Viruses usually have an “infect” phase and an “attack” phase. When an embedded virus runs during the infect phase, it will try to infect other executables by copying itself into them. Viruses that do not have attack phases are considered harmless. These viruses just replicate themselves and generally do not impact the normal system operation. However, most viruses also have a destructive attack phase in which they do considerable damage. These viruses usually activate their attack phase based on some sort of event. During the attack phase, viruses will reveal themselves in tangible ways—by doing anything from displaying silly messages to destroying all of the computer’s data. The trigger event might be a date, the number of times the virus has been replicated, or something similar [24].

2.1 Antivirus Defense Techniques

Techniques for generating viruses have advanced over time, as have the anti-virus techniques for detecting such advanced viruses. In this section, we will outline some of the popular antivirus techniques.

2.1.1 Signature Detection

Signature detection is the earliest anti-virus technique and is still the most widely used technique today [4]. In general, a signature of a virus is a string of bits found in a virus, but not in other executables [17]. When a new virus is analyzed, its signature will be put into the virus scanner database. During the scanning process, a signature-based virus detection tool will search all of the files in a system for known signatures. It will flag the file as infected if a known virus signature is found. For example, when an executable file is infected by the W32/Beast virus, it

will contain a binary signature of “83EB 0274 EB0E 740A 81EB 0301 0000” [13]. The virus scanner searches executables for this signature. If this signature is present in any executable file, it is declared to be the Beast virus.

2.1.2 Heuristic Analysis

Heuristic analysis [23] is a method designed to detect previously unknown computer viruses, as well as new variants of viruses already in the wild. Heuristic analysis detects viruses by executing questionable programs or scripts in a virtual machine and monitoring them for common viral activities, such as replication, overwriting files, and attempts to hide. If such actions are detected, the suspicious programs will be flagged as viruses and will raise alerts.

Another method of heuristic analysis is to decompile the viral program, then analyze the code. This type of heuristic analysis then looks for instructions that are commonly found in viral programs. If the source code contains a certain percentage of instructions that match common viral instructions, the file is flagged and users are alerted.

The effectiveness of heuristic analysis is fairly low due to the number of false positives. The reason for this is that heuristic analysis mostly operates on the basis of past experience [23]; it might miss new viruses that contain codes not found in any previously known viruses. However, heuristic analysis is also evolving in terms of its ability to reveal new viruses, so it still provides some measure of detection.

2.2 Virus Evolution

2.2.1 Virus Obfuscation Techniques

Virus-like programs first appeared on microcomputers in the 1980s [19]. Since then, the battle between anti-virus (AV) researchers and virus writers continues. Virus writers constantly develop new obfuscation techniques to make virus code more difficult to detect [19]. To escape from generic scanning, a virus can modify its code and alter its appearance on each infection. The techniques that have been employed to achieve this end range from encryption, to polymorphic techniques, to modern metamorphic techniques [20].

2.2.2 Encrypted Viruses

The simplest method to hide the virus body is to encrypt it with different encryption keys. A virus generated by this method usually consists of a small decrypting engine (a decryptor) and the encrypted body. When executed, the virus will first decrypt itself and then create another copy by encrypting itself with a different encryption key. Since a different encryption key is used for each infection, the virus body will look different as well. Efficient crypto methods, such as XOR, the key with the virus body, are typically used for this type of virus [2]. Although the virus body looks different from generation to generation, the decryptor that is embedded in the virus remains constant for all generations. As a result, it is possible to detect the virus indirectly by recognizing the code pattern of the decryptor.

2.2.3 Polymorphic Viruses

Polymorphism is a more sophisticated technique that virus writers implement to overcome the weakness of encrypted viruses. Polymorphic viruses hide the decryptor code by mutating it. This makes the decryptor code appear to be different from generation to generation. In addition, polymorphic viruses have the capability of generating a large number of unique decryptors that use different encryption methods to encrypt virus bodies [2]. Therefore, polymorphic viruses lack unique patterns on each infection.

Although a polymorphic virus lacks a unique pattern on each infection, it is still possible to detect the actual virus code through the code emulation technique. To detect polymorphic viruses, anti-virus software incorporates a code emulator that allows the viruses to run within the emulation environment and dynamically detects the decrypted virus bodies.

2.2.4 Metamorphic Viruses

Instead of trying to hide the decryptor, as in polymorphic viruses, virus writers have developed advanced metamorphic techniques to change the actual virus code for each infection [15, 18]. According to Muttik [14], “Metamorphics are bodypolymorphics.” Since the virus body already has different appearances, encryption is no longer needed for hiding the virus. Different generations of a metamorphic virus can have different “shapes” while maintaining the virus’

original behavior. Figure 2 shows the diagrammatical illustration by Szor in [20]. In this section, we will discuss some of the metamorphic techniques employed by metamorphic virus writers. In general, metamorphic virus generators incorporate more than one of these techniques in order to produce highly morphed metamorphic viruses.

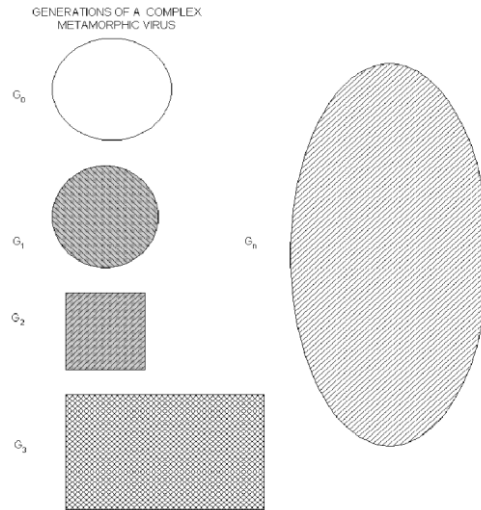


Figure 2. Multiple shapes of a metamorphic virus body [20]

2.2.4.1 Register Swap

Register swap is the simplest metamorphic technique. It mutates the virus body by swapping the operand registers with different registers. For example, instruction “pop edx” might be replaced with “pop eax.” The W95/Regswap virus [7] is among the early metamorphic viruses that use register swap technique. With this technique, the opcode sequence remains unchanged, as shown in Figure 3. Such viruses can usually be detected by a wildcard string [19].

```

a.)
5A          pop     edx
BF04000000 mov     edi,0004h
8BF5       mov     esi,ebp
B80C000000 mov     eax,000Ch
81C2880000 add     edx,0088h
8B1A       mov     ebx,[edx]
899C8618110000 mov    [esi+eax*4+00001118],ebx

b.)
58          pop     eax
BB04000000 mov     ebx,0004h
8BD5       mov     edx,ebp
BF0C000000 mov     edi,000Ch
81C0880000 add     eax,0088h
8B30       mov     esi,[eax]
89B4BA18110000 mov    [edx+edi*4+00001118],esi

```

Figure 3. Two different generations of RegSwap [9]

2.2.4.2 Subroutine Permutation

Subroutine permutation technique changes the appearances of a virus by reordering the virus' subroutines. If a virus has n different subroutines, then it can generate n! different generations without repeating. The W32/Ghost virus [19] is one of the viruses that incorporates the subroutine permutation technique. This particular virus has 10 subroutines. Therefore, it can generate 10! (or 3,628,800) unique copies. However, the virus may still be detected with search strings [19], as the content of each subroutine remains constant.

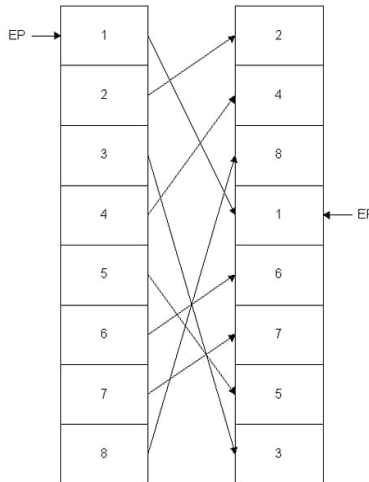


Figure 4. Subroutine permutation

2.2.4.3 Garbage Instruction Insertion

Many complex metamorphic viruses [2] incorporate the garbage instruction insertion technique due to its effectiveness. Garbage instructions are instructions that are either not executed (dead code) or have no effect (do nothing) on program outcomes [13]. By inserting garbage instructions between core instructions randomly, a virus can potentially generate infinite unique copies.

Examples of “do nothing” instructions are “nop,” “add R 0”, or “sub R 0”. [6]. A complete list of “do nothing” instructions can be found in Appendix C. Dead code instructions are usually generated by inserting “jump” instructions to point to the next actual instructions. Any instructions between the “jump” instructions and the next actual instructions will never be executed. The Win95/Zperm virus is one of the viruses that incorporates this technique. As illustrated in Figure 5 [19], the Win95/Zperm family of viruses creates new mutations by reordering core instructions and inserting jump and garbage instructions.

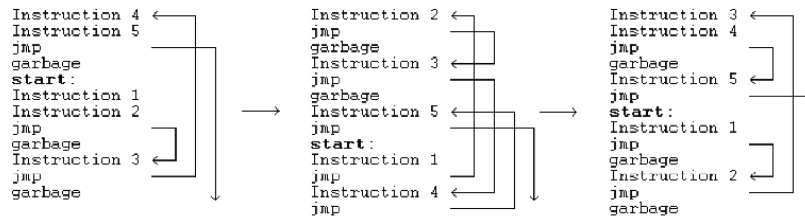


Figure 5. Zperm virus [19].

2.2.4.4 Instruction Substitution

Instruction substitution is another common technique for generating metamorphic viruses. Instruction substitution is the replacement of an instruction or a group of instructions with an equivalent instruction or group [15]. For example, “inc eax” is equivalent to “add eax, 1,” and “move eax, edx” can be replaced by “push edx” followed by “pop eax.” The W32/MetaPhor virus is one of the metamorphic virus generators that incorporates the instruction substitution technique. Some examples of instruction substitution used by the W32/MetaPhor virus [19] are presented in Table 1.

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 1. Examples of instruction substitution used by W32/MetaPhor virus [19]

2.2.4.5 Transposition

Transposition is the reordering of the instruction execution sequence. This can only be done if the affected instructions have no dependency between them. For example, if the second instruction does not depend on the result of the first instruction, then the execution order of these two instructions can be swapped. Consider the following example from [24]:

```
op1 [r1] [, r2]
op2 [r3] [, r4] ; here r1 and/or r3 are to be modified
```

We can swap the above two instructions only if:

1. r1 not equal to r4; and
2. r2 not equal to r3; and
3. r1 not equal to r3.

2.2.5 Formal Grammar Mutation

Formal grammar mutation is the formalization of existing code mutation techniques widely used in viruses (polymorphism and metamorphism) by means of formal grammars and automata [10]. In general, classic metamorphic generators can be presented as bulky, non-deterministic automata, because all possible input characters are specified for each state of automata [10]. By formalizing existing code mutation techniques into formal grammar, one can then apply formal grammar rules to create new viral copies with great variations.

A simple polymorphic decryptor code, as shown in Figure 6, can generate a new viral copy (Figure 7) that looks very different than the original copy.

```

1. mov R1, len
2. mov R2, beg
3. xor [R2], key
4. add R2, 4
5. sub R1, 4
6. jnz step_3

```

Figure 6. Simple polymorphic decryptor

00 PUSH 44554433	00 XOR EDI,EDI
01 POP ESI	01 LEA EDI, [EDI+124]
02 SUB EBX,EBX	02 PUSH 44554433
03 ADD EBX, 124	03 POP ESI
04 XOR [ESI],d20b9a65	04 MOV EDX, [ESI]
05 ADD ESI, 4	05 NOT EDX
06 SUB EBX, 4	06 AND EDX,d75d40bc
07 JZ \$ + 2	07 AND [ESI],28a2bf43
08 JMP \$ + f0	08 OR [ESI],EDX
	09 ADD ESI,4
	10 SUB EDI,4
	11 JZ \$ + 2
	12 JMP \$ + e4

Figure 7. Generated polymorphic decryptor

3. Similarity and the HMM

This section outlines the similarity test and the HMM developed in [2] for detecting metamorphic viruses.

3.1 Similarity Test

Metamorphism is, arguably, the best approach to escape detection. Different generations of a virus must look different in order to avoid detection by signature-based scanning. Some of the virus creation toolkits come with the ability to generate morphed versions of the same virus, even from identical configurations. The similarity test previously studied [2] has shown that it is suitable to measure the effectiveness of a metamorphic virus generator. In this section, we outline the steps of the similarity test and the meaning of its result.

3.1.1 Similarity Test Method

The similarity test employed the method developed by Mishra in [11]. It compares two assembly programs and assigns a quantitative score to represent the percentage of similarity between the two programs. Mishra's method is outlined below and is illustrated graphically in Figure 8.

- 1) Given two assembly programs, X and Y, first extract the sequence of opcodes for each of the programs, excluding comments, blank lines, labels, and other directives. The result is two opcode sequences of length n and m, where n and m are the numbers of opcodes in programs X and Y, respectively. Each opcode is assigned an opcode number: the first opcode is 1, the second is 2, and so on.

- 2) Compare the two opcode sequences by considering all subsequences of three consecutive opcodes from each sequence. Then count as a match any case where all three opcodes are the same in any order. A mark will be placed on a graph coordinate (x, y) of the match, where x is the opcode number of the first opcode of the three-opcode subsequence in program X, and y is the opcode number of the opcode subsequence in program Y.

- 3) After comparing the entire opcode sequences and marking all the match coordinates, a graph plotted on a grid of dimension $n \times m$ is obtained. Opcode numbers of program X are represented on the x-axis and those of program Y are represented on the y-axis. To remove noise and random matches, only those line segments of length greater than some threshold values (e.g., five) will be retained.
- 4) Since the test is basically a sequential match between the two opcode sequences, identical segments of opcodes will form line segments parallel to the main diagonal (if $n = m$, the main diagonal is simply the 45 degree line). If a line segment falls right on the diagonal, the matching opcodes are at identical locations on the two opcode sequences. A line off the diagonal indicates that the matching opcodes appear at different locations in the two files.
- 5) For each axis, the sum of the number of opcodes that are covered by one or more of the matching line segments is calculated. This number is divided by the respective total number of opcodes (n for program X and m for program Y) to give the percentage of opcodes that match some opcodes in the other program. The similarity score for the two programs is the average of these two percentages.

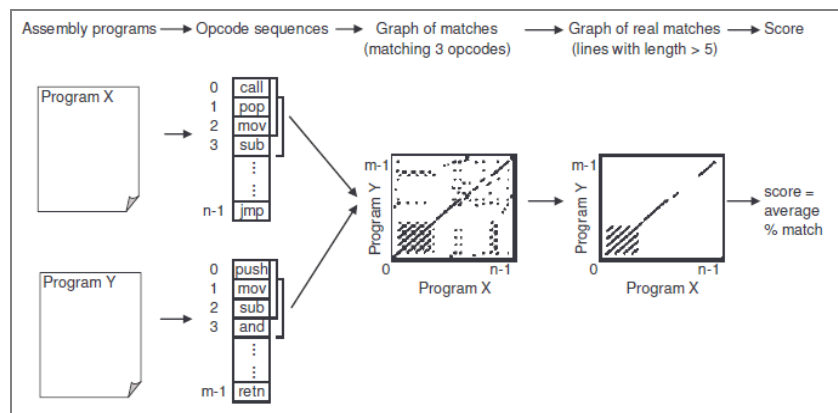


Figure 8. Process of finding the similarity between two assembly programs [2]

3.1.2 Similarity Test Results

Upon the completion of the similarity test, a graph will be generated to visualize the similarity results of the assembly files. Usually, a graph generated by plotting all of the matches for file X and Y (see Figure 9-a) is very populated. This makes it difficult to understand the similarity result. A cleaner graph can be generated by dropping all of the matches that are less than a

specified threshold. Figure 9-b shows a graph that is generated with threshold of 5. The latter provides a clearer visualization of the similarity result. Previous studies in [2] have shown that the best metamorphic engine (NGVCK) achieves a similarity score of about 10%, the lowest similarity score recorded, whereas normal files usually have a similarity score of about 30%.

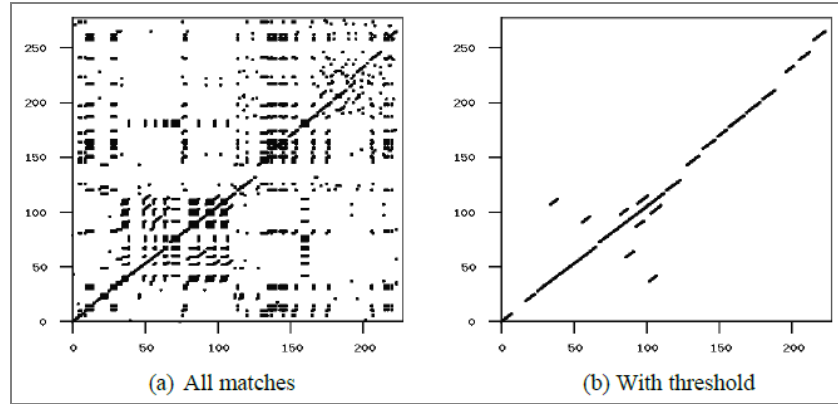


Figure 9. Similarity graph [2]

3.2 HMM

The Hidden Markov Model (HMM) is a statistical pattern analysis algorithm. The notations used in the HMM are as follows:

T = Length of the observed sequence

N = Number of states in the model

M = number of distinct observation symbols

O = Observation sequence $\{O_0, O_1, \dots, O_{T-1}\}$

A = State transition probability matrix

B = Observation probability distribution matrix

π = Initial state distribution matrix

A generic Hidden Markov Model is illustrated in Figure 10. The state and observation at time *t* are represented by X_t and O_t respectively. The Markov process—which is hidden behind the dashed line—is determined by the initial state X_0 and the *A* matrix. Only the O_t is observable, which is related to the actual states of the Markov process by the matrices *B* and *A*.

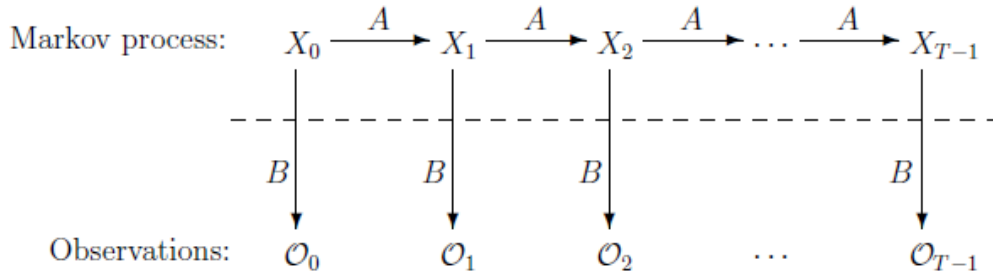


Figure 10. Generic Hidden Markov Model [14]

HMMs are widely used for protein modeling and speech recognition applications [3]. In general, an HMM first creates a training model that represents the input data (training data). The training model contains a list of unique symbols observed in the input data and their positions in the input sequence. This model will be used by the HMM to determine if a given new input sequence has a pattern similar to that of the model.

Recently, HMMs have been successfully used to detect metamorphic viruses [2, 9]. Although metamorphic engines use various code obfuscation techniques to change the appearance of viral copies, some similar patterns exist within the same family of viruses. An HMM collects the input data from all known viruses and builds the training models (one for each family virus) based on these input data. Subsequently, any file can be tested against these models to determine if it belongs to one of them. If an input file belongs to a model, then it is a member of the virus family that the model represents.

3.2.1 HMM Example

A simple example in [14] illustrates the inner working of an HMM. Suppose that one has no prior knowledge of the average annual temperature for any given year and wants to determine this information based on the observation of tree sizes (S-small, M-medium, L-large). To keep the example simple, let us assume that the annual temperature can be either hot (H) or cold (C). In addition, we know the probability of the annual temperature trend: a hot year followed by another hot year (HH) is 0.7; a hot year followed by a cold year (HC) is 0.3; a cold year followed by a hot year is 0.4; and a cold year followed by another cold year is 0.6. Figure 11 shows the matrix representation of these probabilities.

$$\begin{array}{c}
 \\
 H \\
 C
 \end{array}
 \begin{array}{cc}
 H & C \\
 \left[\begin{array}{cc}
 0.7 & 0.3 \\
 0.4 & 0.6
 \end{array} \right]
 \end{array}$$

Figure 11. Temperature transition probability

Furthermore, we know that the correlation between tree sizes and temperature is as follows:

—In a hot year, the probability of a tree being small is 0.1, being medium 0.4, and being large 0.5.

