Spring 2012

METAMORPHIC WORM THAT CARRIES ITS OWN MORPHING ENGINE

Sudarshan Madenur Sridhara
San Jose State University

METAMORPHIC WORM THAT CARRIES ITS OWN MORPHING ENGINE

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sudarshan Madenur Sridhara

May 2012

The Designated Project Committee Approves the Project Titled

METAMORPHIC WORM THAT CARRIES ITS OWN MORPHING ENGINE

by

Sudarshan Madenur Sridhara

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2012

Dr. Mark Stamp	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Teng Moh	Department of Computer Science

ABSTRACT

METAMORPHIC WORM THAT CARRIES ITS OWN MORPHING ENGINE

by Sudarshan Madenur Sridhara

Metamorphic malware changes its internal structure across generations, but its functionality remains unchanged. Well-designed metamorphic malware will evade signature detection. Recent research has revealed techniques based on hidden Markov models (HMMs) for detecting many types of metamorphic malware, as well as techniques for evading such detection.

A worm is a type of malware that actively spreads across a network to other host systems. In this project we design and implement a prototype metamorphic worm that carries its own morphing engine. This is challenging, since the morphing engine itself must be morphed across replications, which imposes significant restrictions on the structure of the worm. Our design also employs previously developed techniques to evade detection. We provide test results to confirm that this worm effectively evades signature and HMM-based detection, and we consider possible detection strategies. This worm provides a concrete example that should prove useful for additional malware detection research.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mark Stamp for trusting me with his idea. His continued support and guidance were the keys to the success of this project.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER 1

Introduction

Metamorphism is the process of transforming a piece of software into unique instances [20]. In metamorphic software, copies of the software are functionally equivalent but their internal structure differs. Metamorphism is used by virus writers to avoid detection by antivirus software which primarily use signature based detection techniques [2].

Metamorphism provides virus writers the opportunity to develop malware that is undetectable, with respect to static analysis [10]. Therefore, it is natural to expect an increase in volume as well as complexity of metamorphic viruses in the near future.

Although metamorphic viruses have been extensively studied [1, 6, 12, 18, 25, 26, 30], a metamorphic worm presents significant challenges. Metamorphic viruses do not need to carry their own morphing engine. In the case of some highly metamorphic viruses, such as NGVCK [19], the metamorphic generator is separate from the virus body.

Unlike viruses, worms are self-propagating [2], and therefore a metamorphic worm would, most likely, need to carry its own morphing engine. Since the morphing engine itself can act as a signature, a worm that carries its own morphing engine must morph its own morphing engine, as well as the actual worm code, across replications. This presents significant complications and imposes some restrictions on the structure of the morphing engine.

In this paper, we develop and analyze a worm that carries its own morphing engine. That is, the morphing engine morphs itself and the worm across replications.

The resulting metamorphic worms are evaluated based on the lack of similarity between successive generations [14] and their ability to evade detection [30].

CHAPTER 2

Malware types and detection techniques

Malware is a term used to refer to software with malicious functionality. Malware can exist as an independent executable or infect a benign executable by becoming a part of it. There are different kinds of malware, primarily distinguished from one another by their methods of replication and infection [2].

2.1 Malware types

We discuss two of the most prominent kinds of malware: viruses and worms. Each one of these types is explained in more detail in the sections that follow.

2.1.1 Viruses

A Virus is a malicious piece of code that tries to attach itself to other executable code upon execution. The executable file to which the virus successfully attaches to, is said to be "infected" [2]. Viruses employ numerous methods to escape detection by common detection methods like signature based detection. The most prominent methods used for this purpose are encryption, polymorphism and metamorphism [2].

2.1.1.1 Encrypted viruses

Most of the executable portion of an encrypted virus is encrypted. A small block of decryptor code exists in the virus to decrypt its encrypted body, when the virus is being executed. Encryption using different keys defeats signature based detection by not providing a common signature that can be used to detect the virus.

However, the decryptor remains constant across generations, because of which, the code pattern of the decryptor can be used for detection.

2.1.1.2 Polymorphic viruses

A polymorphic virus is essentially an encrypted virus which changes its decryptor loop across generations. A polymorphic virus, theoretically, has an infinite number of variations of the decryptor loop and therefore, has no common part in the virus body across replications [2].

The most common method to detect polymorphic viruses is code emulation. Although the decryptor changes across replications, the encrypted body will result in the same block of code once decrypted. This enables in memory detection of the virus, once the decryptor has performed the decryption.

2.1.1.3 Metamorphic viruses

Metamorphic viruses do not use encryption and decryptor functions. Instead, the entire body of the virus is changed across generations, while retaining functionality. This produces a new virus body for each replication.

A key component of metamorphic viruses is a mutation engine [15]. The mutation engine is responsible for morphing the body of the metamorphic virus across generations. The mutation engine can be independent of the resultant metamorphic virus, or it can be stored as part of the virus body. In the former case a higher degree of metamorphism can be achieved because the mutation engine itself need not be morphed. In the latter case, the mutation engine itself needs to be morphed across generations. This places restrictions on the structure of the mutation engine, and also the level of metamorphism that can be achieved using it [7, 28].

Some of the morphing techniques employed by metamorphic viruses are explained in more detail in the Chapter 3.

2.1.2 Worms

Like viruses, worms are self-replicating malware. However, there are several characteristics that distinguish a worm from a virus.