—In a cold year, the probability of a tree being small is 0.7, being medium 0.2, and being large 0.1.

Figure 12 shows the correlation probability between temperatures and tree sizes in a matrix representation.

$$\begin{array}{c}
 \\
 H \\
 C
 \end{array}
 \begin{array}{ccc}
 S & M & L \\
 \left[\begin{array}{ccc}
 0.1 & 0.4 & 0.5 \\
 0.7 & 0.2 & 0.1
 \end{array} \right]
 \end{array}$$

Figure 12. Tree size probability

In this example, the annual temperatures are the states, and the tree sizes are the observable symbols. The probability of different tree sizes at each temperature represents the probability of the observation symbols in each state. The states (H and C) are hidden, since the temperature cannot be seen directly. We can only see the observation symbols (S, M, and L), which are statistically related to the states. With the knowledge of correlation probabilities for annual temperature and tree sizes, we can build an HMM model as shown in Figure 13.

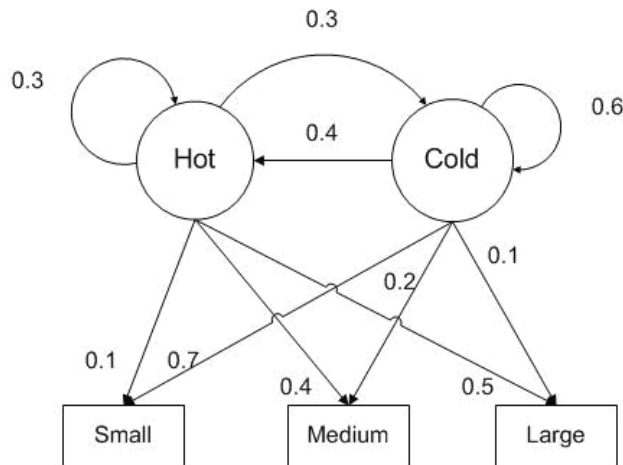


Figure 13. HMM model [14]

Suppose we have observed that the tree sizes (observation symbols) sequence for four consecutive years is (S, M, S, L). We want to use this observed sequence to find the annual temperature sequence (states).

To solve this problem with an HMM algorithm, we must first construct our HMM parameters as follows:

—State transition probability matrix

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$$

—The observation probability distribution matrix

$$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

—Number of states in the model $N = 2$ (hot and cold)

—Number of distinct observation symbols $M = 3$ (small, medium, and large)

—Given initial state distribution matrix

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$$

The HMM steps used to determine the state transition for a given observation (S, M, S, L) of length $T = 4$ will be as follows:

1. Determine all possible state transitions = N^T .
2. Calculate the probability of a given observation sequence for each state transition of step 1 (shown in Table 2). For example, calculate the probability of sequence HHCC as follows:

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

3. The annual temperature sequence is the one with the highest probability. In this case, the answer would be “CCCH,” since it has the highest probability.

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 2. Probabilities of observing (S, M, S, L) for all possible state sequences [14]

The above brute-force method of computing HMM results requires an exponential amount of work. This is generally infeasible. The beauty an HMM is that it includes efficient algorithms to solve the three problems in which we are interested. These are [14]:

1. Given the model $\lambda = (A, B, \pi)$ and an observation sequence O , find $P(O | \lambda)$, which is the likelihood of observing the sequence O given the model.
2. Given the model $\lambda = (A, B, \pi)$, find an optimal state sequence for the underlying Markov process. That is, uncover the hidden part of the HMM.

3. Given an observation sequence O , the number of symbols M , and the number of states N , find the model $\lambda = (A, B, \pi)$ that maximizes the probability of O . In other words, the model is trained to best fit the observed data.

The fact that there are efficient algorithms for solving the three HMM problems provides a fundamental building block for constructing the HMM-based virus detector. More precisely, the HMM-based virus detector developed in [2] was implemented the following algorithms to solve the three HMM problems:

1. The Forward algorithm:

The equation to find the likelihood of an observed sequence is given as

$$P(O, X | \lambda) = \pi_{x_0} b_{x_0}(O_0) a_{x_0, x_1} b_{x_1}(O_1) a_{x_1, x_2} \dots a_{x_{T-2}, x_{T-1}} b_{x_{T-1}}(O_{T-1}).$$

The naïve approach to solve the above equation would be to generate all possible state sequences X_i of length T and sum over the probabilities $P(O, X_i | \lambda)$:

$$\begin{aligned} P(O | \lambda) &= \sum_{X_i} P(O, X_i | \lambda) \\ &= \sum_{X_i} \pi_{x_0} b_{x_0}(O_0) a_{x_0, x_1} b_{x_1}(O_1) a_{x_1, x_2} \dots a_{x_{T-2}, x_{T-1}} b_{x_{T-1}}(O_{T-1}) \end{aligned}$$

However, this direct computation requires $2TN^T$ computations.

The forward algorithm (sometimes called the α -pass), which requires only N^2T computations, is a much more efficient algorithm for solving the above equation. Instead of computing all possible states directly, the forward algorithm inductively computes the states as follows:

For $t = 0, 1, \dots, T-1$ and $i = 0, 1, \dots, N-1$, define a forward variable

$$\alpha_t(i) = P(O_0, O_1, \dots, O_t, x_t = q_i | \lambda)$$

which denotes the probability of observing the partial sequence (O_0, O_1, \dots, O_t) up to time t and being in state q_i at time t . The forward variables can then be computed recursively using the following steps:

Step 1. Initialization

$$\alpha_0(i) = \pi_i b_i(O_0), \quad \text{for } i = 0, 1, \dots, N-1$$

Step 2. Compute forward variables inductively

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right] b_i(O_t), \quad \text{for } t = 1, 2, \dots, T-1 \text{ and } i = 0, 1, \dots, N-1.$$

The probability of $P(O | \lambda)$ can then be calculated as

$$\begin{aligned} P(O | \lambda) &= \sum_{i=0}^{N-1} P(O_0, O_1, \dots, O_T, x_T = q_i | \lambda) \\ &= \sum_{i=0}^{N-1} \alpha_{T-1}(i). \end{aligned}$$

2. The Viterbi algorithm:

The Viterbi algorithm finds an optimal state sequence by finding a highest scoring overall path X^* that maximizes the probability $P(O, X | \lambda)$ as follows:

For $t = 0, 1, \dots, T-1$ and $i = 0, 1, \dots, N-1$, let $\delta_t(i)$ denote the probability of the most probable state path (x_0, x_1, \dots, x_t) that generates the partial sequence (O_0, O_1, \dots, O_t) up to time t and ending in state q_i ,

$$\delta_t(i) = \max_{x_0 \dots x_{t-1}} P(O_0, O_1, \dots, O_t, x_0, x_1, \dots, x_{t-1}, x_t = q_i | \lambda)$$

To find $\delta_t(i)$ recursively:

$$\begin{aligned} \delta_t(i) &= \max_{0 \leq j \leq N-1} [\delta_{t-1}(j) a_{ji}] b_i(O_t), \quad \text{for } t = 1, 2, \dots, T-1 \text{ and } i = 0, 1, \dots, N-1. \\ \delta_0(i) &= \pi_i b_i(O_0), \quad \text{for } i = 0, 1, \dots, N-1 \end{aligned}$$

Then the most likely state sequence P^* is computed as:

$$P^* = \max_{0 \leq i \leq N-1} [\delta_{T-1}(i)]$$

3. The Baum-Welch algorithm:

This algorithm provides an efficient method for adjusting the model parameters to best fit the observations. The sizes of the matrices (N and M) are fixed, but the elements of A, B, and π are free, subject only to the row stochastic condition. The process to re-estimate the model, which is one of the most amazing aspects of HMMs, is as follows:

- a. Initialize $\lambda = (A, B, \pi)$ with a best guess. If a best guess is not available, random values such that $\pi_i \approx 1/N$, $a_{ij} \approx 1/N$ and $b_j(k) \approx 1/M$ can be used.

- b. Compute $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i, j)$ and $\gamma_t(i)$ using the following equations:

For $t = 0, 1, \dots, T - 2$ and $i, j \in \{0, 1, \dots, N - 1\}$, define “di-gammas” as

$$\gamma_t(i, j) = P(x_t = q_i, x_{t+1} = q_j | \mathcal{O}, \lambda).$$

The di-gammas can be written in terms of α , β , A , and B as

$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)}{P(\mathcal{O} | \lambda)}.$$

The $\gamma_t(i)$ and $\gamma_t(i, j)$ are related by:

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i, j).$$

- c. Re-estimate the model $\lambda = (A, B, \pi)$ as follows:

For $i = 0, 1, \dots, N - 1$, let

$$\pi_i = \gamma_0(i)$$

For $i = 0, 1, \dots, N - 1$ and $j = 0, 1, \dots, N - 1$, compute

$$a_{ij} = \frac{\sum_{t=0}^{T-2} \gamma_t(i, j)}{\sum_{t=0}^{T-2} \gamma_t(i)}.$$

For $j = 0, 1, \dots, N - 1$ and $k = 0, 1, \dots, M - 1$, compute

$$b_j(k) = \frac{\sum_{\substack{t \in \{0, 1, \dots, T-2\} \\ \mathcal{O}_t = k}} \gamma_t(j)}{\sum_{t=0}^{T-2} \gamma_t(j)}.$$

- d. If $P(\mathcal{O} | \lambda)$ increases, go to step b.

3.2.2 HMM as a Virus Detection Tool

HMM as virus detection tool requires training data to produce a model. The goal is to train one or more HMMs to represent the statistical properties of the virus family. These trained models can then be used to determine whether a given program is similar to the

viruses in the training set. Each model is trained by collecting training data from files generated by the same generator. This also means that the resulting model is specific to the generator from which the training data originate.

To produce a training model, a set of virus files generated by the same generator must first be converted to assembly files using IDA Pro [22]. Unique assembly opcodes found in these files will constitute the HMM symbols. A long observation sequence is formed by concatenating all of the virus files within the same family. Given unique symbols and a unique observation sequence, an HMM training model can then be constructed using the solution of the third HMM problem discussed above. For example, given training data as shown in Figure 14, an HMM model can then be constructed as shown in Figure 15.

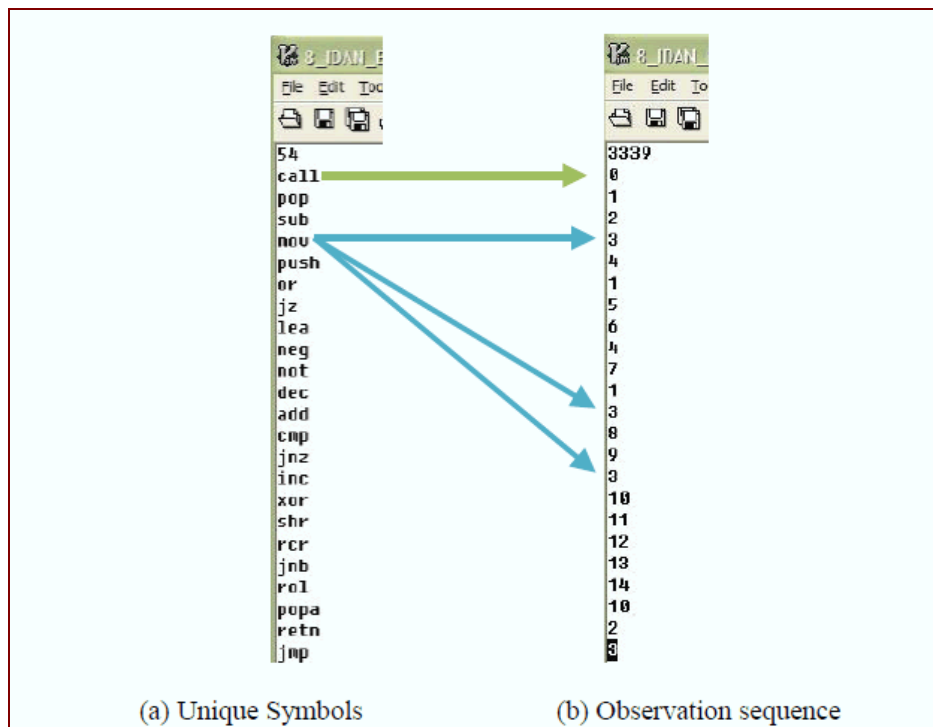


Figure 14. Training data example [5]

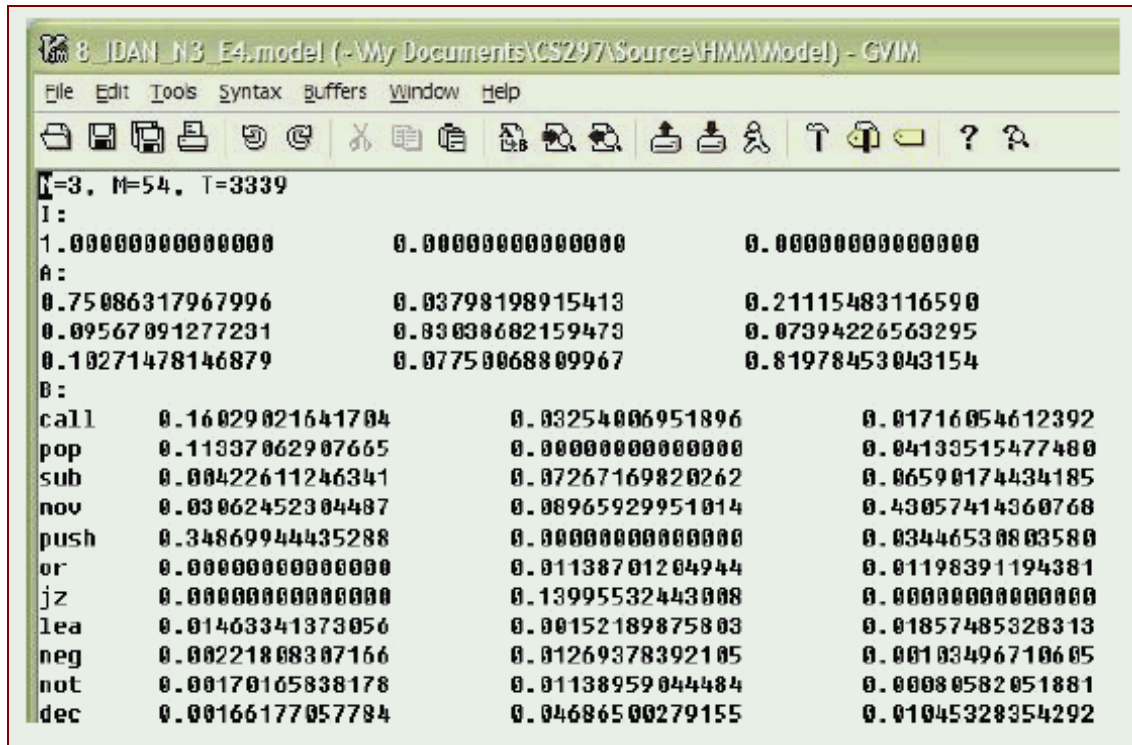
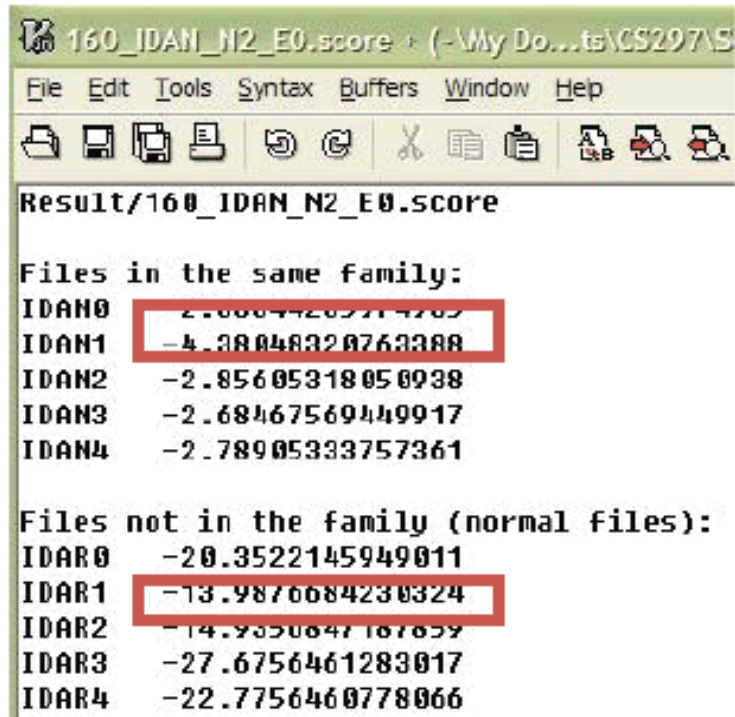


Figure 15. HMM model example [5]

After the HMM model is produced for a virus family, it can then be used to test any file to determine if it belongs to the same family. If a file has a score greater than a certain threshold, then it belongs to the same family. Otherwise, it is not in this family. An example of HMM output is shown in Figure 16. With this particular example, IDAN0 to IDAN4 are in the same virus family as the HMM model. IDAR0 to IDAR4 are not in the same virus family.



The image shows a Notepad window titled "160_IDAN_N2_E0.score" with a menu bar (File, Edit, Tools, Syntax, Buffers, Window, Help) and a toolbar. The text content is as follows:

```
Result/160_IDAN_N2_E0.score

Files in the same family:
IDAN0 2.88644289714769
IDAN1 -4.38048320763388
IDAN2 -2.85605318050938
IDAN3 -2.68467569449917
IDAN4 -2.78905333757361

Files not in the family (normal files):
IDAR0 -20.3522145949011
IDAR1 -13.9876684230324
IDAR2 -14.9350647167659
IDAR3 -27.6756461283017
IDAR4 -22.7756460778066
```

In the original image, the values for IDAN1 and IDAR1 are highlighted with red boxes.

Figure 16. HMM result example [5]

4. Implementation

4.1 Introduction

In order to produce viral copies that are hard to detect, a metamorphic engine needs to implement many code obfuscation techniques. Each implementation may have its own process to decide when and how to apply the techniques.

Even though a metamorphic engine implements all of the code obfuscation techniques, the HMM detector developed in [2] is still able to recognize the generated viruses and classify them into the same virus family. An unsuccessful attempt to escape from the HMM-based detector created in [5] has also shown that the HMM is very effective in detecting highly morphed viruses. In this project, we will develop another metamorphic engine to try to evade the HMM virus detector.

4.2 Goal

Our implementation was geared toward achieving the following goals:

- Generate morphed copies of a single input virus. These morphed copies should have a similarity of approximately 30% (match scores of normal files) with the base virus and among themselves.
- The morphed copies should have the same functionality as the base virus.
- The morphed copies should be “close” to the normal programs. The assumption here is that the normal programs are the cygwin utility files of the same size as the base virus. The reason for using cygwin utility files is that they are probably engaged in the same low-level operations as viruses. A morphed virus is “close” to a normal program if its statistics, such as its opcode counts and opcode sequences, are more like normal files than those of an un-morphed virus. The notion of “close” will be discussed in greater detail in the next section.
- The metamorphic engine should work on any assembly program.
- Generated viral copies should be capable of escaping HMM detection.

4.3 Code Obfuscation Techniques Used

Our metamorphic engine leverages the code obfuscation techniques implemented in previous work [5]. However, viruses generated by the engine developed in [5] are still detectable by the HMM detector. We have analyzed the previous engine's failure. We suspect that the reason for the failure is due to the fact that the engine applies code obfuscation techniques randomly. This randomness does not make the virus more like a normal program. Therefore, in our engine, we will apply the result of a code obfuscation operation only if it makes the virus more like a normal program. A scoring algorithm, namely the Dynamic Scoring Algorithm, has been developed for comparing the resulting virus code against the normal program code after each morph operation.

4.3.1 Dynamic Scoring Algorithm

To make a virus similar to a normal file, we developed an algorithm to calculate the score of similarity between the two files. The lower the score, the better the match. Since this algorithm will need to run each time we try to change an instruction, it must be as efficient as possible. The Dynamic Scoring Algorithm developed in this project need not compute the entire file each time. Instead, it only needs to compute the modified opcodes each time.

4.3.1.1 Algorithm Initialization

To initialize the dynamic scoring algorithm, two files will be passed into it as parameters. The first one is a virus, and the second one is a normal file.

Algorithm initialization will generate four master lists: individual opcode counts of the virus file; opcode-pair counts of the virus file; individual opcode counts of the normal file; and opcode-pair counts of the normal file. For example, given two short files with five opcodes, as shown in Table 3, the initializing of the algorithm will generate four lists, as shown in Table 4.