Firstly, worms are standalone [2]. They do not rely on a host executable to which they need to attach to. Secondly, unlike viruses which only replicate and infect executable programs within their host machine, worms spread from host to host across the network. Worms also operate without human intervention [22].

Similar to viruses, worms can employ different techniques to avoid signature based detection. Polymorphism and metamorphism have been employed by worms [3, 7].

2.2 Detection techniques

As viruses continue to evolve, there has been a corresponding evolution in virus detection technologies as well. This section presents some techniques that antivirus software most commonly employ. Some niche techniques are also presented.

2.2.1 Signature-based detection

Signature-based detection is by far, the most commonly used technique for virus detection [23]. A signature comprises of sequences of bytes extracted from a virus, which can be used to uniquely identify the virus. A signature scanner scans executable files for such signatures, using a database of virus signatures [2]. If a

match occurs, the executable is assumed to be infected by the virus corresponding to the matching signature.

Signature scanning is implemented using algorithms that can be used to scan for a large number of signatures quickly. However, there are several drawbacks in signature based detection. To be able to detect new virus, a signature needs to be extracted from the virus and added to the signature database. Also, signature based scanning is easily defeated by using techniques like polymorphism and metamorphism [23].

2.2.2 Anomaly-based detection

Heuristic methods can be employed by anti-virus software to detect anomalous behavior, instead of looking for specific virus signatures. Heuristic methods can be either static or dynamic. Static heuristics involve static code analysis to look for suspicious structures like decryption loops, self-modifying code, use of undocumented API calls, manipulation of interrupt vectors, etc. [2, 27].

Dynamic heuristic methods determine whether an executable is infected by analyzing its behavior while it is running. Common dynamic methods are behavior monitoring and code emulation [27].

Anomaly-based detection systems can provide protection against zero-day attacks by detecting even unknown viruses. However, anomaly-based detection systems have much higher false positive and false negative rates compared to other detection methods [9].

2.2.3 Integrity checkers

Integrity checkers detect changes to files by comparing their check-sum to their original check-sum stored in a white list. Viruses, with very few exceptions,

operate by changing files. Integrity checkers watch for unauthorized file modifications to determine virus like behavior [4].

2.2.4 Hidden Markov Model based detection

Hidden Markov Models (HMMs) are statistical models, widely used in many problems involving pattern recognition. In the past few years, considerable research has been done on the use of Hidden Markov Models for metamorphic virus detection. A method to detect metamorphic viruses is presented in [30]. It involves training an HMM with opcode sequences extracted from viruses belonging to a certain family. This trained HMM is then used to score other executable files, to check whether they belong to the same virus family. A detailed explanation of the process is given in Section 5.2.

CHAPTER 3

Metamorphic techniques

The metamorphic worm described in this paper makes use of morphing techniques like equivalent instruction substitution and dead code insertion. Apart from these two techniques, there are a number of other morphing techniques employed by metamorphic malware. These techniques may be as elementary as equivalent instruction substitution, or as advanced as formal grammar mutation.

Some of the metamorphic techniques presented in [2] and [3] are explained in this section.

3.1 Register Swap

Register swap is a simple metamorphic technique. It mutates the virus body by swapping the operand registers with different registers. For example, `POP ECX` might be replaced with `POP EBX`, if it is permissible. Opcode sequence remains the same using this technique.

3.2 Subroutine permutation

In this technique changes in the structure of a virus is obtained by reordering the virus' subroutines. If a virus has n different subroutines, then it can generate n-factorial different generations without repeats. Viruses which only use this method may be detected matching multiple short signatures in the same binary. An example of one such permutation is shown in Figure 3.1.

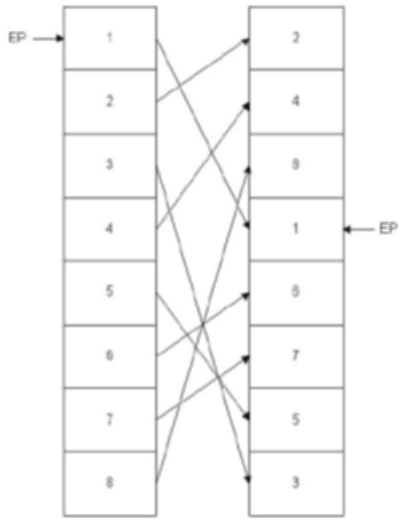

Figure 3.1: Subroutine permutation [24]

3.3 Garbage Instruction Insertion

Garbage instructions are instructions are of two types:

(1) Instructions that are either not executed

(2) Instructions that have no effect.

By adding garbage instructions, a virus can potentially generate an unlimited number of unique copies. Examples of instructions that have no effect include `NOP`, `ADD EAX, 0`, etc. Such instructions can contribute greatly to metamorphism.

3.4 Instruction substitution

This involves substituting a single instruction or a group of instructions with another instruction or a group of instructions with the same functionality. For instance, `MOV R1, R2` is equivalent to `PUSH R1` followed by `POP R2`.

3.5 Transposition

Transposition involves instruction re-ordering. If instructions have no dependency between them, their order of execution can be changed without any change in the overall functionality of the program. For example, instructions:

```
1: ADD [Op1], [Op2]
2: ADD [Op3], [Op4]
```

can be re-ordered as shown below without any resultant change in functionality.

```
1: ADD [Op3], [Op4]
2: ADD [Op1], [Op2]
```

The same is true for groups of instructions which have no dependency on one another. This helps to evade signature based detection, as the order of instruction bytes change in the morphed executable.