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

Retn	Retn
------	------

Table 3. Opcode sequences of virus file and normal files

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(1)	Add_mov(0)	1
Pop (1)	Pop(1)	0	Mov_pop(1)	Mov_pop(0)	1
Retn(1)	Retn(1)	0	Pop_retn(1)	Pop_retn(1)	0
Sub (0)	Sub (1)	1	Mov_mov(0)	Mov_mov(1)	1
			Mov_sub(0)	Mov_sub(1)	1
			Sub_pop(0)	Sub_pop(1)	1

Table 4. Opcode and opcode-pair counts lists

We also compute the initial score by summing the difference of each opcode and opcode-pair counts between the two files. In the above example, the initial score will be computed as 8.

4.3.1.2 Scoring the Changes

To check if a change will yield a better score, we only need to compute the score change by the opcode sequence change. It will take the old sequence and the new sequence as input and generate a score. A score less than 0 means the new sequence makes the two files closer to each other. A score greater than 0 means the new sequence makes the two files less similar to each other. A score of 0 means no change. This method only computes the score and does not make any changes to the master lists. For example, when we transpose “add, mov” to “mov, add,” we will pass the original subsequence and the new subsequence that includes one opcode before and one opcode after plus the change itself. In this example, the two subsequences pass to the scoring changes method will be “mov, add, mov, pop” (original subsequence) and “mov, mov, add, pop” (new subsequence).

We then compute the changes in scores as follows:

1. Compute and save the to-be-affected counts.
2. Subtract the counts of the original subsequence from the master lists.
3. Add the counts of the new subsequence to the master lists.
4. Compute the affected counts against the normal file.

Table 5 shows the result of computing the original to-be-affected score (5 in this case).

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)	Mov (2)	0	Mov_add (1)	Mov_add (0)	1
Add (1)	Add (0)	1	Add_mov(1)	Add_mov(0)	1
Pop (1)	Pop(1)	0	Mov_pop(1)	Mov_pop(0)	1
			Mov_mov(0)	Mov_mov(1)	1

Table 5. Saved original subsequence score

Table 6 shows the subtraction and addition of the original subsequence and new subsequence. The final subsequence counts and the relative normal file counts are show in Table 5. The new score is also reflected as a difference in Table 7. Note that the “Add_pop” is a new counter.

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

Table 6. Subtract from old count and add new count

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)	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
			Mov_mov(1)	Mov_mov(1)	0
			Add_pop(1)	Add_pop(0)	1

Table 7. New score after changes

As shown in Table 7, the new score of the affected subsequence is 3, and the original score is 5 (shown in Table 5). This tells us that if we do such transposition, we will make the virus file closer to the normal file by 2 points.

4.3.1.3 Updating the Changes

This method is similar to the scoring-the-changes method except it will make permanent changes to the master lists.

For the transpose change, as shown in the previous section, the master score will be decreased from 8 to 6 (since we improved the score by 2). The master lists of the virus file will be updated, as highlighted in Table 8.

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 8. Updated opcode count lists

4.3.2 Dead Code Insertion

Instructions in our base virus are statistically different than normal programs. Previous work [5] has analyzed the statistics of virus instructions vs. normal program instructions. Their statistics are shown below in Figure 17 and Figure 18.

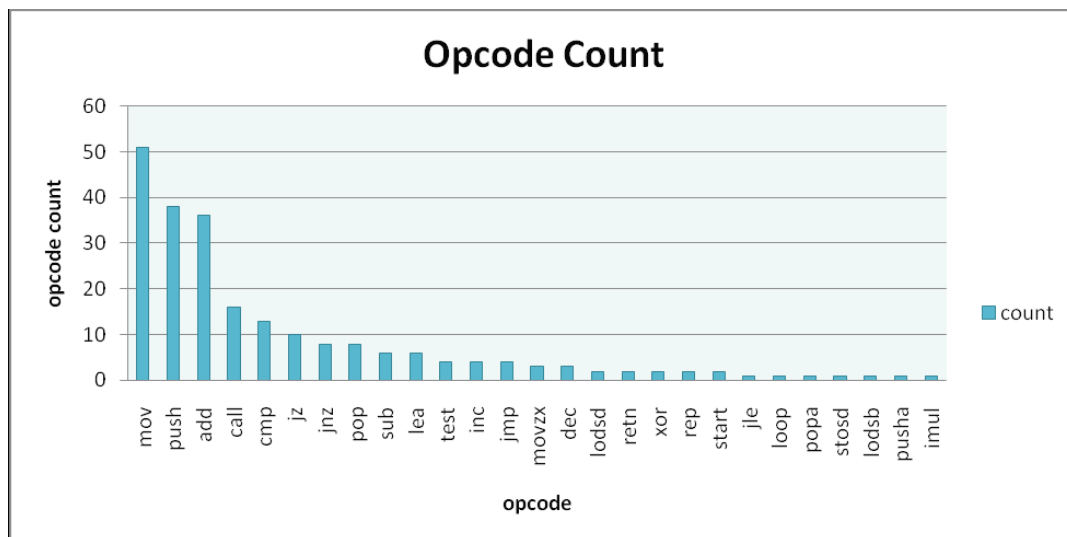


Figure 17. Base virus opcodes and their frequencies [5]

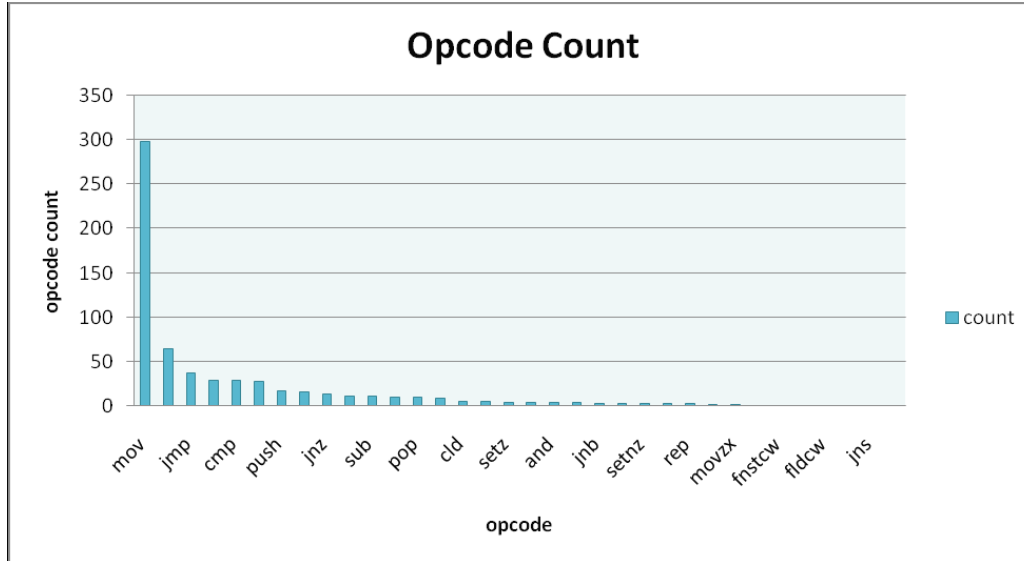


Figure 18. Opcodes of normal files and their frequencies [5]

In order to make our base virus more like a normal program, we will insert some dead code into the generated viruses. However, the set of dead codes that we built into our engine is finite. Applying these dead codes over and over will result in the HMM detector recognizing them. Therefore, we will also generate “dead codes” on the fly by copying blocks of instructions and subroutines from the normal program. Since the dynamically generated “dead codes” are “real codes” in the normal program, inserting them into our viruses will make our viruses look more like normal programs.

4.3.2.1 Inserting Dead Code

Our engine leveraged the build-in dead code library from the work done in [5]. However, instead of randomly inserting dead codes, we will try to insert some dead code before and/or after each instruction in a virus file only if that makes our virus more like a normal program. Our build-in dead code insertion algorithm is shown in Figure 19.

For each instruction

1. Insert dead code before and/or after it.
2. Compute a score using the Dynamic Scoring Algorithm.
3. If the score is better or remains the same, keep the changes and update the Dynamic Scoring Algorithm counters.

4. If the score worsens, discard the change made in step 1.

Figure 19. Dead code insertion algorithm

4.3.2.1 Inserting Dead Code Generated from a Normal File

Since the build-in dead code library only included a limited set of instruction combinations, applying them into many virus copies will provide a pattern for HMM to detect the generated viruses. In order to overcome this limitation, we added logics into our engine to copy sets of instructions from normal files. Theoretically, this implementation will allow us to generate infinite sets of dead codes, since there is an infinite number of normal files.

The instruction sets that we copied from a normal file can function as a block of five or more continuous instructions, or as a complete subroutine. If we are copying a block of continuous instructions, we will insert an unconditional “jmp” instruction before the block so that these instructions will not be executed. In addition to inserting a “jmp” instruction, we might also need to modify the operands of some instructions so that the generated virus assembly file can be re-compiled by FASM [21]. For example, if an instruction contains a label that is only valid in the normal file, then we will need to replace that label with a label that is valid in the generated virus file.

When copying a subroutine, we also need to modify the operands of some instructions. However, we do not need to insert jump instructions. The copied subroutine will be placed between two subroutines in the virus file. Since the copied subroutine never gets called, it will not impact the original behaviors of the virus.

4.3.3 Equivalent Instruction Substitution

Some opcodes, such as mov, push, and add instructions, appear more frequently in the base virus [5]. To make the generated virus’ opcode count statistically closer to normal programs, we

substitute equivalent instructions in place of these instructions. For example, “add instruction” can be replaced with “sub,” “lea,” or “not,” followed by “neg” instructions, as shown in Table 9.

add R, imm	1. sub R, new_imm where new_imm = imm x (- 1) 2. lea R, [R + imm]
add R, 1	1. not R neg R

Table 9. Equivalent instructions for add [5]

While substituting equivalent instructions, we also keep track of the score changes between virus copies and normal files. We only perform substitution if the score improves. Our algorithm for equivalent instruction substitution is shown in Figure 20 below:

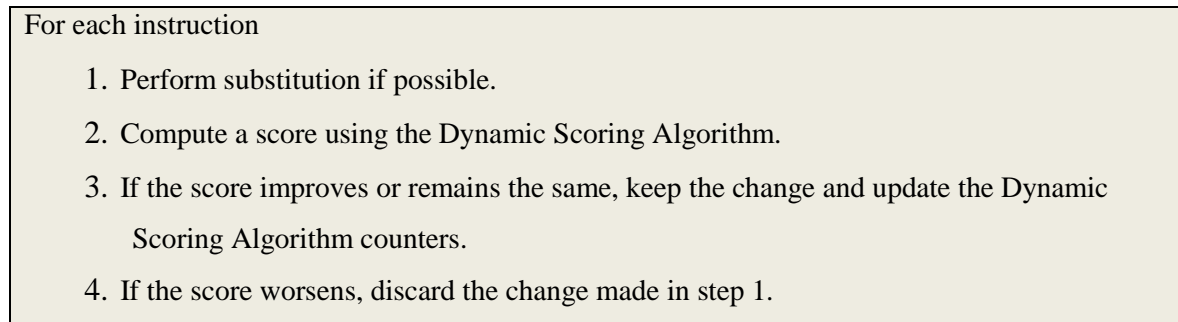
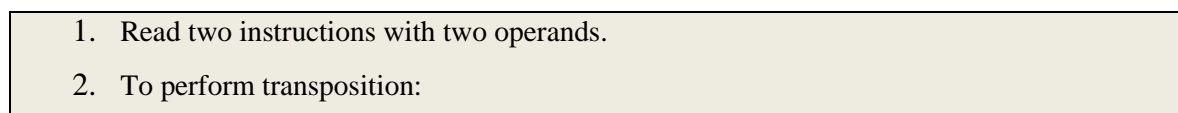


Figure 20. Equivalent instruction substitution algorithm

4.3.4 Transposition

After generating dead code and performing equivalent instruction substitution, we perform transposition to make our virus even closer to a normal program. The transposition algorithm in [5] performs transposition randomly with a probability of 25%. We removed the randomized nature of that algorithm. Instead, we used the Dynamic Scoring Algorithm to perform transposition in order to make our virus closer to a normal program. Therefore, our final transposition algorithm is as shown in Figure 21:



- a. Read third instruction
- b. If the third instruction is not any conditional jump instruction then
 - i. If the to-operands of both instructions are not equal
And
If the to-operand of the first instruction is not equal to the from-operand of the second instruction
and
If the from-operand of the first instruction is not equal to the to-operand of the second instruction
then
Swap the two instructions.
3. Compute the score of the transposition.
4. If the score improves, keep the change. Otherwise, discard the change.

Figure 21. Transposition algorithm

4.4 Metamorphic Engine Algorithm

The code obfuscation techniques described in section 4.3 are implemented as individual modules in our engine. We execute each module in sequence to generate our final virus copy. The overall engine algorithm is shown in Figure 22 below:

1. Read in a base virus and a normal file.
2. Initialize the Dynamic Scoring Algorithm.
3. Insert dead code between each instruction if it makes the virus “closer” to the normal file.
4. Perform equivalent instruction substitution for each instruction if it makes the virus closer to the normal file.
5. Perform transposition for every instruction pair if it makes the virus closer to a normal file.
6. For each instruction, generate a random number between 0 and 99:
 - a. If random number < configured percentage for junk block, copy a junk block from normal file.

7. Between each subroutine, generate a random number between 0 and 99.
 - a. If random number < configured percentage for junk function, copy a subroutine from normal file.

Figure 22. Metamorphic engine algorithm

5. Experiments

We used the similarity test and HMM test tools developed in [2] to perform our test. After we successfully demonstrated that our engine was able to escape HMM-based detection, we repeated our test with different engine settings (i.e., reduced the number of dead code copied from the normal file to find the threshold at which the HMM detector began to fail).

5.1 Base Virus

To test our engine, we first used NGVCK to generate 200 virus files of the same family. These 200 virus files served as our base viruses. We then constructed 40 normal files from cygwin utility files.

After we generated our base viruses and normal files, we used the HMM detector to verify that viruses generated by NGVCK were still detectable. The procedures we executed with the HMM detector are the same as those in [2]. We first generated the HMM model with 160 viruses. We then generated the scores for the remaining 40 viruses against the HMM model. We also generated the scores for the 40 normal files against the same HMM model.

If none of the normal files score higher than the viruses, then the HMM detector will be able to detect the family viruses, since there is a threshold that it could use to determine whether a given file is a virus (score higher than the threshold) or a normal file (score lower than the threshold). On the other hand, if some normal files score higher than some of the virus files, then the HMM detector will not have a good threshold to determine a given file. This means that some viruses will escape HMM-based detection.

Figure 23 shows the result of base viruses against normal files. All of the normal files score lower than virus files. Therefore, the base viruses we generated from NGVCK are detectable by the HMM detector.

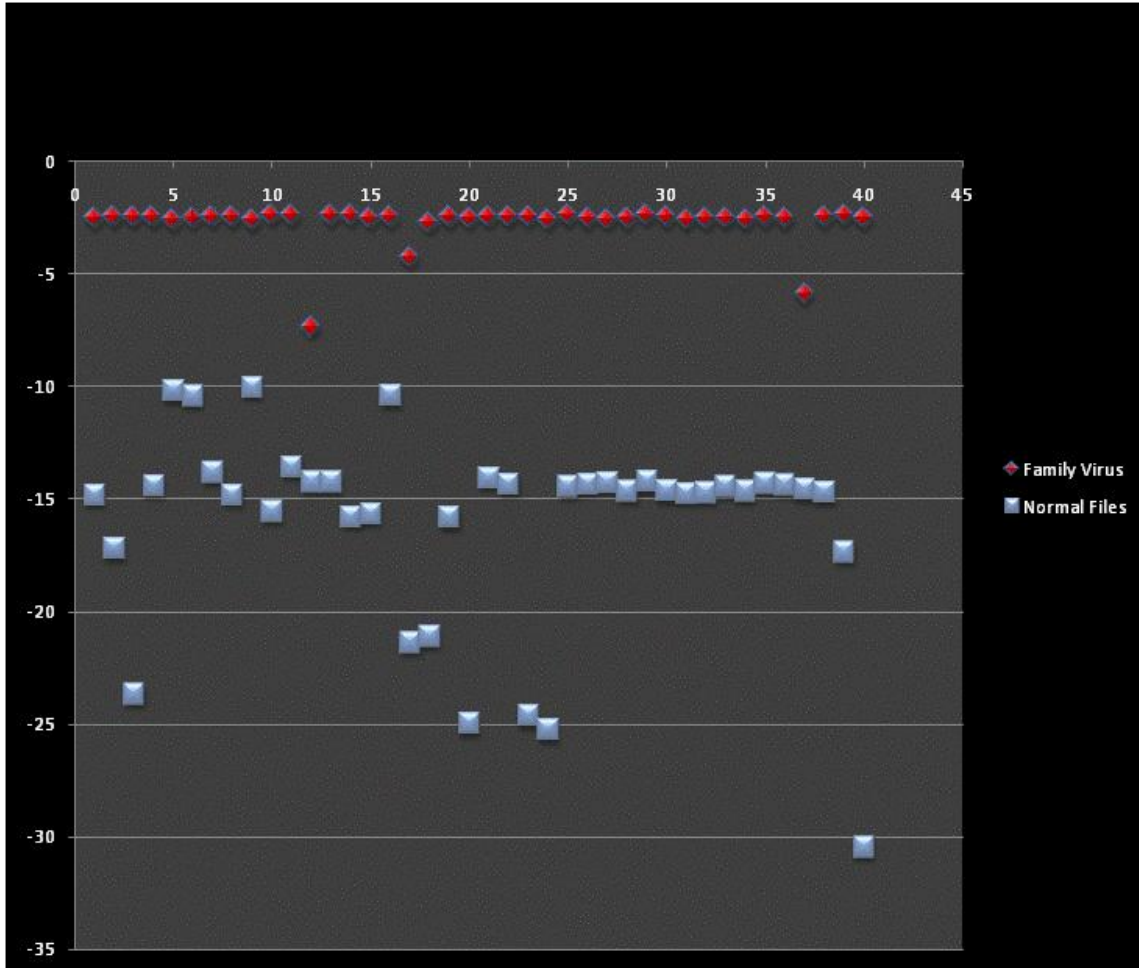


Figure 23. HMM results for base viruses generated by NGVCK

After we generated base viruses, we used our engine to perform additional code obfuscation. Our engine will take one base virus and one normal file as inputs. It will apply additional code obfuscation techniques to the base virus so that the generated virus copy will be statistically closer to a normal file. In our experiment, we will make five virus copies closer to one normal file. For example, virus copies 1, 41, 81, 121, and 161 will look like normal file 1. Virus copies 2, 42, 82, 122, and 162 will look like normal file 2.

5.2 Similarity Test

The similarity test compares and reports the percentage of similarity among two assembly programs [5]. Since our engine will make a virus file closer to a normal file, we will compare the similarity of a virus file with its peer normal file as we increase the percent of dead code copied from the normal file.

We first compared a base virus against a normal file, and there was no similarity between them at all. Then we ran the two files through our engine without any dead code copying configured. We were able to obtain a similarity score of 13.8% between the two files. After that, we copied dead code from the normal file into the virus and computed the similarity score again. The more dead code we copied from the normal file into the virus, the higher the similarity score, which is what we expected. Table 10 shows the table format of the similarity scores as we increased the file sizes by copying more dead codes.

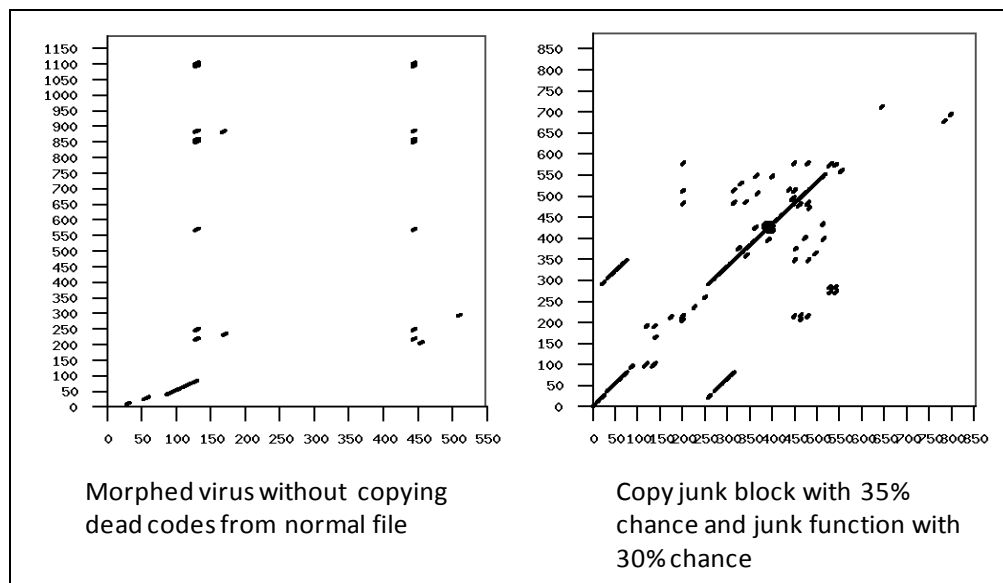


Figure 24. Similarity score for morphed virus against normal files

	File size	Similarity Score
Base virus	17K	0
Morphed virus without copying dead codes from normal file	21.8K	13.8%

Copy junk block with 10% chance	23.2K	15.8%
Copy junk block with 25% chance	23.8K	16.2%
Copy junk block with 35% chance	24.3K	15.0%
Copy junk block with 35% chance and junk function with 5% chance	25.8K	16.5%
Copy junk block with 35% chance and junk function with 15% chance	26.3K	16.6%
Copy junk block with 35% chance and junk function with 20% chance	28.5K	17.2%
Copy junk block with 35% chance and junk function with 30% chance	28.6K	18.1%

Table 10. Similarity score of virus and its peer normal file

Since dead code blocks copied from the normal file were of different sizes, we will use the file size for the x-axis of our graphs to show our experiment results for the rest of this report. Figure 25 shows the similarity scores for the generated viruses against normal files. As more dead code was copied from the normal file, the virus file size increased. The similarity score between the virus and normal file also increased as expected.

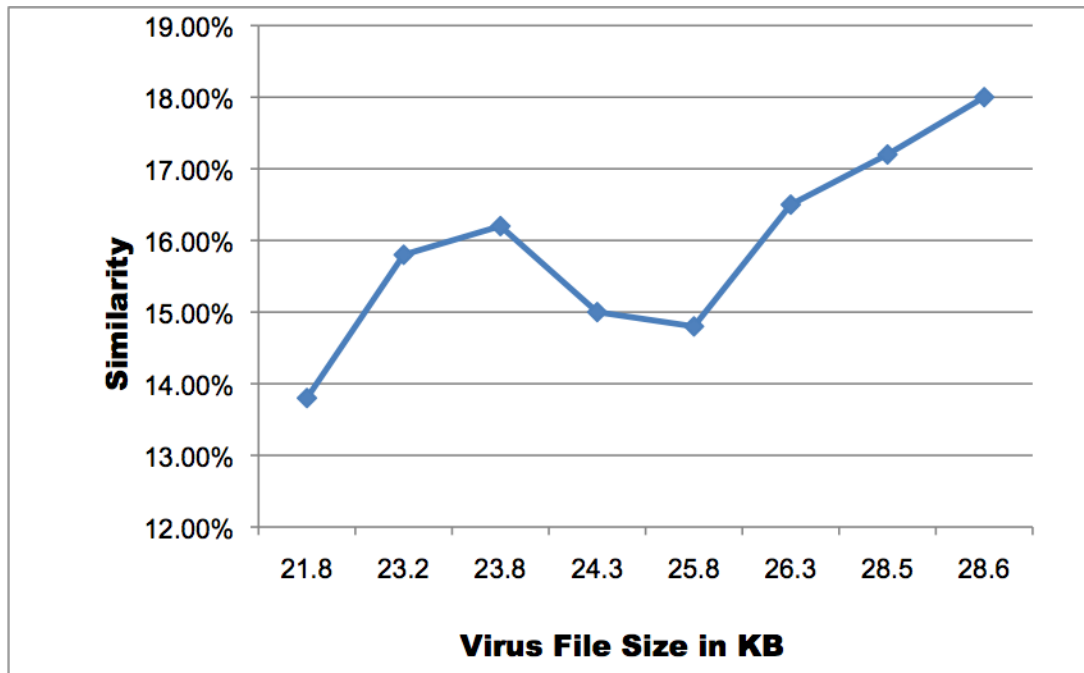


Figure 25. Similarity score of virus and normal file

5.3 HMM

As expected, similarity tests show that copying more dead code from a normal file into a virus makes the virus closer to the normal file. We will perform HMM tests for each set of viruses that we generate. The idea is to see how much dead code we need to copy from normal files in order to make our viruses undetectable.

We executed HMM tests on our viruses with a number of hidden states from 2 to 5. However, based on the previous work [2, 5] and our results, it appears that the number of hidden states will not affect the results. Therefore, we will focus on analyzing the results for the HMM tests with three hidden states. We will include some results with different numbers of hidden states in appendix A.

For all our tests, we constructed the HMM model from 160 virus files. Then, we compared the remaining 40 virus files against the HMM model. We also compared the 40 normal files against the HMM model.

5.3.1 Zero Percent Dead Code

This set of viruses was generated by applying our engine without any dead code copying from normal files. With this setting, the average file size increase from 17KB to 21.8KB. The similarity score also increase from 0% to 13.8%.

0% dead code with N=3							
Family Viruses			Normal Files				
N120	-3.82742	N140	-2.64343	IDAR0	-8.62106	IDAR20	-12.8004
N121	-2.60846	N141	-2.78598	IDAR1	-5.98894	IDAR21	-8.56907
N122	-2.74388	N142	-3.07189	IDAR2	-13.8067	IDAR22	-15.1685
N123	-2.93378	N143	-2.74519	IDAR3	-18.3758	IDAR23	-16.5805
N124	-2.74935	N144	-2.70781	IDAR4	-4.98846	IDAR24	-8.55813
N125	-2.83254	N145	-2.71075	IDAR5	-9.09981	IDAR25	-8.41451
N126	-2.649	N146	-2.6337	IDAR6	-12.527	IDAR26	-8.22832
N127	-2.76175	N147	-2.77156	IDAR7	-12.4914	IDAR27	-8.6588
N128	-2.7331	N148	-2.68296	IDAR8	-8.16147	IDAR28	-8.21336
N129	-3.01979	N149	-3.92464	IDAR9	-9.20784	IDAR29	-8.49555
N130	-4.03604	N150	-2.7027	IDAR10	-10.5268	IDAR30	-10.7358
N131	-2.57982	N151	-2.75938	IDAR11	-6.27038	IDAR31	-8.62145

N132	-2.75932	N152	-2.64362	IDAR12	-6.27038	IDAR32	-8.18047
N133	-2.75863	N153	-4.08823	IDAR13	-14.3691	IDAR33	-9.47778
N134	-2.57266	N154	-2.49181	IDAR14	-11.6093	IDAR34	-8.41929
N135	-2.73927	N155	-2.81595	IDAR15	-8.44406	IDAR35	-8.75353
N136	-2.63524	N156	-2.79971	IDAR16	-16.2526	IDAR36	-8.53441
N137	-2.72739	N157	-2.73247	IDAR17	-11.8591	IDAR37	-11.3747
N138	-3.91962	N158	-2.61171	IDAR18	-12.5659	IDAR38	-11.1086
N139	-2.91821	N159	-2.62985	IDAR19	-15.9829	IDAR39	-23.4772
Min Score = -4.088				Max Score = -4.988			
0 viruses with scores < -4.988				0 normal files with scores > -4.088			

Table 11. HMM Results with 0% dead code copied

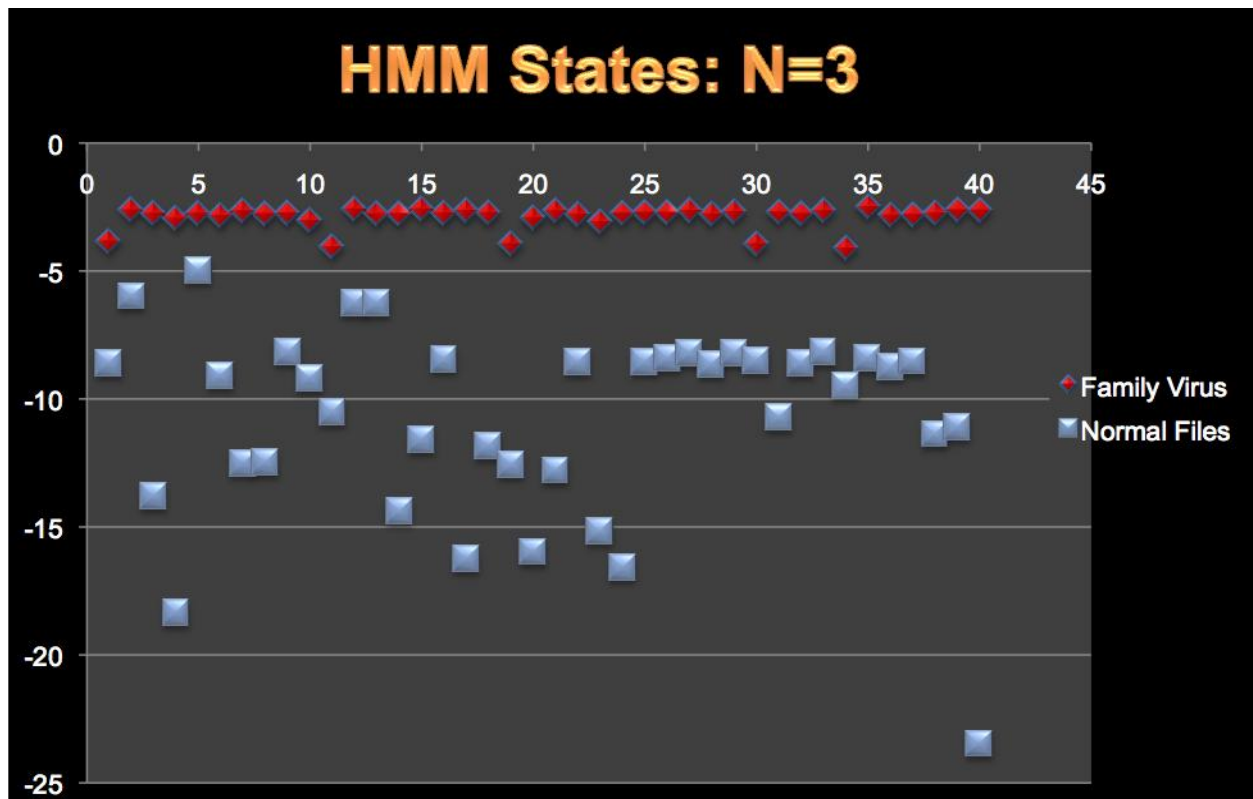


Figure 26. HMM result with 0% dead code copied

5.3.2 Copying Blocks of Dead Code from Normal File

This set of viruses was generated by applying our engine with a probability of 10% to 35% for copying dead code blocks into viruses for each virus instruction. With this setting, the average file size increased from 17KB to 24.3KB.

Even with the higher percentage setting at 35%, the HMM was still able to recognize the family virus. However, we observed that the maximum score of normal files increased as we increased the percentage. Figure 27 shows the maximum score of the normal files as we increased the percentage of dead code copied. The HMM results for 35% dead code copied are shown in Figure 28 and Table 12.

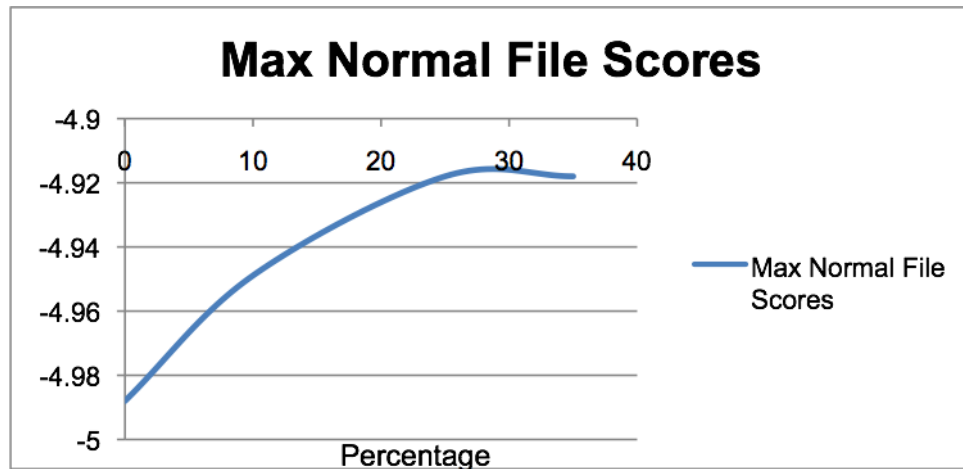


Figure 27. Maximum normal file scores vs. percentage increase

35% dead code copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.540	IDAN140	-2.044	IDAR0	-8.529
IDAN121	-1.930	IDAN141	-2.102	IDAR1	-5.906
IDAN122	-2.051	IDAN142	-2.405	IDAR2	-13.798
IDAN123	-2.439	IDAN143	-2.386	IDAR3	-18.345
IDAN124	-2.138	IDAN144	-2.387	IDAR4	-4.889
IDAN125	-2.171	IDAN145	-2.397	IDAR5	-9.212
IDAN126	-2.129	IDAN146	-2.392	IDAR6	-12.640
IDAN127	-2.383	IDAN147	-2.384	IDAR7	-12.589
IDAN128	-2.444	IDAN148	-2.372	IDAR8	-8.299
IDAN129	-2.309	IDAN149	-2.938	IDAR9	-9.151
IDAN130	-2.707	IDAN150	-2.343	IDAR10	-10.643
IDAN131	-2.091	IDAN151	-2.467	IDAR11	-6.190
IDAN132	-2.497	IDAN152	-2.241	IDAR12	-6.190
				IDAR20	-12.926
				IDAR21	-8.473
				IDAR22	-14.865
				IDAR23	-16.304
				IDAR24	-8.460
				IDAR25	-8.316
				IDAR26	-8.131
				IDAR27	-8.563
				IDAR28	-8.117
				IDAR29	-8.400
				IDAR30	-10.617
				IDAR31	-8.527
				IDAR32	-8.084

IDAN133	-2.479	IDAN153	-2.970	IDAR13	-14.500	IDAR33	-9.386
IDAN134	-2.032	IDAN154	-2.155	IDAR14	-11.527	IDAR34	-8.328
IDAN135	-2.268	IDAN155	-2.295	IDAR15	-8.563	IDAR35	-8.663
IDAN136	-2.079	IDAN156	-2.472	IDAR16	-16.196	IDAR36	-8.440
IDAN137	-2.085	IDAN157	-2.411	IDAR17	-11.622	IDAR37	-11.259
IDAN138	-3.301	IDAN158	-2.418	IDAR18	-12.509	IDAR38	-11.030
IDAN139	-2.331	IDAN159	-2.082	IDAR19	-15.686	IDAR39	-23.439
Min Score = -3.301				Max Score = -4.889			
0 viruses with scores < -4.889				0 normal files with scores > -3.301			

Table 12. HMM results with 35% dead code copied

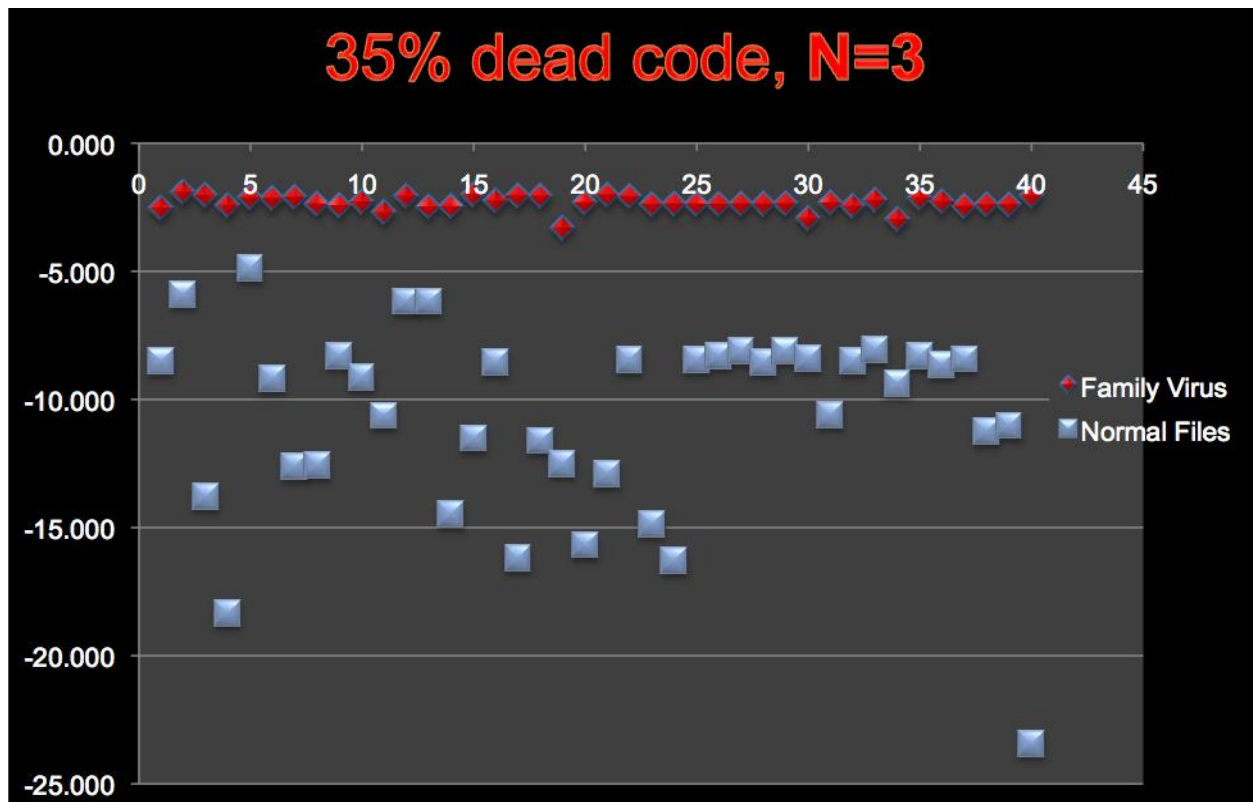


Figure 28. HMM results with 35% dead code copied

5.3.3 Copying Subroutines and Blocks of Dead Code from Normal File

We continued our experiment by adding subroutines copied from the normal files into our viruses. We first configured the sub-routine copying probability to 5% and ran our test. We repeated our tests with different sub-routine copying probabilities of 15%, 20%, and 30%.

Even with a small probability of 5%, we started to see some viruses score lower than the maximum score of the normal files.

Table 13 and Figure 29 show the scores of family viruses and normal files against the HMM model. Sixteen viruses score lower than the maximum normal file score. This means that if the HMM threshold is set to the maximum normal file score, 16 viruses will be undetectable by the HMM. On the other hand, if the HMM threshold is set to the minimum virus file score, then three normal files will be classified as viruses.