3.6 Formal grammar mutation

Formal grammar mutation is a formalization of many existing morphing techniques [3, 8, 31]. Classical morphing engines can be viewed as non-deterministic automata, since transitions are possible from every symbol to every other symbol [31], where the symbol set is the set of all possible instructions. In other words, any instruction can be followed by any other instruction. By formalizing mutation techniques, one can apply formal grammar rules and create viral copies with great variation.

Figure 3.2 illustrates a simple polymorphic decryptor template and two possible mutations of the decryptor code achieved using the formal grammar shown

in Figure 3.3. With this decryptor template and formal grammar combination, it is possible to generate 960 different decryptors [31].

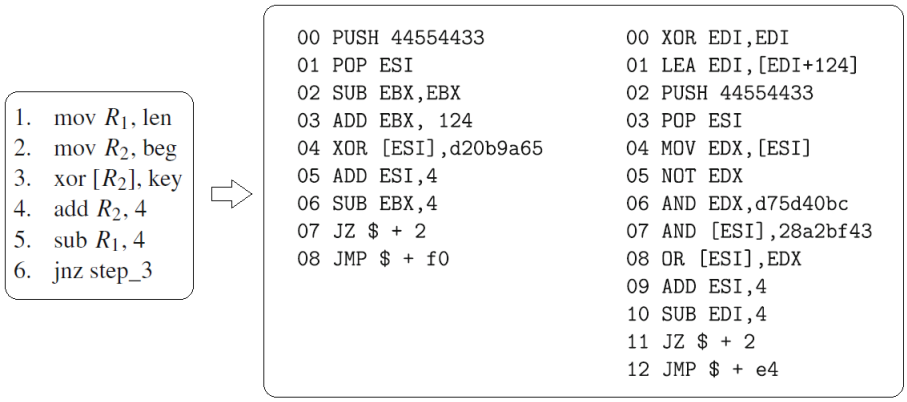

Figure 3.2: A simple polymorphic decryptor and two variants [31]

$$
\begin{aligned}
A &\rightarrow XB \\
B &\rightarrow Y_4\varepsilon \\
X &\rightarrow X_1X_2|X_2X_1 \\
X_1 &\rightarrow GX_1|mov\ R_1, len|push\ len \oplus pop\ R_1|xor\ R_1, \\
&\quad R_1 \oplus lea\ R_1, [R_1 + len]|sub\ R_1, R_1 \oplus add\ R_1, len \\
X_2 &\rightarrow GX_2|mov\ R_2, beg|push\ beg \oplus pop\ R_2|xor\ R_2, \\
&\quad R_2 \oplus lea\ R_2, [R_2 + beg]|sub\ R_2, R_2 \oplus add\ R_2, beg \\
Y_4 &\rightarrow GY_4|W_1|S_4W_4 \\
W_1 &\rightarrow GW_1|xor\ [R_2], key\ H_1 \\
W_1 &\rightarrow not\ [R_2] \oplus xor\ [R_2], key \oplus not[R_2]\ H_1 \\
W_1 &\rightarrow mov\ R_3, [R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and\ [R_2], \\
&\quad \neg key \oplus or\ [R_2], R_3\ H_1 \\
H_1 &\rightarrow GH_1|add\ R_2, 4\ H_2|sub\ R_2, -4\ H_2 \\
S_4 &\rightarrow GS_1|sub\ R_2, 4|add\ R_2, -4 \\
W_2 &\rightarrow GW_2|xor\ [R_1][R_2], key\ H_2 \\
W_2 &\rightarrow not\ [R_1][R_2] \oplus xor\ [R_1][R_2], key \oplus not[R_1][R_2]\ H_2 \\
W_2 &\rightarrow mov\ R_3, [R_1][R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and \\
&\quad [R_1][R_2], \neg key \oplus or\ [R_1][R_2], R_3\ H_2 \\
H_2 &\rightarrow GH_2|sub\ R_1, 4 \oplus jnz\ xxx|sub\ R_1, 4 \oplus jz\ yyy \oplus jmp\ xxx \\
H_2 &\rightarrow add\ R_1, -4 \oplus jnz\ xxx|add\ R_1, -4 \oplus jz\ yyy \oplus jmp\ xxx \\
H_2 &\rightarrow sub\ ecx, 3 \oplus loop\ xxx \Leftrightarrow R_1 \equiv ecx
\end{aligned}
$$

Figure 3.3: Formal grammar for decrpytor mutation [31]

CHAPTER 4

Binary similarity

The ability of the metamorphic worm described in this paper to evade signature-based detection, is evaluated by analyzing the similarity between various generations of the worm.

4.1 N-gram similarity

In [14], an n-gram based similarity measure is proposed and analyzed. This method can be used to compare sequences of instructions in two assembly program files [12, 30]. This method calculates a score that represents the percentage of similarity between the two files. The method is summarized below:

(1) Extract instruction opcodes from the two given assembly programs X and Y. Let the length of the extracted opcode sequences from programs X and Y be 'm' and 'n' respectively. Assign numbers in a sequence to the extracted opcodes: 1 to the first opcode, 2 to the second opcode, etc.

(2) For all opcode sub-sequences of length three from X's opcode sequence, check if corresponding sub-sequences occur in the opcode sequence extracted from Y. If a match occurs, (x, y) will be marked on a graph where, x is the position of the first opcode of the matching sub-sequence in Xs opcode sequence, and y is the position of the first opcode in the matching sub-sequence in Y's opcode sequence.

(3) After matching all opcode sequences, an m x n graph is plotted on which all matching sub-sequences are marked. The x-axis corresponds to the opcode

numbers extracted from X and the y-axis corresponds to the opcode numbers extracted from Y. Retain only those line segments on the graph whose length is above a certain threshold value (say five). This is done to eliminate random matches and noise.

(4) Since a sequential match is done between both sequences, matching sequences of opcodes result in line segments that are parallel to the diagonal ((0,0), (m,n)). If the matching sequence occurs in the same starting location in both opcode sequences, the resulting line segment will fall on the diagonal. If the matching sequence occurs at different starting locations in both opcode sequences, the resulting line segment will be parallel to the diagonal.

(5) For each axis, calculate the sum of the number of opcodes that are covered by one or more of the line segments retained in (3). This sum is divided by the total number of opcodes on the corresponding axis to find the percentage of match for the assembly program represented by the axis. The final similarity score is the average of the similarity percentage calculated for both axes.

The method summarized above is illustrated graphically in Figure 4.1.

This n-gram based similarity measure is used both in [30] and [12], to compare the similarity between assembly programs that are obtained by disassembling benign files of viruses belonging to the NGVCK family.

An example comparison between two virus files, generated by the NGVCK metamorphic generator [19], presented in [30] is shown in Figure 4.2. The left part of the graph shows matching sub-sequences without eliminating noise. The right

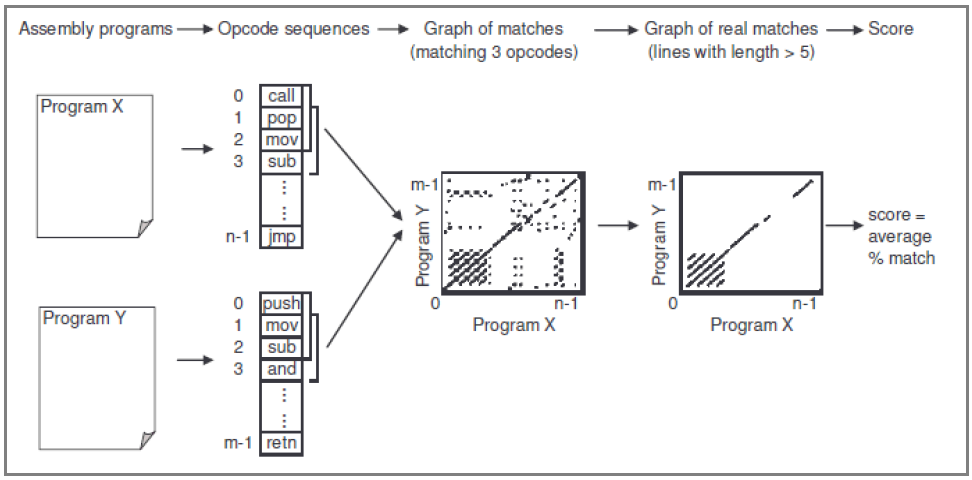

Figure 4.1: Similarity based on n-gram analysis [14]

part shows matching sub-sequences retained after eliminating noise. The final similarity score in this particular instance is 21%.

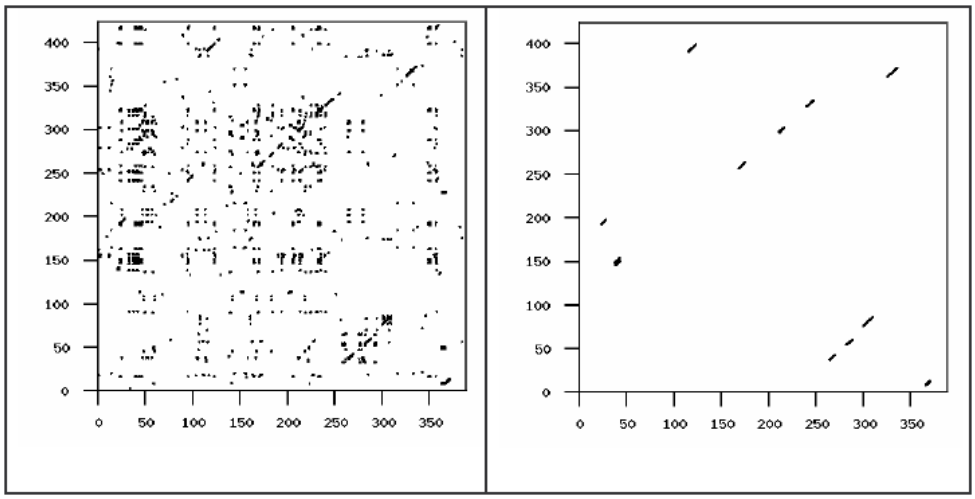

Figure 4.2: n-gram similarity of two NGVCK viruses [30]

4.2 Similarity using graph technique

A method for measuring similarity between executable files using opcode graphs is presented in [18]. This method involves creating weighted directed graphs

using opcodes extracted from executable files. We now summarize the methods used to generate opcode graphs, and to compute similarity using these generated graphs.