35% dead code block, and 5% subroutine copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.441	IDAN140	-2.069	IDAR0	-3.700
IDAN121	-1.960	IDAN141	-2.111	IDAR1	-3.506
IDAN122	-2.014	IDAN142	-2.432	IDAR2	-4.533
IDAN123	-2.482	IDAN143	-2.431	IDAR3	-3.541
IDAN124	-2.170	IDAN144	-2.388	IDAR4	-3.286
IDAN125	-2.140	IDAN145	-2.421	IDAR5	-3.281
IDAN126	-2.154	IDAN146	-2.379	IDAR6	-3.292
IDAN127	-2.431	IDAN147	-2.413	IDAR7	-4.321
IDAN128	-2.474	IDAN148	-2.368	IDAR8	-3.268
IDAN129	-2.275	IDAN149	-2.940	IDAR9	-3.103
IDAN130	-2.720	IDAN150	-2.395	IDAR10	-3.731
IDAN131	-2.115	IDAN151	-2.554	IDAR11	-3.212
IDAN132	-2.492	IDAN152	-2.162	IDAR12	-3.212
IDAN133	-2.515	IDAN153	-2.980	IDAR13	-4.459
IDAN134	-2.079	IDAN154	-2.233	IDAR14	-2.882
IDAN135	-2.311	IDAN155	-2.322	IDAR15	-3.187
IDAN136	-2.191	IDAN156	-2.527	IDAR16	-4.355
IDAN137	-2.123	IDAN157	-2.456	IDAR17	-4.114
IDAN138	-2.643	IDAN158	-2.406	IDAR18	-3.057
IDAN139	-2.514	IDAN159	-2.122	IDAR19	-7.818
Min Score = -2.980			Max Score = -2.242		
16 viruses with scores < -2.242			3 normal files with scores > -2.980		

Table 13. HMM results with 35% dead code blocks and 5% subroutine copied

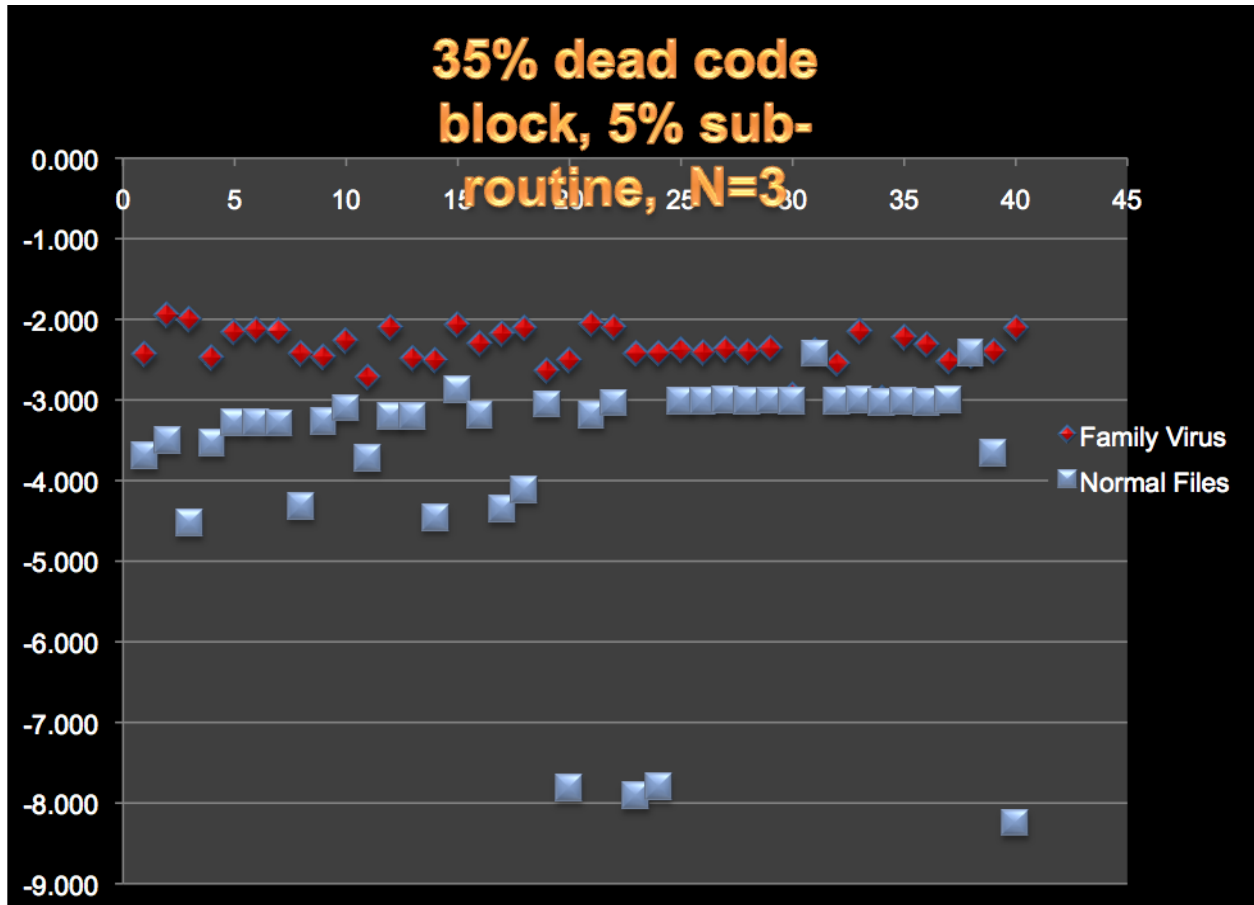


Figure 29. HMM result with 35% dead code blocks and 5% subroutine copied

5.3.4 Copying Subroutines Only from Normal File

Based on the results shown in the previous section, we noticed copying subroutines from normal files significantly impacted our scores. Therefore, we conducted additional experiments by copying only subroutines into our base viruses without additional code obfuscation. The results showed that even with as little as 5% subroutines copied from the normal files, we were able to make the HMM detector misclassify some of our viruses.

5% subroutine copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-3.465	IDAN140	-2.644	IDAR0	-5.886
IDAN121	-2.731	IDAN141	-2.785	IDAR1	-4.786
IDAN122	-2.671	IDAN142	-2.720	IDAR2	-8.619
IDAN123	-2.708	IDAN143	-2.640	IDAR3	-4.688
IDAN124	-2.595	IDAN144	-2.748	IDAR4	-4.428
IDAN125	-2.623	IDAN145	-2.628	IDAR5	-4.530
IDAN126	-2.585	IDAN146	-2.557	IDAR6	-4.509
				IDAR20	-4.098
				IDAR21	-5.566
				IDAR22	-12.088
				IDAR23	-11.677
				IDAR24	-5.344
				IDAR25	-5.349
				IDAR26	-5.328

IDAN127	-3.659	IDAN147	-2.634	IDAR7	-5.763	IDAR27	-5.324
IDAN128	-2.664	IDAN148	-2.472	IDAR8	-4.524	IDAR28	-5.323
IDAN129	-2.634	IDAN149	-4.361	IDAR9	-6.294	IDAR29	-5.472
IDAN130	-4.252	IDAN150	-2.665	IDAR10	-5.150	IDAR30	-5.253
IDAN131	-2.550	IDAN151	-2.700	IDAR11	-6.259	IDAR31	-5.309
IDAN132	-2.521	IDAN152	-2.586	IDAR12	-6.259	IDAR32	-5.306
IDAN133	-2.718	IDAN153	-3.779	IDAR13	-5.937	IDAR33	-5.275
IDAN134	-2.670	IDAN154	-2.599	IDAR14	-3.276	IDAR34	-5.426
IDAN135	-2.701	IDAN155	-2.720	IDAR15	-4.149	IDAR35	-5.295
IDAN136	-2.714	IDAN156	-4.330	IDAR16	-10.162	IDAR36	-5.266
IDAN137	-2.662	IDAN157	-2.550	IDAR17	-5.931	IDAR37	-5.318
IDAN138	-2.441	IDAN158	-2.531	IDAR18	-5.649	IDAR38	-6.460
IDAN139	-2.683	IDAN159	-2.543	IDAR19	-11.896	IDAR39	-12.879
Min Score = -4.361				Max Score = -3.276			
6 viruses with scores < -3.276				3 normal files with scores > -4.361			

Table 14. HMM results with 5% subroutine copied

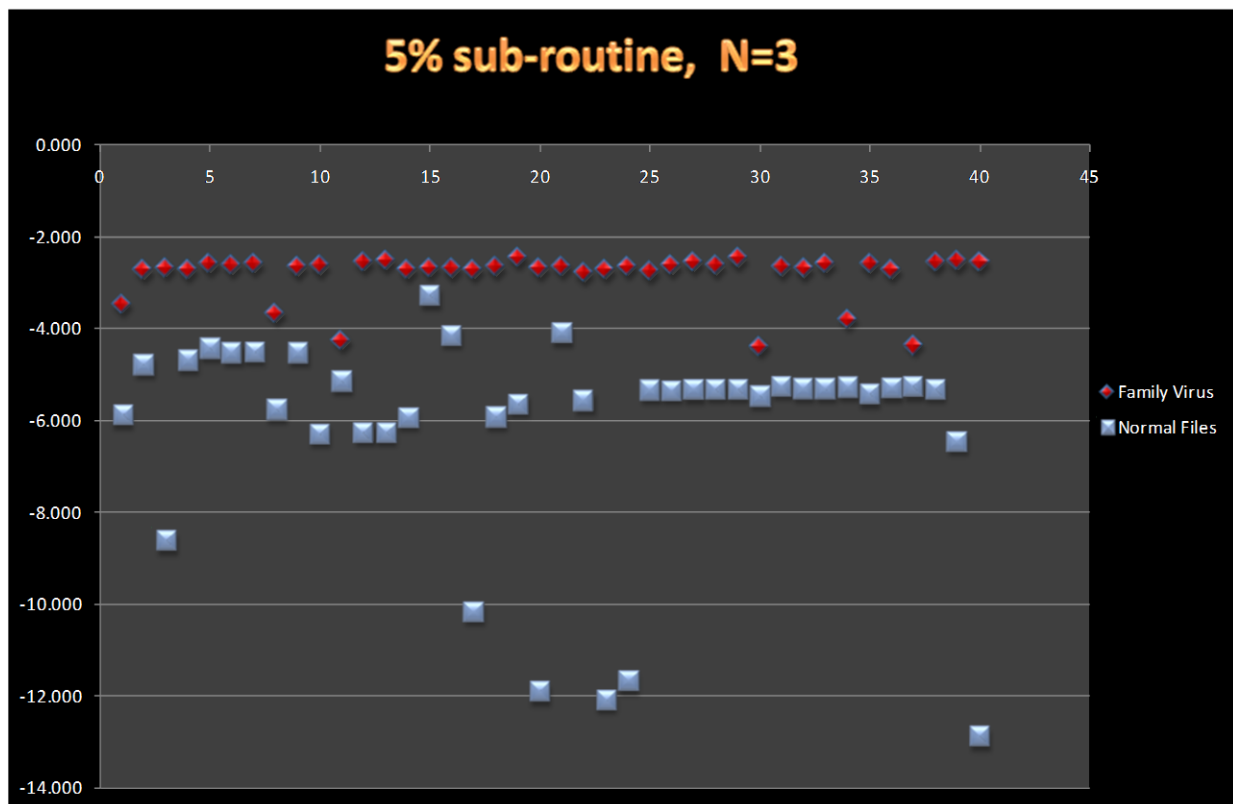


Figure 30. HMM results with 5% subroutine copied

Additional experiment results can be found in Appendix A.

6. Detection Technique for Our Engine

The engine developed in this project copies a fair amount of dead codes and subroutines from normal files. Dead codes are usually surrounded by a “`jmp label_name`” instruction and a `label_name`. Subroutines copied were not executed in the viral copies..

To detect viruses generated by this engine, we can develop a utility to remove these copied dead codes prior to using the HMM detection tool. The utility logic should be as follows:

1. Scan the file and record a block for each “`jmp`” and label pair. In addition, write down the names of all subroutines.
2. For each block, scan the file to see if other instructions might branch into it. If no instruction branches into it, remove it.
3. For each subroutine, scan the file to see if this subroutine is called by outside instructions. If not, remove it.

After a virus is pre-processed by the utility, it will be detectable by the HMM.

We experimented with our detection technique by developing a simple java program to remove the dead code copied from normal files. We first generated the morphed viruses by copying 30% subroutines from normal files. Then we used our program to remove the subroutines copied from normal files. We were able to obtain the same results, as we never copied any subroutines. Note that we trained our HMM model from files with subroutines removed. In other words, we reverted portions of what our engine did in order to detect viruses generated by our engine.

Since our detection method only scans unexecuted codes, virus writers can easily enhance the engine by including code to execute these dead codes with very small probability. For example, virus writers can embed code to first generate two large random numbers, then call the subroutine copied if the two random numbers are equal. The probability of two large consecutive generated random numbers being equal is almost 0, but it will make the copied subroutines harder to remove, since these subroutines will now appear as they “might” be used.

7. Conclusions

By making our viruses closer to normal files, we were able to make them undetectable using an HMM-based detector. The HMM-based detector began to fail when we copied 5% of subroutines from normal files. With our highest setting of 35% dead code blocks and 30% subroutines, most of the scores for viruses and normal files were very close to each other, as shown in Figure 38 of Appendix A.

We also observed that as we increased the amount of dead codes copied, the average scores for viruses and normal files were also closer to each other. In addition, the deviation of normal file scores decreased as more dead codes were copied into virus files. Figure 31 shows the average HMM scores of normal files and family viruses as we increased the percentage of subroutines copied from normal files. With 5% of subroutines copied, the average score of normal files is -6.15, and the average score of family viruses is -2.83. As we increased to 30% subroutines copied, the average score of normal files increased to -3.8, and the average score of family viruses remained almost constant at -2.83.

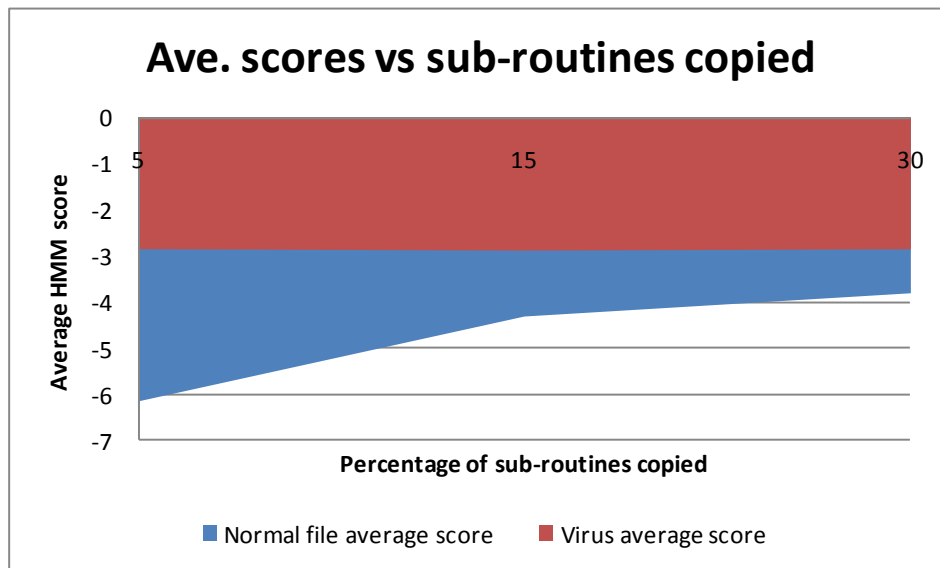


Figure 31. Ave. scores vs. subroutines copied

Figure 32 shows the average HMM scores as we increased the percentage of dead code (not subroutines) copied from normal files. Unlike subroutine copying, more dead code copying did

not seem to change the average scores of normal files. This might be an indication that HMM favors long identical opcode sequences over short identical opcode sequences.

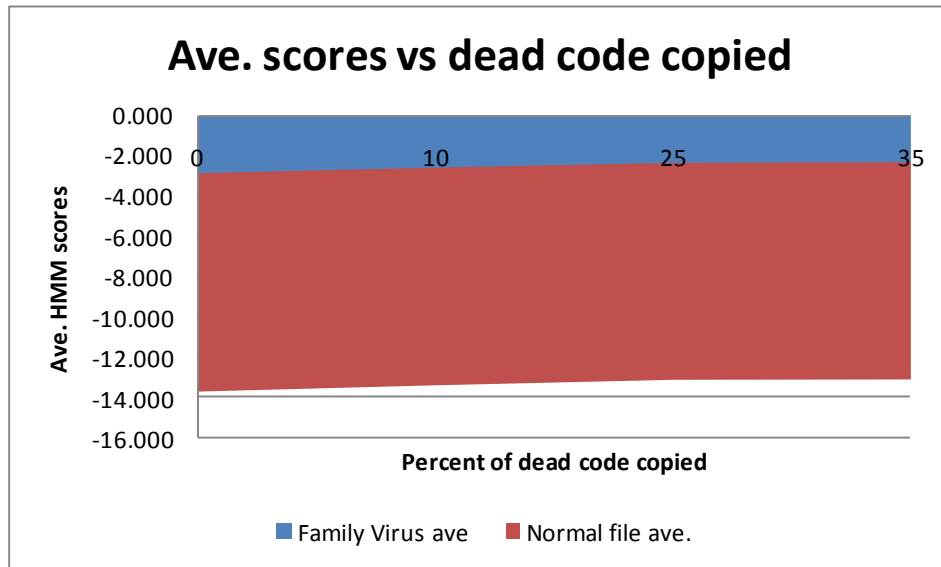


Figure 32. Average scores vs. percent of dead code copied

8. Future Works

Based on previous projects [5] and this project, we concluded that in order to escape HMM-based detection, our metamorphic engine must generate viruses that are highly morphed so that similarities of viruses in the same family are low. However, at the same time, we must maintain some similarity between the viruses and normal files. In order to achieve this goal, we copied some codes from normal files. When inserting the copied code into the virus, we simply used unconditional jump instructions to bypass the copied code so that it would not affect the virus operation. These dead codes could easily be detected and removed. More complicated techniques for copying dead codes from normal files are needed in order to prevent virus detectors from detecting and removing dead codes before scanning.

In this project, we showed that the HMM detector was not able to detect all of our viruses. The logical next step would be to enhance the HMM detector. One possible technique would be to first remove dead codes from a virus before computing that virus' HMM score. Another possible

technique would be to avoid detecting normal files as family viruses by creating more intelligent HMM thresholds.

References

- [1] M. Stamp, "Information Security: Principles and Practice," August 2005.
- [2] 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>
- [3] 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
- [4] P. Szor, "The Art of Computer Virus Defense and Research," Symantec Press 2005.
- [5] P. Desai, "Towards an Undetectable Computer Virus," Master's thesis, San Jose State University, 2008. http://www.cs.sjsu.edu/faculty/stamp/students/Desai_Priti.pdf
- [6] Orr, "The Viral Darwinism of W32.Evol: An In-depth Analysis of a Metamorphic eEngine," 2006. <http://www.antilife.org/files/Evol.pdf>
- [7] Orr, "The Molecular Virology of Lexotan32: Metamorphism Illustrated," 2007.
<http://www.antilife.org/files/Lexo32.pdf>
- [8] E. Konstantinou, "Metamorphic Virus: Analysis and Detection," January 2008.
- [9] A. Venkatesan, "Code Obfuscation and Metamorphic Virus Detection," Master's thesis, San Jose State University, 2008.
http://www.cs.sjsu.edu/faculty/stamp/students/ashwini_venkatesan_cs298report.doc
- [10] P. Zbitskiy, "Code mutation techniques by means of formal grammars and automata," Springer-Verlag France 2009. <http://www.springerlink.com/content/q0k10h3611827181>
- [11] P. Mishra, "A Taxonomy of Software Uniqueness Transformations," December 2003.
<http://www.cs.sjsu.edu/faculty/stamp/students/FinalReport.doc>
- [12] J. Aycock, "Computer Viruses and Malware," Springer Science + Business Media, 2006.
- [13] E. Daoud and I. Jebri, "Computer Virus Strategies and Detection Methods," Int. J. Open Problems Compt. Math., Vol. 1, No. 2, September 2008.
[http://www.emis.de/journals/IJOPCM/files/IJOPCM\(vol.1.2.3.S.08\).pdf](http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf)
- [14] M. Stamp, "A Revealing Introduction to Hidden Markov Models," January 2004.
<http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf>

- [15] Walenstein, R. Mathur, M. Chouchane, R. Chouchane, and A. Lakhota, "The Design Space of Metamorphic Malware," In Proceedings of the 2nd International Conference on Information Warfare, March 2007.
- [16] Wikipedia, "Computer Virus," May 2009, http://en.wikipedia.org/wiki/Computer_virus
- [17] F. Cohen, "Computer viruses: theory and experiments," Computer Security, 6(1):22-35, 1987.
- [18] "Benny/29A," Theme: metamorphism,
<http://www.vx.netlux.org/lib/static/vdat/epmetam2.htm>
- [19] J. Borello and L. Me, "Code Obfuscation Techniques for Metamorphic Viruses," Feb 2008,
<http://www.springerlink.com/content/233883w3r2652537>
- [20] A. Lakhota, "Are Metamorphic Viruses Really Invincible?" Virus Bulletin, December 2005.
- [21] FASM, <http://flatassembler.net/>
- [22] IDA Pro, <http://www.hex-rays.com/idapro/>
- [23] Wikipedia, "Heuristic analysis," March 2009,
http://en.wikipedia.org/wiki/Heuristic_analysis
- [24] HowStuffWorks, "Computer & Internet Security," May 2008,
<http://computer.howstuffworks.com/virus.htm>

Appendix A: Additional HMM results

This section contains the HMM results (with state N=3) for different amounts of dead code block and dead functions copied from normal files.

10% dead code copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.794	IDAN140	-2.299	IDAR0	-8.588
IDAN121	-2.173	IDAN141	-2.426	IDAR1	-5.966
IDAN122	-2.281	IDAN142	-2.790	IDAR2	-13.791
IDAN123	-2.924	IDAN143	-2.540	IDAR3	-18.344
IDAN124	-2.411	IDAN144	-2.565	IDAR4	-4.949
IDAN125	-2.522	IDAN145	-2.574	IDAR5	-9.099
IDAN126	-2.263	IDAN146	-2.502	IDAR6	-12.520
IDAN127	-2.621	IDAN147	-2.639	IDAR7	-12.485
IDAN128	-2.658	IDAN148	-2.479	IDAR8	-8.165
IDAN129	-2.669	IDAN149	-3.526	IDAR9	-9.174
IDAN130	-3.388	IDAN150	-2.617	IDAR10	-10.509
IDAN131	-2.255	IDAN151	-2.647	IDAR11	-6.220
IDAN132	-2.617	IDAN152	-2.425	IDAR12	-6.220
IDAN133	-2.597	IDAN153	-3.493	IDAR13	-14.365
IDAN134	-2.288	IDAN154	-2.413	IDAR14	-11.565
IDAN135	-2.372	IDAN155	-2.594	IDAR15	-8.438
IDAN136	-2.300	IDAN156	-2.622	IDAR16	-16.226
IDAN137	-2.378	IDAN157	-2.526	IDAR17	-11.811
IDAN138	-3.642	IDAN158	-2.558	IDAR18	-12.528
IDAN139	-2.737	IDAN159	-2.439	IDAR19	-15.952
Min Score = -3.642			Max Score = -4.949		
0 viruses with scores < -4.949			0 normal files with scores > -3.642		

Table 15. HMM results with 10% dead code copied



Figure 33. HMM results with 10% dead code copied

25% dead code copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.459	IDAN140	-2.054	IDAR0	-8.564
IDAN121	-1.915	IDAN141	-2.140	IDAR1	-5.949
IDAN122	-2.023	IDAN142	-2.404	IDAR2	-13.793
IDAN123	-2.589	IDAN143	-2.478	IDAR3	-18.363
IDAN124	-2.197	IDAN144	-2.390	IDAR4	-4.919
IDAN125	-2.180	IDAN145	-2.468	IDAR5	-9.133
IDAN126	-2.083	IDAN146	-2.403	IDAR6	-12.549
IDAN127	-2.473	IDAN147	-2.451	IDAR7	-12.514
IDAN128	-2.451	IDAN148	-2.401	IDAR8	-8.203
IDAN129	-2.421	IDAN149	-2.943	IDAR9	-9.181
IDAN130	-2.804	IDAN150	-2.295	IDAR10	-10.538
IDAN131	-2.045	IDAN151	-2.479	IDAR11	-6.213
IDAN132	-2.496	IDAN152	-2.219	IDAR12	-6.213
IDAN133	-2.504	IDAN153	-3.099	IDAR13	-14.397
IDAN134	-2.173	IDAN154	-2.172	IDAR14	-11.555
IDAN135	-2.241	IDAN155	-2.334	IDAR15	-8.473
				IDAR20	-12.830
				IDAR21	-8.501
				IDAR22	-14.762
				IDAR23	-16.202
				IDAR24	-8.492
				IDAR25	-8.347
				IDAR26	-8.162
				IDAR27	-8.593
				IDAR28	-8.147
				IDAR29	-8.430
				IDAR30	-10.623
				IDAR31	-8.557
				IDAR32	-8.115
				IDAR33	-9.415
				IDAR34	-8.357
				IDAR35	-8.694

IDAN136	-2.091	IDAN156	-2.563	IDAR16	-16.212	IDAR36	-8.470
IDAN137	-2.043	IDAN157	-2.401	IDAR17	-11.631	IDAR37	-11.264
IDAN138	-3.240	IDAN158	-2.489	IDAR18	-12.530	IDAR38	-11.051
IDAN139	-2.448	IDAN159	-2.053	IDAR19	-15.586	IDAR39	-23.424
Min Score = -3.24				Max Score = -4.918			
0 viruses with scores < -4.918				0 normal files with scores > -3.24			

Table 16. HMM results with 25% dead code copied

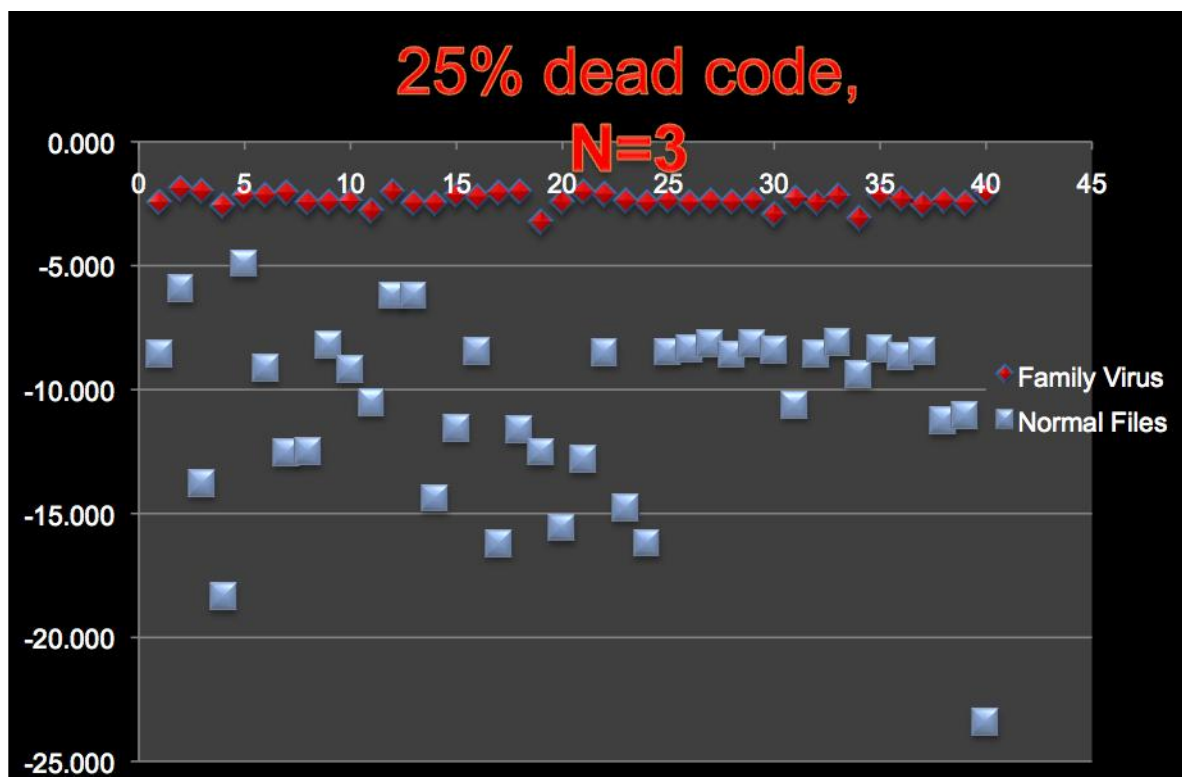


Figure 34. HMM results with 25% dead code copied

35% dead code copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.540	IDAN140	-2.044	IDAR0	-8.529
IDAN121	-1.930	IDAN141	-2.102	IDAR1	-5.906
IDAN122	-2.051	IDAN142	-2.405	IDAR2	-13.798
IDAN123	-2.439	IDAN143	-2.386	IDAR3	-18.345
IDAN124	-2.138	IDAN144	-2.387	IDAR4	-4.889
IDAN125	-2.171	IDAN145	-2.397	IDAR5	-9.212
IDAN126	-2.129	IDAN146	-2.392	IDAR6	-12.640
IDAN127	-2.383	IDAN147	-2.384	IDAR7	-12.589
				IDAR20	-12.926
				IDAR21	-8.473
				IDAR22	-14.865
				IDAR23	-16.304
				IDAR24	-8.460
				IDAR25	-8.316
				IDAR26	-8.131
				IDAR27	-8.563

IDAN128	-2.444	IDAN148	-2.372	IDAR8	-8.299	IDAR28	-8.117
IDAN129	-2.309	IDAN149	-2.938	IDAR9	-9.151	IDAR29	-8.400
IDAN130	-2.707	IDAN150	-2.343	IDAR10	-10.643	IDAR30	-10.617
IDAN131	-2.091	IDAN151	-2.467	IDAR11	-6.190	IDAR31	-8.527
IDAN132	-2.497	IDAN152	-2.241	IDAR12	-6.190	IDAR32	-8.084
IDAN133	-2.479	IDAN153	-2.970	IDAR13	-14.500	IDAR33	-9.386
IDAN134	-2.032	IDAN154	-2.155	IDAR14	-11.527	IDAR34	-8.328
IDAN135	-2.268	IDAN155	-2.295	IDAR15	-8.563	IDAR35	-8.663
IDAN136	-2.079	IDAN156	-2.472	IDAR16	-16.196	IDAR36	-8.440
IDAN137	-2.085	IDAN157	-2.411	IDAR17	-11.622	IDAR37	-11.259
IDAN138	-3.301	IDAN158	-2.418	IDAR18	-12.509	IDAR38	-11.030
IDAN139	-2.331	IDAN159	-2.082	IDAR19	-15.686	IDAR39	-23.439
Min Score = -3.301				Max Score = -4.889			
0 viruses with scores < -4.889				0 normal files with scores > -3.301			

Table 17. HMM results with 35% dead code copied

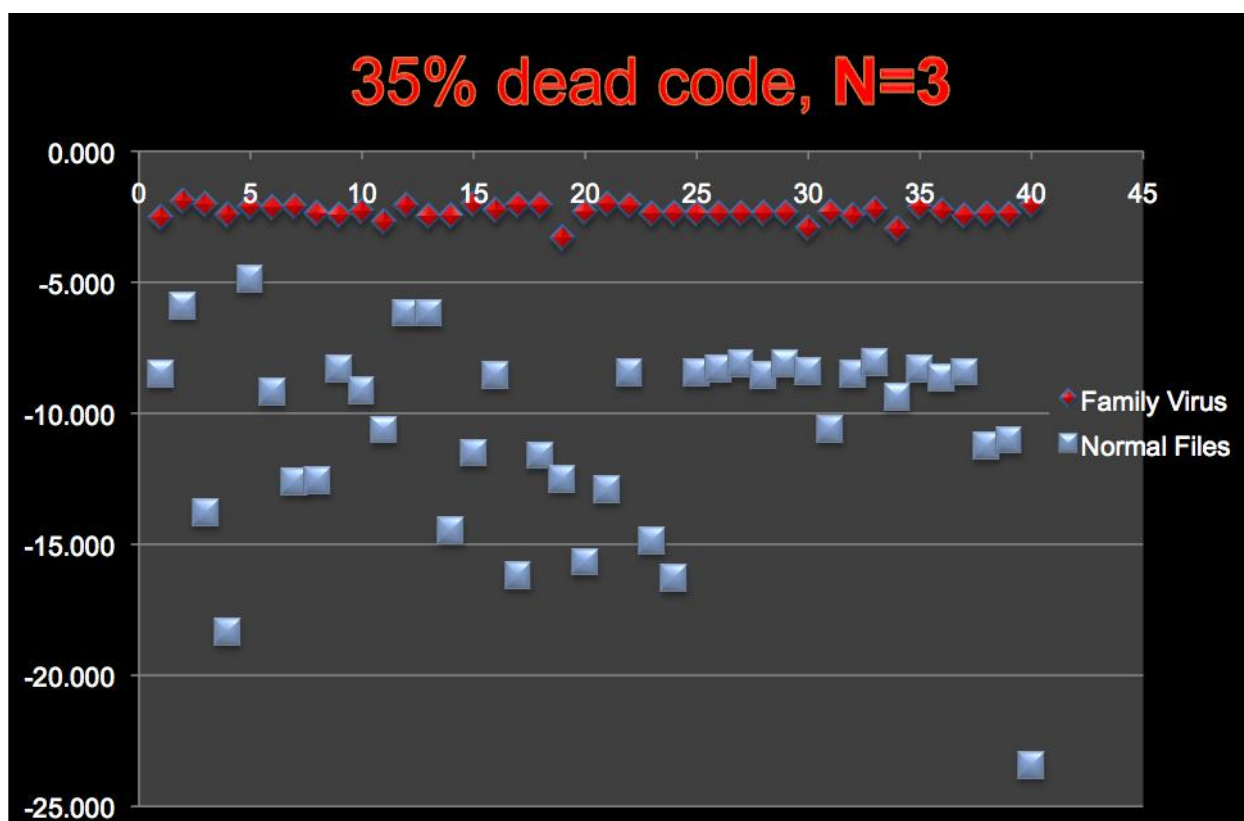


Figure 35. HMM results with 35% dead code copied

35% dead code block, and 15% subroutine copying model with N=3							
Family Viruses			Normal Files				
IDAN120	-2.433	IDAN140	-1.992	IDAR0	-4.287	IDAR20	-3.224
IDAN121	-1.904	IDAN141	-2.103	IDAR1	-3.415	IDAR21	-2.993
IDAN122	-1.965	IDAN142	-2.361	IDAR2	-4.419	IDAR22	-8.343
IDAN123	-2.398	IDAN143	-2.464	IDAR3	-3.465	IDAR23	-8.195
IDAN124	-2.071	IDAN144	-2.372	IDAR4	-3.200	IDAR24	-2.974
IDAN125	-2.109	IDAN145	-2.400	IDAR5	-3.193	IDAR25	-2.966
IDAN126	-2.039	IDAN146	-2.351	IDAR6	-3.189	IDAR26	-2.961
IDAN127	-2.421	IDAN147	-2.361	IDAR7	-4.436	IDAR27	-2.971
IDAN128	-2.439	IDAN148	-2.353	IDAR8	-3.167	IDAR28	-2.961
IDAN129	-2.242	IDAN149	-2.879	IDAR9	-3.724	IDAR29	-2.966
IDAN130	-4.095	IDAN150	-2.341	IDAR10	-3.618	IDAR30	-3.360
IDAN131	-2.080	IDAN151	-2.412	IDAR11	-3.165	IDAR31	-2.971
IDAN132	-2.443	IDAN152	-2.100	IDAR12	-3.165	IDAR32	-2.960
IDAN133	-2.572	IDAN153	-2.893	IDAR13	-4.511	IDAR33	-2.980
IDAN134	-2.018	IDAN154	-2.636	IDAR14	-2.824	IDAR34	-2.968
IDAN135	-2.212	IDAN155	-2.247	IDAR15	-3.089	IDAR35	-2.984
IDAN136	-2.104	IDAN156	-2.435	IDAR16	-4.977	IDAR36	-2.958
IDAN137	-2.076	IDAN157	-2.486	IDAR17	-4.049	IDAR37	-3.594
IDAN138	-2.573	IDAN158	-2.363	IDAR18	-3.538	IDAR38	-3.454
IDAN139	-2.458	IDAN159	-2.085	IDAR19	-8.236	IDAR39	-6.864
Min Score = -4.095			Max Score = -2.824				
3 viruses with scores < -2.824			31 normal files with scores > -4.095				

Table 18. HMM results with 35% dead code blocks and 15% subroutine copied

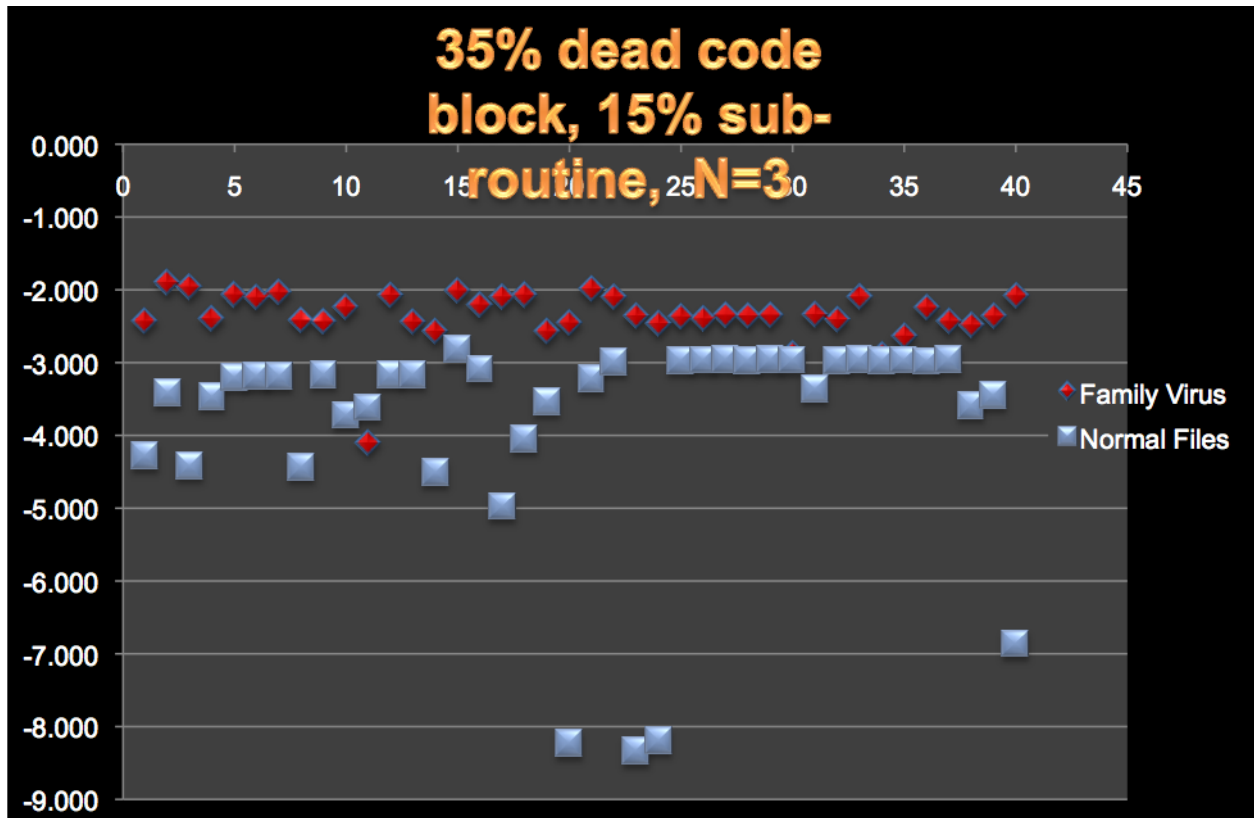


Figure 36. HMM results with 35% dead code blocks and 15% subroutines copied

35% dead code block, and 20% subroutine copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-2.336	IDAN140	-2.028	IDAR0	-3.551
IDAN121	-1.881	IDAN141	-2.073	IDAR1	-3.334
IDAN122	-1.989	IDAN142	-2.363	IDAR2	-4.352
IDAN123	-2.431	IDAN143	-2.459	IDAR3	-3.396
IDAN124	-2.077	IDAN144	-2.358	IDAR4	-3.130
IDAN125	-2.198	IDAN145	-2.373	IDAR5	-3.121
IDAN126	-2.068	IDAN146	-2.376	IDAR6	-3.122
IDAN127	-2.415	IDAN147	-2.543	IDAR7	-4.162
IDAN128	-2.449	IDAN148	-2.333	IDAR8	-3.103
IDAN129	-2.220	IDAN149	-2.902	IDAR9	-2.943
IDAN130	-2.664	IDAN150	-2.351	IDAR10	-3.557
IDAN131	-2.093	IDAN151	-2.379	IDAR11	-3.066
IDAN132	-2.565	IDAN152	-2.086	IDAR12	-3.066
IDAN133	-2.531	IDAN153	-2.973	IDAR13	-4.282
IDAN134	-1.985	IDAN154	-2.143	IDAR14	-2.718
IDAN135	-2.230	IDAN155	-2.355	IDAR15	-3.025
IDAN136	-2.071	IDAN156	-2.396	IDAR16	-4.178
				IDAR20	-3.028
				IDAR21	-2.884
				IDAR22	-6.927
				IDAR23	-6.878
				IDAR24	-2.865
				IDAR25	-2.858
				IDAR26	-2.853
				IDAR27	-2.863
				IDAR28	-2.853
				IDAR29	-2.859
				IDAR30	-2.263
				IDAR31	-2.861
				IDAR32	-2.852
				IDAR33	-2.871
				IDAR34	-2.862
				IDAR35	-2.875
				IDAR36	-2.853

IDAN137	-2.046	IDAN157	-2.513	IDAR17	-3.953	IDAR37	-2.259
IDAN138	-2.544	IDAN158	-2.434	IDAR18	-2.874	IDAR38	-3.357
IDAN139	-2.380	IDAN159	-2.080	IDAR19	-6.850	IDAR39	-6.793
Min Score = -2.973				Max Score = -2.2594			
16 viruses with scores < -2.2594				18 normal files with scores > -2.973			