4.2.1 Opcode graphs

Each opcode that appears in an executable file's extracted opcode sequence is a node in the directed graph. A directed edge is inserted from this node to every other node corresponding to the successor opcodes. Edge weights represent the transition probabilities to successor nodes.

Counts for opcode pairs are tabulated from the extracted opcode sequence to form a matrix. For each opcode, the counts are converted to probabilities by dividing the count for a digram by the row sum. The resulting matrix is an opcode graph that represents the program using which it was created.

4.2.2 Similarity Score

Let N be the number of distinct opcodes. The opcodes are mapped to numbers 0 to $N-1$. Let A and B be the opcode graphs for the executable files in question. Elements of the matrices A and B are represented by a_{ij} and b_{ij} respectively.

To compare the matrices, the similarity score $S(A, B)$ is computed using the following scoring function:

$$S(A,B) = \frac{1}{N^2}\left(\sum_{i,j=0}^{N-1} |a_{ij} - b_{ij}|\right)^2$$

If A is the same as B, then the minimal score of 0 is obtained. On the other hand, if $a_{ij} = 1$ and $b_{ik} = 1$ with $j \neq k$, the maximum possible row sum of 2 is obtained. If this maximum row sum is obtained for each row, $S(A, B) = 4$. Therefore, $0 \leq S(A, B) \leq 4$.

CHAPTER 5

Hidden Markov Models and virus detection

Over the past few years, there has been significant research on the use of Hidden Markov Models (HMMs) for metamorphic virus detection [1, 6, 12, 18, 25, 26, 30]. A method is presented in [30] in which an HMM is trained using sequences of opcodes from viruses that belong to a particular family. This trained HMM is then used to score binaries, to determine whether the binaries are viruses that belong to the same family. A threshold can be obtained based on the Log Likelihood Per Opcode (LLPO) score for viruses and benign binaries, which is used to categorize new binaries as viruses or benign binaries based on their LLPO score. This section explains how HMMs work and the way HMMs can be used in virus detection.

5.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a statistical model used to model a Markov process whose states are unknown [21]. Some HMM notations are now presented. The notations presented here are based on the notations used in [21]:

$T \rightarrow$ Length of the observation sequence

$N \rightarrow$ Number of states in the model

$M \rightarrow$ Number of observation symbols

$Q \rightarrow \{q_0, q_1, \ldots, q_{N-1}\} -$ Number of observation symbols

$V \rightarrow \{0, 1, \ldots, M-1\} -$ Set of possible observations

$A \rightarrow$ state transition probabilities

$B \rightarrow$ observation probability matrix

$\pi \rightarrow$ initial state distribution

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

Figure 5.1 illustrates a generic HMM. The state of the HMM and the observation at time t are represented by X_t and O_t respectively. The initial state X_0 and the A matrix determine the hidden Markov process which is represented in the figure by the portion on top of the dotted line.

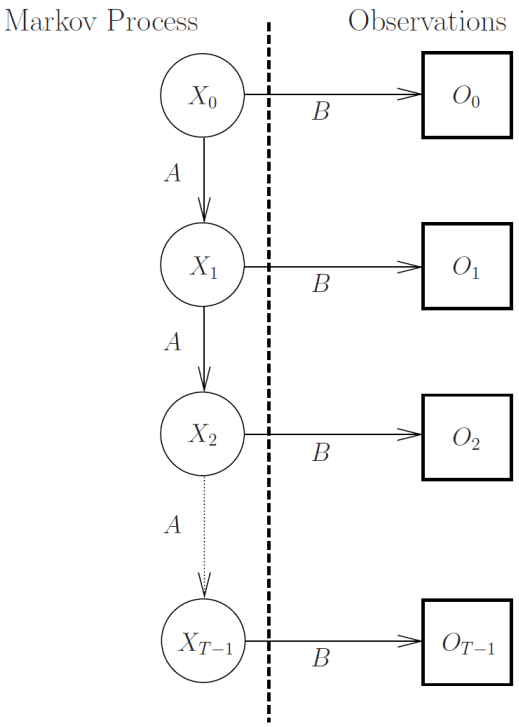

Figure 5.1: Generic Hidden Markov Model

First, the HMM is trained using input data that the HMM needs to represent. Each individual element in the training data maps to an observation symbol. Unique observation symbols are extracted from the set of observations. The trained

model will then be used to determine if a given new sequence of observations is a pattern similar to the one represented by the model.

5.1.1 An example

Here we illustrate the inner workings of an HMM using a simple example [17]. A genie is sitting behind a curtain with urns U_0, U_1 and U_2, each containing balls colored red, green and blue in different proportions. Balls colored red, green and blue are indicated by R, G and B respectively. The color of the ball chosen by the genie, at any point of time, is governed by a Markov process unknown to the observer. The observer can only see the color of the chosen ball.

The objective is to predict the urn from which the balls are retrieved based on the order and color of the balls drawn from them. In this example, U_0, U_1 and U_2 are the states. The possible observations are R, G and B.