Table 19. HMM results with 35% dead code blocks and 20% subroutines copied

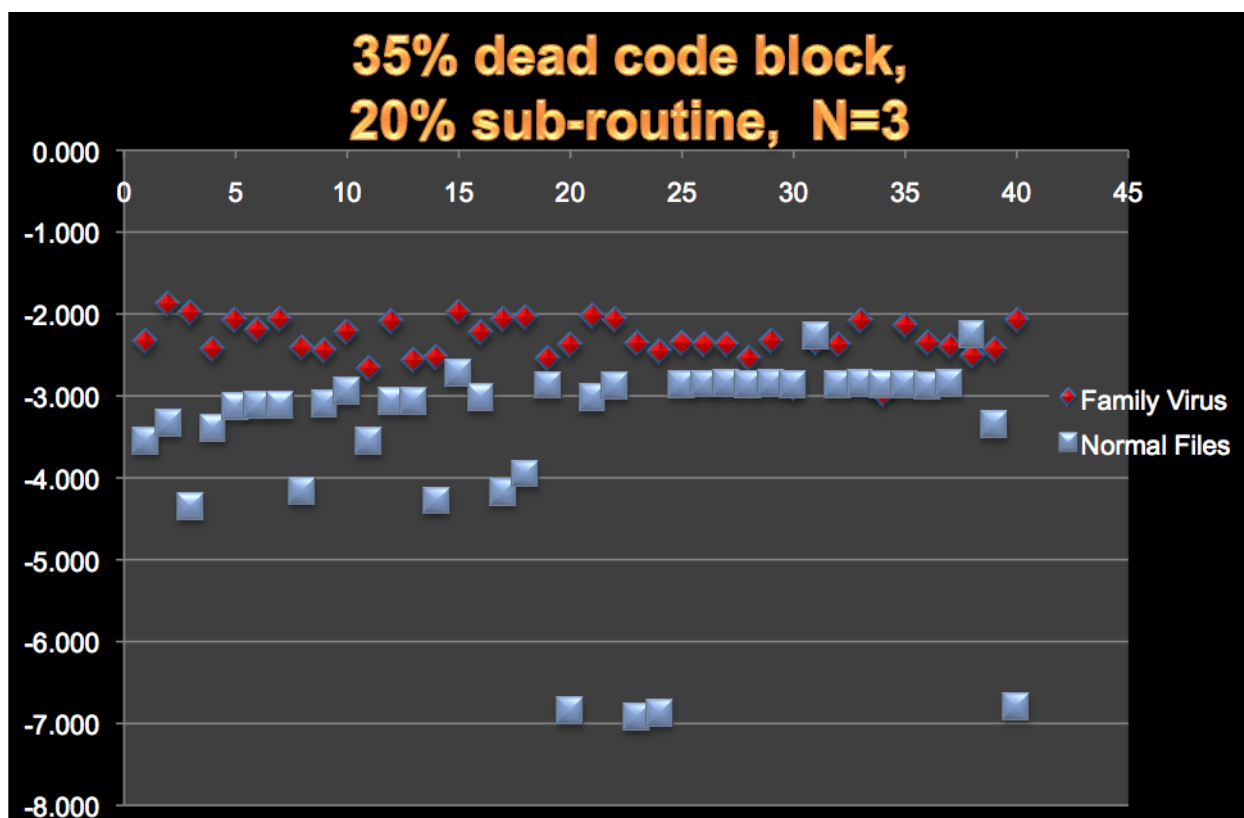


Figure 37. HMM results with 35% dead code blocks and 20% subroutines copied

35% dead code block, and 30% subroutine copying model with N=3							
Family Viruses			Normal Files				
IDAN120	-2.294	IDAN140	-2.029	IDAR0	-2.856	IDAR20	-2.849
IDAN121	-1.891	IDAN141	-2.101	IDAR1	-2.727	IDAR21	-2.708
IDAN122	-2.043	IDAN142	-2.364	IDAR2	-3.754	IDAR22	-6.352
IDAN123	-2.520	IDAN143	-2.437	IDAR3	-2.971	IDAR23	-6.339
IDAN124	-2.100	IDAN144	-2.482	IDAR4	-2.595	IDAR24	-2.693
IDAN125	-2.455	IDAN145	-2.429	IDAR5	-2.880	IDAR25	-2.686
IDAN126	-2.088	IDAN146	-2.388	IDAR6	-2.885	IDAR26	-2.682

IDAN127	-2.443	IDAN147	-2.487	IDAR7	-3.922	IDAR27	-2.691
IDAN128	-2.479	IDAN148	-2.317	IDAR8	-2.881	IDAR28	-2.682
IDAN129	-2.256	IDAN149	-2.813	IDAR9	-2.692	IDAR29	-2.689
IDAN130	-2.708	IDAN150	-2.324	IDAR10	-3.351	IDAR30	-2.119
IDAN131	-2.102	IDAN151	-2.464	IDAR11	-2.853	IDAR31	-2.691
IDAN132	-2.449	IDAN152	-2.153	IDAR12	-2.853	IDAR32	-2.682
IDAN133	-2.531	IDAN153	-2.874	IDAR13	-4.071	IDAR33	-2.700
IDAN134	-2.035	IDAN154	-2.127	IDAR14	-2.508	IDAR34	-2.693
IDAN135	-2.233	IDAN155	-3.993	IDAR15	-2.828	IDAR35	-2.704
IDAN136	-2.128	IDAN156	-2.426	IDAR16	-3.978	IDAR36	-2.685
IDAN137	-2.019	IDAN157	-2.489	IDAR17	-3.762	IDAR37	-2.116
IDAN138	-2.549	IDAN158	-2.401	IDAR18	-2.659	IDAR38	-3.191
IDAN139	-2.439	IDAN159	-2.103	IDAR19	-6.286	IDAR39	-6.628
Min Score = -3.993				Max Score = -2.116			
30 viruses with scores < -2.116				35 normal files with scores > -3.993			

Table 20. HMM results with 35% dead code blocks and 30% subroutines copied

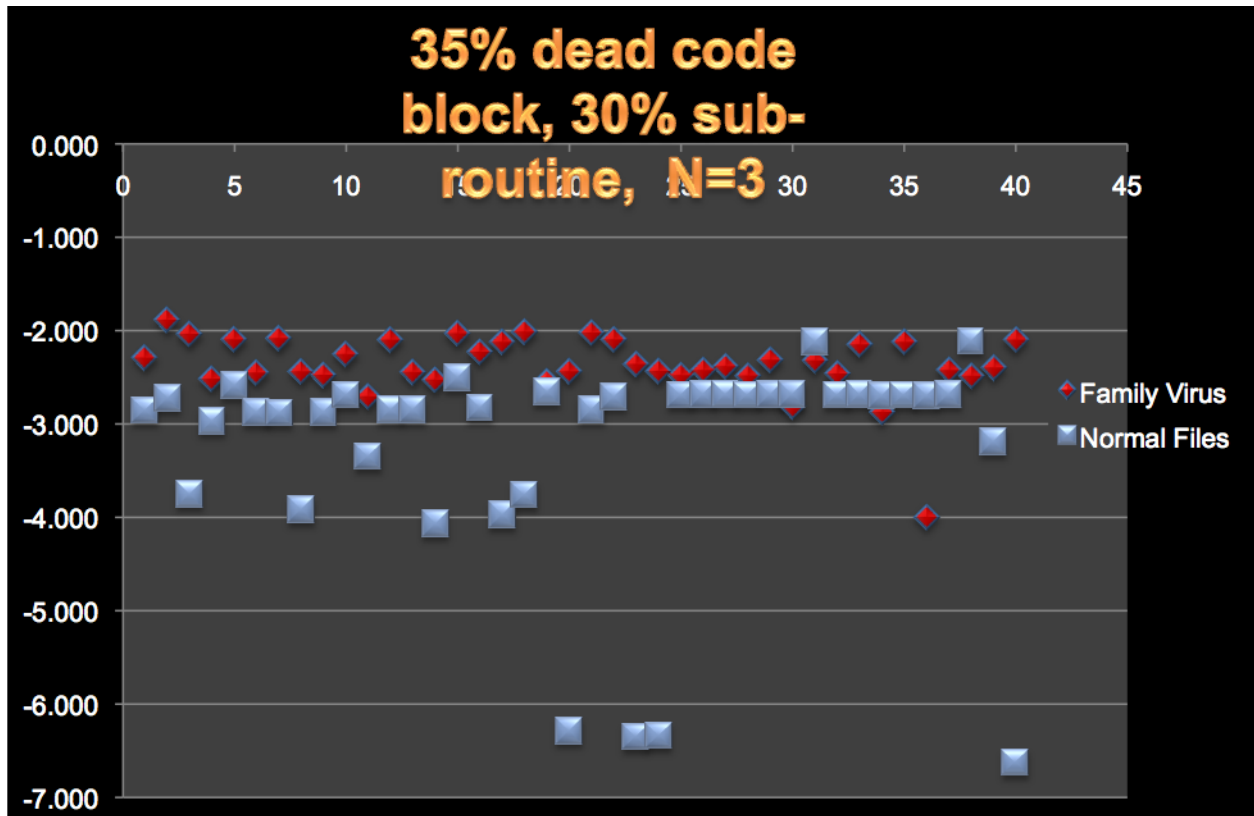


Figure 38. HMM results with 35% dead code blocks and 30% subroutines copied

15% subroutine copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-4.380	IDAN140	-2.445	IDAR0	-4.382
IDAN121	-2.435	IDAN141	-2.740	IDAR1	-4.047
IDAN122	-2.589	IDAN142	-2.735	IDAR2	-4.971
IDAN123	-4.017	IDAN143	-2.606	IDAR3	-3.784
IDAN124	-2.640	IDAN144	-2.706	IDAR4	-3.759
IDAN125	-2.750	IDAN145	-2.624	IDAR5	-3.105
IDAN126	-2.480	IDAN146	-2.505	IDAR6	-3.098
IDAN127	-3.909	IDAN147	-2.689	IDAR7	-4.146
IDAN128	-2.633	IDAN148	-2.632	IDAR8	-3.031
IDAN129	-2.821	IDAN149	-4.411	IDAR9	-3.202
IDAN130	-4.276	IDAN150	-2.804	IDAR10	-3.957
IDAN131	-2.358	IDAN151	-2.624	IDAR11	-3.348
IDAN132	-2.562	IDAN152	-2.642	IDAR12	-3.348
IDAN133	-2.717	IDAN153	-3.834	IDAR13	-4.222
IDAN134	-2.630	IDAN154	-2.441	IDAR14	-2.934
				IDAR20	-2.943
				IDAR21	-3.104
				IDAR22	-14.217
				IDAR23	-13.655
				IDAR24	-3.078
				IDAR25	-3.076
				IDAR26	-3.068
				IDAR27	-3.074
				IDAR28	-3.069
				IDAR29	-3.075
				IDAR30	-2.627
				IDAR31	-3.073
				IDAR32	-3.066
				IDAR33	-3.072
				IDAR34	-3.066

IDAN135	-2.747	IDAN155	-2.746	IDAR15	-2.976	IDAR35	-3.073
IDAN136	-2.514	IDAN156	-3.204	IDAR16	-4.420	IDAR36	-3.060
IDAN137	-2.704	IDAN157	-2.596	IDAR17	-5.026	IDAR37	-2.615
IDAN138	-2.504	IDAN158	-2.581	IDAR18	-3.066	IDAR38	-3.702
IDAN139	-2.882	IDAN159	-2.524	IDAR19	-13.979	IDAR39	-7.980
Min Score = -4.411				Max Score = -2.615			
26 viruses with scores < -2.615				33 normal files with scores > -4.411			

Table 21. HMM results with 15% subroutines copied

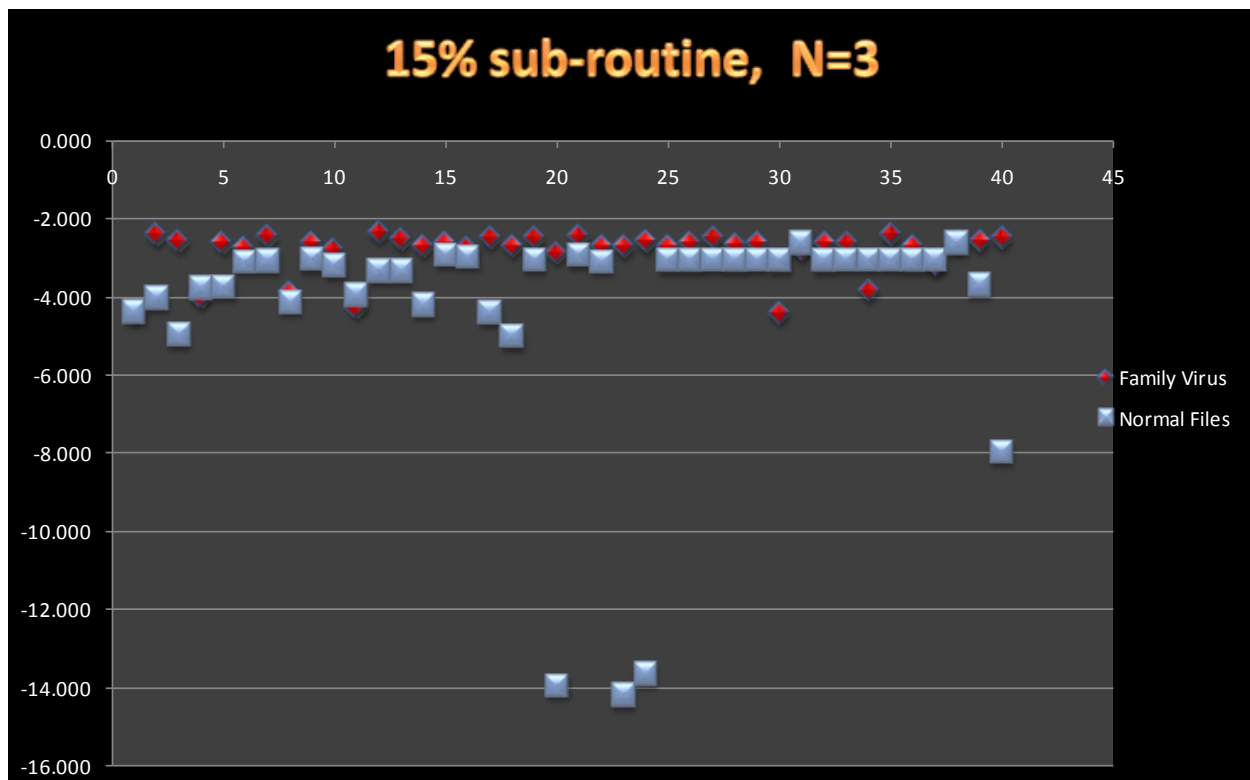


Figure 39. HMM results with 15% subroutines copied

30% subroutine copying model with N=3					
Family Viruses			Normal Files		
IDAN120	-3.415	IDAN140	-2.404	IDAR0	-4.316
IDAN121	-2.206	IDAN141	-2.760	IDAR1	-3.961
IDAN122	-2.641	IDAN142	-2.765	IDAR2	-5.483
IDAN123	-4.149	IDAN143	-2.606	IDAR3	-3.720
IDAN124	-2.502	IDAN144	-2.619	IDAR4	-3.683
IDAN125	-2.711	IDAN145	-2.653	IDAR5	-3.061
IDAN126	-2.427	IDAN146	-2.508	IDAR6	-3.051
IDAN127	-3.693	IDAN147	-2.770	IDAR7	-4.099
				IDAR20	-2.901
				IDAR21	-3.030
				IDAR22	-8.602
				IDAR23	-8.422
				IDAR24	-3.003
				IDAR25	-3.001
				IDAR26	-2.994
				IDAR27	-2.999

IDAN128	-2.606	IDAN148	-2.563	IDAR8	-2.991	IDAR28	-2.994
IDAN129	-3.111	IDAN149	-3.975	IDAR9	-3.123	IDAR29	-3.000
IDAN130	-5.471	IDAN150	-2.626	IDAR10	-3.573	IDAR30	-2.420
IDAN131	-2.301	IDAN151	-2.648	IDAR11	-3.280	IDAR31	-2.998
IDAN132	-2.679	IDAN152	-2.397	IDAR12	-3.280	IDAR32	-2.991
IDAN133	-2.748	IDAN153	-3.717	IDAR13	-4.179	IDAR33	-2.997
IDAN134	-2.632	IDAN154	-2.461	IDAR14	-2.855	IDAR34	-2.992
IDAN135	-2.769	IDAN155	-2.839	IDAR15	-2.935	IDAR35	-2.998
IDAN136	-2.586	IDAN156	-2.985	IDAR16	-4.698	IDAR36	-2.985
IDAN137	-2.702	IDAN157	-2.581	IDAR17	-4.285	IDAR37	-2.411
IDAN138	-2.549	IDAN158	-2.532	IDAR18	-2.989	IDAR38	-3.488
IDAN139	-2.707	IDAN159	-2.530	IDAR19	-8.480	IDAR39	-7.071
Min Score = -5.471				Max Score = -2.411			
36 viruses with scores < -2.411				35 normal files with scores > -5.471			

Table 22. HMM results with 30% subroutines copied

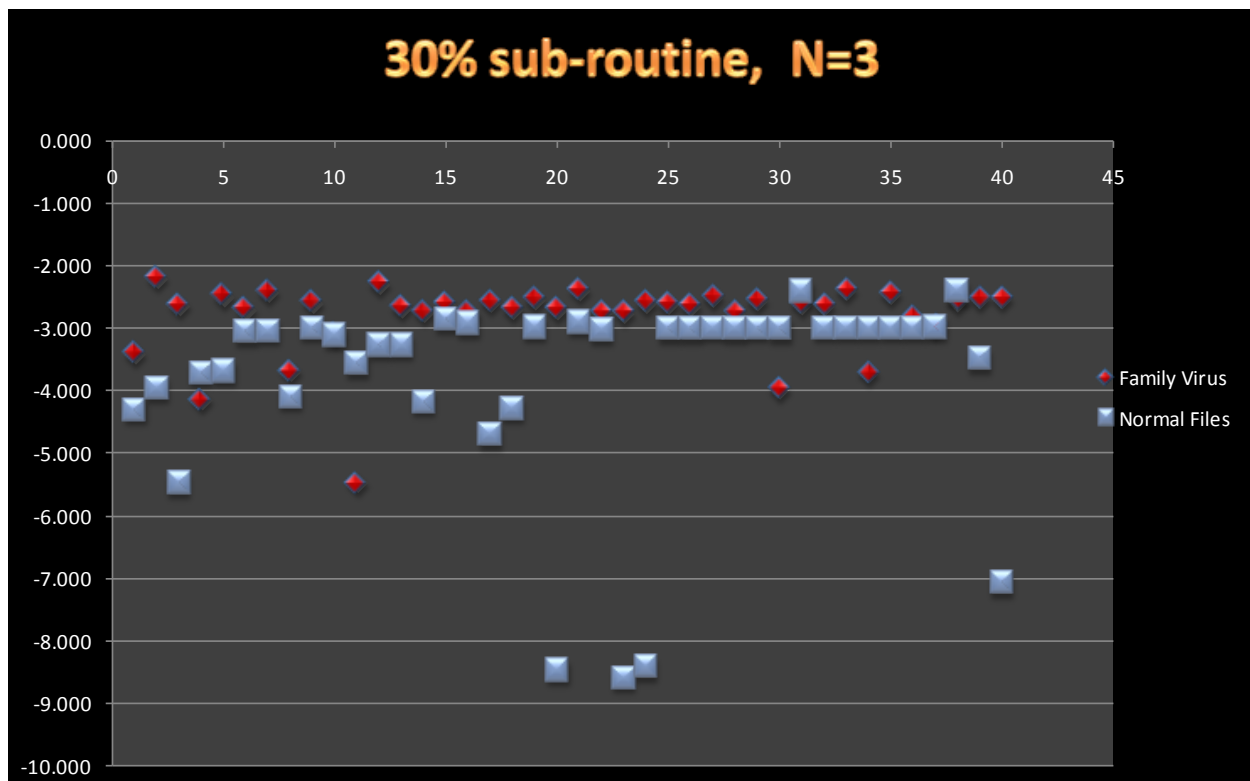


Figure 40. HMM results with 30% subroutines copied

Appendix B: Selected HMM models

Table 23. HMM parameters (A, B, π) of the base virus with N = 3

N=3, M=73, T=66244			
π :			
0.000000000000000	1.000000000000000		
0.000000000000000			
A:			
0.80454192898191	0.000000000000000		
0.19545807101810	0.66390684899439		
0.03930561042629	0.29678754057936		
0.000000000000000	0.97054063542234		
0.02945936457764			
B:			
call	0.03926694759773	0.20431470057355	0.000000000000000
pop	0.02438402763040	0.21185069328542	0.00213671767186
sub	0.06302726222401	0.000000000000000	0.03772599914325
mov	0.34690705188975	0.00000000077741	0.00262771240722
push	0.02582556566454	0.13630720773186	0.41419763794012
or	0.00651875442997	0.000000000000000	0.02158114819045
jz	0.000000000000000	0.18888668026439	0.000000000000000
lea	0.02075606611344	0.00506254058497	0.01312094475559
neg	0.00462031582456	0.000000000000000	0.000000000000000
not	0.00598486505772	0.000000000000000	0.000000000000000
dec	0.01474263146102	0.000000000000000	0.04881216933356
add	0.22077478530833	0.000000000000000	0.01419482179363
cmp	0.000000000000000	0.00008344196438	0.20866347338960
jnz	0.000000000000000	0.13270041420210	0.000000000000000
inc	0.02250586606442	0.00032105013350	0.01585002333567
xor	0.02488896727332	0.00030084773963	0.00802107526509
shr	0.00215096528101	0.00114739274865	0.00797190834412
rcr	0.000000000000000	0.00487869169861	0.000000000000000
jnb	0.00313606929025	0.000000000000000	0.000000000000000
rol	0.00417608939050	0.00102096728348	0.000000000000000
popa	0.02219187963404	0.000000000000000	0.000000000000000
retn	0.01591399543235	0.00894794152082	0.11330336701740
jmp	0.02585992370857	0.000000000000000	0.02922783775542
shl	0.01234775469242	0.00026094037689	0.000000000000000
pusha	0.00238322121666	0.04269195579033	0.01079525374719
and	0.01915015919805	0.00061518155219	0.00439455725426
jb	0.00842914483402	0.00727378117346	0.00439033612996
movzx	0.00991093653559	0.000000000000000	0.000000000000000
ja	0.000000000000000	0.00609836462326	0.000000000000000
adc	0.00371564502868	0.00339785532065	0.00271116030700
xchg	0.00248970386401	0.000000000000000	0.000000000000000
div	0.00207147435775	0.00589267069450	0.000000000000000

imul	0.00383031363694	0.00000000000000	0.00000000000000
rep	0.00126879139224	0.00000000000000	0.00000000000000
lodsw	0.00107727571039	0.00000000000000	0.00000000000000
stosw	0.00100545732970	0.00000000000000	0.00000000000000
cld	0.01149094091083	0.00000000000000	0.00000000000000
stc	0.00381651698490	0.01305670552198	0.00000000000000
test	0.00000000000000	0.00000000000000	0.03154629933991
cld	0.00388283886887	0.00307752411317	0.00155738131141
start	0.00020179056938	0.00572008164448	0.00634401538749
jno	0.00002393946023	0.00000000000000	0.00000000000000
rcl	0.00548213639288	0.00000000000000	0.00000000000000
movsb	0.00270515900609	0.00000000000000	0.00000000000000
lodsb	0.00062242596600	0.00000000000000	0.00000000000000
stosb	0.00064636542623	0.00000000000000	0.00000000000000
sar	0.00140238727500	0.00082913628495	0.00042902930886
sbb	0.00115428773412	0.00130111848997	0.00006419363985
ror	0.00400373046040	0.00282606560572	0.00000000000000
jbe	0.00000000000000	0.01024525256707	0.00000000000000
bound	0.00002950068107	0.00006242257420	0.00000000000000
loop	0.00045484974439	0.00000000000000	0.00000000000000
lodsd	0.00088576002854	0.00000000000000	0.00000000000000
stosd	0.00155606491501	0.00000000000000	0.00000000000000
js	0.00002393946023	0.00000000000000	0.00000000000000
in	0.00001859403840	0.00026209056734	0.00000000000000
std	0.00004787892046	0.00000000000000	0.00000000000000
fild	0.00002393946023	0.00000000000000	0.00000000000000
popf	0.00002393946023	0.00000000000000	0.00000000000000
jnp	0.00000000000000	0.00008023980862	0.00008323461630
ins	0.00000000000000	0.00000000000000	0.00008215182120
fnstenv	0.00000000000000	0.00008131152831	0.00000000000000
scasb	0.00002393946023	0.00000000000000	0.00000000000000
retf	0.00000000000000	0.00016079666922	0.00008399708292
cmc	0.00002393946023	0.00000000000000	0.00000000000000
aad	0.00002393946023	0.00000000000000	0.00000000000000
enter	0.00000000000000	0.00008131152831	0.00000000000000
movsd	0.00002353094247	0.00000000000000	0.00008355371070
jp	0.00002393946023	0.00000000000000	0.00000000000000
repe	0.00004787892046	0.00000000000000	0.00000000000000
jns	0.00000000000000	0.00008131152831	0.00000000000000
fild	0.00002393946023	0.00000000000000	0.00000000000000
fidiv	0.00000000000000	0.00008131152831	0.00000000000000

Table 24. HMM parameters (A, B, π) of the virus without dead code copying with N = 3

N=3, M=87, T=136741			
π :			
	0.0000000000000000	0.0000000000000000	1.0000000000000000
A:			
	0.92839338311941	0.07160661688059	0.0000000000000000
	0.03585375542259	0.89850624506005	0.06563999951736
	0.00005823039915	0.09313151352666	0.90681025607417
B:			
call	0.0000000000000000	0.00441613382699	0.08576566304392
pop	0.0000000000000000	0.05408490232953	0.05470811237037
sub	0.09757405621157	0.06099135705007	0.04217142357619
jz	0.0000000000000000	0.0000000000000000	0.05154796043242
lea	0.0000000000000000	0.02442545039823	0.02060407599871
mov	0.16928513688234	0.47381033417281	0.11182579930031
neg	0.0000000000000000	0.00045499459038	0.00361345332983
jnz	0.0000000000000000	0.000000000000164	0.03572404187886
xor	0.00070419389245	0.01384404338478	0.02770564839415
rol	0.0000000000000000	0.00128466709759	0.00250427493075
dec	0.0000000000000000	0.00673225742317	0.02323511407867
cmp	0.00076220306220	0.00446994170610	0.05457530812780
jmp	0.00920961259212	0.06712556797858	0.03978500846568
push	0.0000000000000000	0.07394660818842	0.16762059542512
test	0.00362804675980	0.01030608264306	0.01127425058831
add	0.0000000000000000	0.12298870624639	0.06515733830152
adc	0.00016071752857	0.00162484582109	0.00371605238089
ret	0.0000000000000000	0.0000000000000000	0.04838775672754
sar	0.0000000000000000	0.00023215165513	0.00141081845623
movzx	0.00020593375645	0.01168520500205	0.00476719183819
and	0.10302654901066	0.01360798070885	0.01954835478095
or	0.10372842051875	0.00863662107323	0.00322952878411
fldcw	0.0000000000000000	0.00270988889517	0.0000000000000000
pusha	0.0000000000000000	0.00008297594549	0.01641601919164
shl	0.10383141500605	0.01015828235549	0.00259894510714
popa	0.0000000000000000	0.0000000000000000	0.02035812386691
jb	0.0000000000000000	0.0000000000000000	0.01131261326238
imul	0.0000000000000000	0.00258084656682	0.0000000000000000
clc	0.0000000000000000	0.0000000000000000	0.01101491291337
rcl	0.0000000000000000	0.00007976046229	0.00515377097507
rep	0.0000000000000000	0.00139486863262	0.00008072016029
shr	0.10029123647780	0.00251187295511	0.00269186633729
sbb	0.00027961770664	0.00243569428346	0.00083147993406
lodsd	0.0000000000000000	0.00011831713882	0.00070222725899
stosd	0.0000000000000000	0.0000000000000000	0.00018320021478
stc	0.0000000000000000	0.0000000000000000	0.00732800859101
cld	0.0000000000000000	0.00210012848363	0.00203357265544
inc	0.0000000000000000	0.01129816855698	0.01402782855571
div	0.0000000000000000	0.00274214947725	0.0000000000000000

jbe	0.00000000000000	0.00000000000000	0.00290830340956
ja	0.00000000000000	0.00000000000000	0.00171750201352
jnb	0.00000000000000	0.00000000000000	0.00302280354379
ror	0.00000000000000	0.00056090070663	0.00389819989411
movsb	0.00000000000000	0.00018853953071	0.00232003518421
not	0.00000000000000	0.00139475046273	0.00353879197888
start	0.00000000000000	0.00000000000000	0.00352660413443
xchg	0.00000000000000	0.00144231485576	0.00042556167394
lodsw	0.00000000000000	0.00000000000000	0.00103050120811
rcr	0.00000000000000	0.00000000000000	0.00137400161082
stosw	0.00000000000000	0.00000000000000	0.00096180112757
loop	0.00000000000000	0.00010759967769	0.00028234223057
movsx	0.00000000000000	0.00072586309692	0.00000000000000
repe	0.00012309451679	0.00101902501266	0.00000000000000
lodsb	0.00000000000000	0.00000000000000	0.00059540069802
repne	0.00000000000000	0.00006452116417	0.00000000000000
fld	0.00023001612580	0.00039654791766	0.00002903242821
fstp	0.00045602501776	0.00019079257682	0.00000000000000
shrd	0.00000000000000	0.00004839087313	0.00000000000000
fmul	0.10584841949022	0.00028346239424	0.00000000000000
fdiv	0.09838969982380	0.00019947028372	0.00000000000000
stosb	0.00000000000000	0.00001170077347	0.00014368870672
js	0.00000000000000	0.00000000000000	0.00002290002685
in	0.00000000000000	0.00000000000000	0.00009160010739
std	0.00000000000000	0.00000000000000	0.00004580005369
jno	0.00000000000000	0.00000000000000	0.00002290002685
popf	0.00000000000000	0.00009951793682	0.00001901548210
bound	0.00000000000000	0.00000000000000	0.00004580005369
jnp	0.00000000000000	0.00000000000000	0.00004580005369
ins	0.00000000000000	0.00000000000000	0.00002290002685
fnstenv	0.00000000000000	0.00001613029104	0.00000000000000
scasb	0.00000000000000	0.00001613029104	0.00000000000000
retf	0.00000000000000	0.00000000000000	0.00006870008054
cmc	0.00000000000000	0.00000000000000	0.00002290002685
fldz	0.00002121763590	0.00002162466282	0.00000000000000
fadd	0.10201907433146	0.00002543691095	0.00000000000000
leave	0.00003224337824	0.00000000000000	0.00002285376964
aad	0.00000000000000	0.00000000000000	0.00002290002685
enter	0.00000000000000	0.00000000000000	0.00002290002685
movsd	0.00000000000000	0.00000000000000	0.00004580005369
jp	0.00000000000000	0.00000000000000	0.00002290002685
jns	0.00000000000000	0.00000000000000	0.00002290002685
file	0.00000000000000	0.00000000000000	0.00002290002685
fidiv	0.00000000000000	0.00000000000000	0.00002290002685
mul	0.00000000000000	0.00020969378355	0.00000000000000
fst	0.00012871351642	0.00000000000000	0.00000000000000
fdivr	0.00006435675821	0.00000000000000	0.00000000000000
cmps	0.00000000000000	0.00009678174626	0.00000000000000

Table 25. HMM parameters (A, B, π) of the virus with most dead code copied with N = 3

N=3, M=112, T=267955			
π :			
0.0000000000000000	1.0000000000000000	0.0000000000000000	
A:			
0.90961917297138	0.08975476506203	0.00062606196659	
0.16278844214058	0.83662974533219	0.00058181252720	
0.00203565693629	0.00296737275980	0.99499697030392	
B:			
call	0.03340242052570	0.04112728877600	0.0000000000000000
pop	0.00049247351741	0.10069227451024	0.00205579749002
sub	0.03198883708658	0.04453184519874	0.10149690582470
or	0.00569552099412	0.00253280176145	0.10358288110428
jmp	0.06844018056837	0.10773195666644	0.00117652102598
push	0.0000000000000000	0.26050410814581	0.00197780928625
mov	0.60610920663406	0.18000930781713	0.17679799939390
test	0.02007972208643	0.00110474220929	0.00154098572633
lea	0.02556527840958	0.01489172008426	0.0000000000000000
neg	0.00029985676599	0.00137469233387	0.0000000000000000
movzx	0.01555110743423	0.00143847222504	0.0000000000000000
and	0.01099206987873	0.00990765183504	0.10019727020307
fldcw	0.00088329378899	0.00044872875468	0.0000000000000000
jnz	0.01065089462651	0.00457439163072	0.0000000000000000
xor	0.01071385431240	0.01624638002724	0.00099218176261
rol	0.00071916443324	0.00092202907328	0.0000000000000000
jz	0.02033761800840	0.00229533890581	0.0000000000000000
add	0.03929308400474	0.10671411456195	0.00183696532896
adc	0.00089512511402	0.00209742585973	0.00013729369729
retn	0.0000000000000000	0.02598057250550	0.0000000000000000
sar	0.00024024157282	0.00077685174143	0.0000000000000000
rep	0.00069560725696	0.00022368202161	0.0000000000000000
shr	0.00505894290636	0.00130804841768	0.09956814041538
shl	0.00858122482443	0.00309136464802	0.10079222536899
jb	0.00140896880094	0.00401375032847	0.00006992784722
imul	0.00020914018121	0.00190338137329	0.0000000000000000
dec	0.00590430588669	0.00911428841993	0.0000000000000000
clc	0.0000000000000000	0.00565715499101	0.0000000000000000
rcl	0.0000000000000000	0.00270508450714	0.0000000000000000
inc	0.01724803646332	0.00521405951870	0.0000000000000000
div	0.00046665719136	0.00088479927294	0.0000000000000000
sbb	0.00068920872227	0.00158769228981	0.00024058451061
lodsd	0.00002161329984	0.00040783226498	0.0000000000000000
stosd	0.0000000000000000	0.00009408989590	0.0000000000000000
stc	0.0000000000000000	0.00376359583602	0.0000000000000000
cld	0.00130775685529	0.00099820664519	0.0000000000000000
popa	0.0000000000000000	0.00815053723237	0.0000000000000000
pusha	0.00020005665856	0.00643612690604	0.0000000000000000
cmp	0.03913069912941	0.00202135521526	0.0000000000000000

jbe	0.00117038365878	0.00000000000000	0.00000000000000
ja	0.00090379626984	0.00000000000000	0.00000000000000
jnb	0.00046571010328	0.00166275403349	0.00000000000000
ror	0.00018295571818	0.00208011816837	0.00000000000000
not	0.00072067122422	0.00126037942031	0.00000000000000
jle	0.00135894547048	0.00000000000000	0.00000000000000
jg	0.00048115772639	0.00000000000000	0.00000000000000
jl	0.00056568543508	0.00000000000000	0.00000000000000
js	0.00014569740258	0.00000696691874	0.00000000000000
movsb	0.00027918621954	0.00082401994636	0.00000000000000
start	0.00000000000000	0.00181123049608	0.00000000000000
xchg	0.00014639238523	0.00061729413806	0.00000000000000
lodsw	0.00000000000000	0.00052925566444	0.00000000000000
rcr	0.00000000000000	0.00070567421925	0.00000000000000
stosw	0.00000000000000	0.00049397195348	0.00000000000000
loop	0.00000000000000	0.00022346350276	0.00000000000000
setnz	0.00040313214914	0.00000000000000	0.00000000000000
movsx	0.00361406683318	0.00177796950003	0.00000000000000
repe	0.00141096252198	0.00000000000000	0.00000000000000
jge	0.00051366838358	0.00000000000000	0.00000000000000
setz	0.00013076523961	0.00003397663227	0.00000000000000
jns	0.00005201705150	0.00000000000000	0.00000000000000
cdq	0.00013654476019	0.00000000000000	0.00000000000000
jp	0.00002629541291	0.00001124230748	0.00000000000000
lodsb	0.00000000000000	0.00030579216168	0.00000000000000
leave	0.00000000000000	0.00027050845071	0.00000000000000
outsd	0.00000000000000	0.00004704494795	0.00000000000000
shld	0.00003251065719	0.00000000000000	0.00000000000000
outsb	0.00000000000000	0.00012937360686	0.00000000000000
repne	0.00005201705150	0.00000000000000	0.00000000000000
fstp	0.00017621498694	0.00034822065477	0.00031892825968
fst	0.00006429696744	0.00000120624953	0.00003462852496
fld	0.00013824276517	0.00045017479467	0.00008453276497
shrd	0.00003901278863	0.00000000000000	0.00000000000000
mul	0.00024057886319	0.00000000000000	0.00000000000000
stosb	0.00000000000000	0.00008232865891	0.00000000000000
in	0.00000000000000	0.00004704494795	0.00000000000000
std	0.00000000000000	0.00002352247398	0.00000000000000
jno	0.00000000000000	0.00001176123699	0.00000000000000
idiv	0.00004551492006	0.00000000000000	0.00000000000000
setl	0.00001905485471	0.00001257799443	0.00000000000000
popf	0.00000000000000	0.00008232865891	0.00000000000000
bound	0.00000000000000	0.00002352247398	0.00000000000000
jnp	0.00000000000000	0.00001583199924	0.00002244459866
ins	0.00000000000000	0.00000000000000	0.00003432509083
fnstenv	0.00000000000000	0.00000000000000	0.00003432509083
scasb	0.00000000000000	0.00000000000000	0.00003432509083
retf	0.00000000000000	0.00003528371096	0.00000000000000
cmc	0.00000650213144	0.00000000000000	0.00000000000000
fldz	0.00001565117786	0.00004488848794	0.00006097059344
fadd	0.00099439591424	0.00073438598041	0.10416377678639

fmul	0.00104298879866	0.00074563300951	0.10263872476798
jo	0.00000000000000	0.00001176123699	0.00000000000000
insd	0.00000000000000	0.00003528371096	0.00000000000000
outsw	0.00000000000000	0.00001176123699	0.00000000000000
arpl	0.00000000000000	0.00001176123699	0.00000000000000
aad	0.00000000000000	0.00001176123699	0.00000000000000
enter	0.00000000000000	0.00001176123699	0.00000000000000
movsd	0.00000000000000	0.00002352247398	0.00000000000000
fdivr	0.00001950639431	0.00000000000000	0.00000000000000
fdiv	0.00111380342971	0.00081453028268	0.09945510750901
fild	0.00000000000000	0.00004704494795	0.00000000000000
fidiv	0.00000000000000	0.00001176123699	0.00000000000000
fxch	0.00000000000000	0.00001099193210	0.00010522048362
fucompp	0.00000000000000	0.00000000000000	0.00010297527248
fnstsw	0.00000000000000	0.00000000000000	0.00017162545414
sahf	0.00000000000000	0.00000000000000	0.00017162545414
fnstcw	0.00000650213144	0.00000000000000	0.00000000000000
fist	0.00001300426288	0.00000000000000	0.00000000000000
fsub	0.00000000000000	0.00000000000000	0.00003432509083
fucom	0.00000000000000	0.00000000000000	0.00003432509083
fucomp	0.00000000000000	0.00000000000000	0.00003432509083
cmps	0.00000000000000	0.00007056742193	0.00000000000000

Appendix C: Built-in Dead code instructions

Built-in dead code leveraged from [5]

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`

Appendix D: Equivalent instruction substitution

Following equivalent instruction substitutions are leveraged from [5].

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
lods b	1. mov al, [esi] add esi, 1
lods d	1. mov eax, [esi] add esi, 4
movs b	1. push eax mov al, [esi] add esi, 1 mov [edi], al add edi, 1 pop eax
movs d	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

or mem, imm	1. push RR mov RR, mem xor RR, imm and mem, imm xor mem, RR pop RR
popad	1. pop edi pop esi pop ebp add esp, 4 pop ebx pop edx pop ecx pop eax
stdcall op1, op2	1. invoke [op1], op2
stosb	1. mov edi, [al] add edi, 1
stosd	1. mov edi, [eax] add edi, 4
sub R, imm	1. add R, new_imm where new_imm = imm x (-1)
sub mem, imm	1. add mem, new_imm where new_imm = imm x (-1)
sub R, 1	1. neg R not R
sub mem, 1	1. neg mem not mem
test R1, R2	1. or R1, R2
xchg R1, R2	1. xor R1, R2 xor R2, R1 xor R1, R2
xor R, R	1. sub R, R 2. mov R, 0 3. and R, 0