The transition probabilities from each state to every other state are represented by matrix A as shown below:

$$A = \begin{array}{c} \\ U_0 \\ U_1 \\ U_2 \end{array} \begin{array}{c} \begin{array}{ccc} U_0 & U_1 & U_2 \end{array} \\ \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.2 & 0.2 \\ 0.5 & 0.2 & 0.3 \end{bmatrix} \end{array}$$

The probability of observations for each of the states is represented by the B matrix as shown below:

$$B = \begin{array}{c} \\ U_0 \\ U_1 \\ U_2 \end{array} \begin{array}{c} \begin{array}{ccc} R & G & B \end{array} \\ \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.3 & 0.6 \end{bmatrix} \end{array}$$

Also, the initial probability distribution is represented by the π matrix as shown below:

$$\pi = \begin{bmatrix} 0.4 & 0.3 & 0.3 \end{bmatrix}$$

The matrices π, A and B are row-stochastic; i.e, each row sums up to 1. Each row represents a probability distribution. Now, given a sequence of observations $O = (R,\ B)$, the objective is to find the most likely state sequence, given the model λ.

In the HMM sense, the most likely state sequence is calculated choosing the state for which the sum of probabilities over all possible state sequences is the highest for each observation. This process is illustrated in Table 5.1. In this case, the optimal state sequence in the HMM sense is U_0U_2.

Table 5.1: Finding HMM optimal state sequence

State sequence probabilities		
State	Probability	Normalized probability
U_0U_0	0.0040	0.0376
U_0U_1	0.0160	0.1504
U_0U_2	0.0600	0.5639
U_1U_0	0.0072	0.0677
U_1U_1	0.0024	0.0226
U_1U_2	0.0072	0.0677
U_2U_0	0.0030	0.0282
U_2U_1	0.0012	0.0113
U_2U_2	0.0054	0.0508

HMM probabilities		
	Element	
	1	2
$P(U_0)$	0.7519	0.1335
$P(U_1)$	0.1580	0.1843
$P(U_2)$	0.0903	0.6824

5.2 The three problems

The following three problems can be solved efficiently using HMMs [21].

(1) Given $\lambda = (A, B, \pi)$ and O, the observation sequence, find $P(O|\lambda)$.

(2) Given $\lambda = (A, B, \pi)$ and O, the observation sequence, find the optimal state sequence for the Markov process.

(3) Given O, the number of unique symbols M and the number of states N, find λ.

Problem 1 involves to determining the likelihood of an observation sequence using the model. Problem 2 deals with uncovering the "hidden" part of the HMM. Problem 3 deals with training the HMM using the given observation sequence O and the parameters M and N.

In this paper we will be dealing with problem 1 and 3. We will make use of methods to solve problem 3 to train a HMM using opcode sequences extracted from executable files. Scoring opcode sequences to be tested using this model involves solving problem 1. We will not be dealing with problem 2, as previous research indicates that the meaning of the HMM states themselves has not been of much consequence to the detection capability of the HMM [12, 30].

5.2.1 Forward Algorithm

The forward algorithm or α pass is used to determine $P(O|\lambda)$.

For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

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

The probability of the partial observation sequence up to time t is $\alpha_t(i)$. Using the forward algorithm, $P(O|\lambda)$ can be computed as shown below:

(1) Let $\alpha_0(i) = \pi_i b_i(O_0)$, for $i = 0, 1, \ldots, N-1$.

(2) For $t = 1, 2, \ldots, T-1$, and $i = 0, 1, \ldots, N-1$, compute

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right] b_i(O_t).$$

(3) $P(O|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$.

5.2.2 Backward Algorithm

Backward algorithm or β pass can be used to determine the most likely state sequence.

For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots, O_{T-1} | x_t = q_i, \lambda)$$

Then, $\beta_t(i)$ can be computed efficiently as shown below:

(1) Let $\beta_{T-1}(i) = 1$, for $i = 0, 1, \ldots, N-1$.

(2) For $t = T-2, T-3, \ldots, 0$, and $i = 0, 1, \ldots, N-1$, compute

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j).$$

For $t = 0, 1, \ldots, T-2$ and $i = 0, 1, \ldots, N-1$, define

$$\gamma_t(i) = P(x_t = q_i | O, \lambda).$$

Since the relevant probability up to time t is measured by $\alpha_t(i)$, and the relevant probability after time t is measured by $\beta_t(i)$,

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}.$$

From the definition of $\gamma_t(i)$, the most likely state at any time t is the state for which $\gamma_t(i)$ is maximum.

5.2.3 Baum-Welch Algorithm

This algorithm adjusts model parameters to best-fit the observations. The number of states N and the number of unique observations symbols M are fixed. However, the contents of the A, B and π are free, subject only to the row stochastic condition. The re-estimation process is explained below:

(1) Initialize $\lambda = (A, B, \pi)$ with an approximate guess. If no such guess is possible, use random values. For example $\pi_i = {}^1/_N$, $A_{ij} = {}^1/_N$, $B_{ij} = {}^1/_M$.

(2) Compute $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$ and $\gamma_t(i,j)$ where $\gamma_t(i,j)$ is a di-gamma. Di-gammas can be defined as:

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

The $\gamma_t(i)$ and $\gamma_t(i,j)$ (or di-gamma) are related by

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

(3) Re-estimate model parameters as follows: For $i = 0, 1, \ldots, N-1$ let

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

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

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

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

$$b_j(k) = \sum_{t \in \{0,1,\ldots,T-2\}, O_t = k} \gamma_t(j) \Big/ \sum_{t=0}^{T-2} \gamma_t(j)$$

(4) If $P(O|\lambda)$ increases, go to step 3.

5.3 HMMs and Virus detection

The use of HMMs for metamorphic virus detection is explained in great detail in [29, 30]. The basic objective is to train an HMM using opcodes extracted from viruses belonging to a particular family. The trained HMM will, in effect, represent the statistical properties of the virus family. Using this trained HMM, we can compute a score for any given program to determine how "close" the file is to the virus family that the HMM represents. We can then classify the file based on a predetermined threshold.

First, a collection of viruses belonging to the same family are disassembled. From each of the disassembled files, only the instruction opcodes are extracted. An example of extracted opcodes is shown in Figure 5.2. Opcodes sequences extracted from all the virus files are concatenated. This concatenated sequence forms the sequence observations used to train an HMM. The set of unique opcodes in the observation sequence is the set of distinct observation symbols.

An HMM is now trained using this sequence of observations, as explained in Section 5.1.1. To detect whether a given program belongs to the virus family, this trained HMM is used to calculate the Log Likelihood per Opcode (LLPO). If the LLPO of the program is within a particular threshold, the file is classified as belonging to the virus family. We now take a briefly look at what Log Likelihood Per Opcode means.

5.3.1 Log Likelihood Per Opcode

Scoring observation sequences and training the HMM involves computation of product of probabilities. The result of multiplication tends to 0 exponentially as T increases. As a result, the use of methods described in Section 5.2 inevitably results

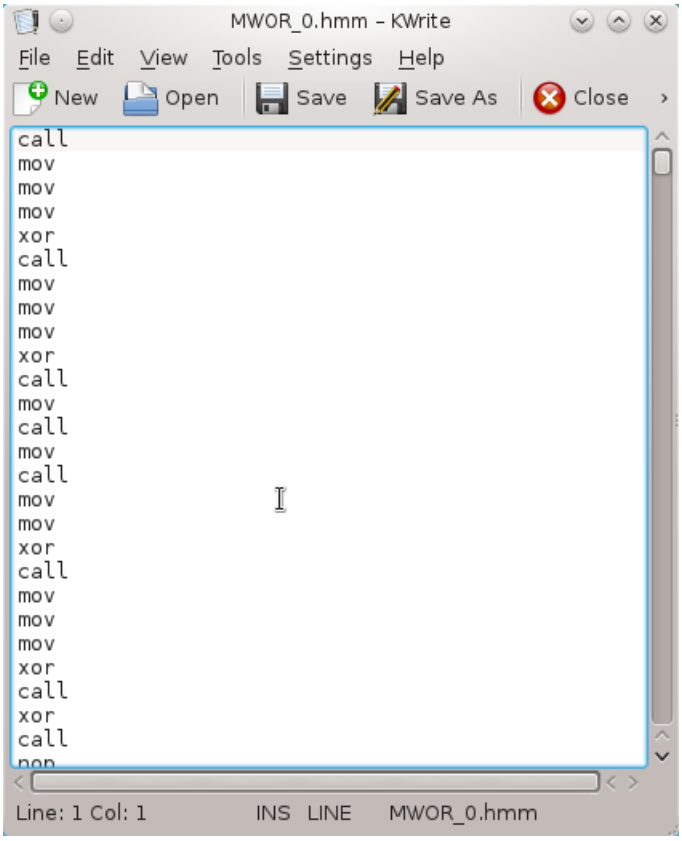

Figure 5.2: Extracted opcode sequence

in underflow. To avoid this problem, the forward and backward algorithms normalize the result of each iteration. This process is called scaling. HMM scaling is explained in detail in [21].

Once scaling is employed, $P(O|\lambda)$ is redefined as:

$$P(O|\lambda) = 1 \Big/ \prod_{j=0}^{T-1} c_j$$

where c_j is the scaling factor at time j. However, this computation is also susceptible to underflow and to avoid that, we compute:

$$\log[P(O|\lambda)] = -\sum_{j=0}^{T-1} \log c_j$$

This is the log likelihood. Log likelihood is length dependent, as the sum of log transition probabilities and log observation probabilities will be higher for a

longer sequence. As the sequences in the test set may be of different lengths compared to the sequences used to train the model, log likelihood is divided by the number of opcodes in the sequence to obtain Log Likelihood Per Opcode which accounts for the length difference [30].

5.3.2 Effectiveness of HMM detection

HMM detection has proven to be very effective in detection of highly metamorphic viruses [30]. Experiments with HMM detection in [29] indicate a detection rate of about 90% and a false positive rate of less than 10%.

5.4 Evading HMM detection

There has been some research on methods to evade HMM detection [12]. The method presented in [12] involves inserting dead code from the benign files in the test set, into the virus files. This helps in making the virus files statistically similar to normal files. This is achieved by making use of a dynamic scoring algorithm which inserts a block of dead code only if it results in the virus file becoming more similar [14] to normal files.

Results presented in [11] indicate that, with an increase in the amount of dead code inserted from normal files, the average LLPO scores for viruses and normal files become closer. The HMM-detector showed indications of failing when 5% of the subroutines were copied from the normal file. The LLPO scores for viruses and normal files were the closest when 35% dead blocks and 30% subroutines were copied from normal files.

Results in [11] also indicate that inserting long sequences of opcodes, like subroutines, are more effective in defeating HMM detection than randomly inserted

blocks of dead code. The worm presented in this paper makes use of this result while inserting dead code.

CHAPTER 6

Design and implementation

We have implemented a metamorphic worm that carries its own morphing engine. As mentioned above, this presents a significant challenge since both the worm body and the morphing engine itself must be morphed. This section presents the structure of our metamorphic worm in detail.

6.1 Structure

Figure 6.1 illustrates the structure of the various components of the worm. The worm consists of the following:

(1) Body - This is the central component that controls the worm's life cycle. It controls and coordinates the activities of all the other active components of the worm.

(2) Disassembler - Disassembles the binary portion of the worm and extracts instructions from it.

(3) Morphing Engine - The morphing engine operates on the set of disassembled instructions. It removes old dead code instructions, adds new dead code and employs equivalent instruction substitution.

(4) Reassembler - The reassembler re-structures the control flow in the morphed body of code and converts the morphed body to binary.

(5) Payload - This is the actual piece of code the worm is intended to run on every computer it infects. In our case, the payload is benign; it simply

appends a line of text to a temporary file.

(6) Pad_block_1 and Pad_block_2 (Padding blocks) - These are blocks of dead code that are replaced from generation to generation. The purpose of doing this is to make the worm statistically similar to normal files and thereby evade HMM detection. The blocks also help to avoid relocating sections and other book-keeping information in the executable from generation to generation [12].

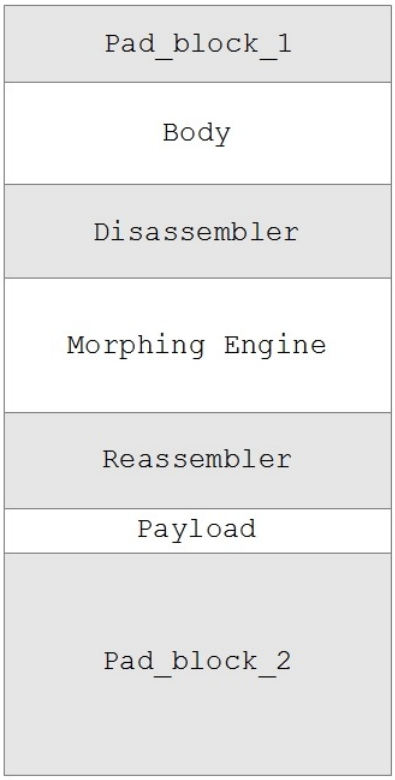

Figure 6.1: Metamorphic worm components

6.2 Memory Layout

We now examine the layout of different components of the worm in the address space of the worm's process. The placement of the worm in memory is illustrated in Figure 6.2.

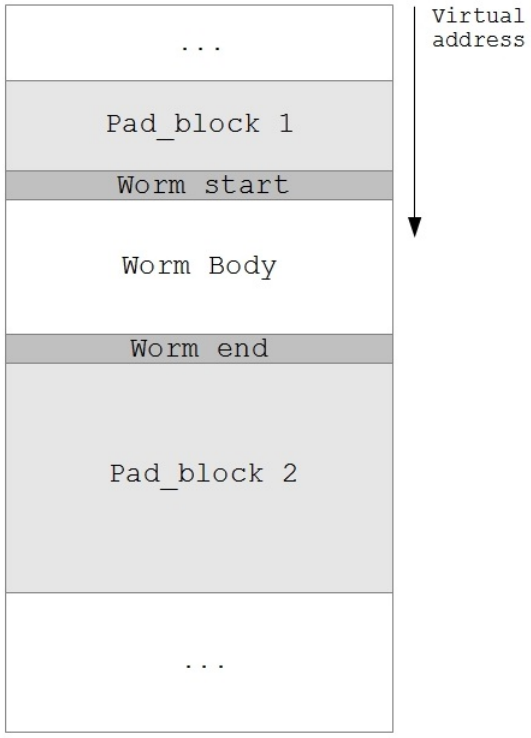

Figure 6.2: Metamorphic worm memory layout

6.3 Metamorphic techniques used

The worm uses two of the metamorphic techniques described in Chapter 3. Specifically, the worm uses equivalent instruction substitution and garbage code insertion.

6.3.1 Equivalent instruction substitution

The primary candidates for morphing are the `MOV` and the `XOR` instructions. The former because, it appears in abundance in binaries, and the latter because it is a usual candidate for substituting the `MOV` instruction and hence needs to be substituted back.

After the disassembler has disassembled the worm-portion of the worm's executable image, the instructions are scanned for possible equivalent instructions to be substituted, by the morphing engine. These instructions are substituted by equivalent instructions with a fixed probability. This is achieved using a substitution table as shown in Table 6.1.

Table 6.1: Equivalent instruction table

Instruction	Equivalent	Action
`0x48 0x89 0xc3`	`0x48 0x31 0xdb 0x48 0x01 0xc3`	NULL
`0x48 0x89 0xc1`	`0x48 0x31 0xc9 0x48 0x01 0xc1`	NULL
...		

The first column of each of the first two rows in Table 6.1 correspond to instructions `MOV %RAX,%RBX` and `MOV %RAX,%RCX` respectively. The second column of the first two rows correspond to instructions `XOR %RBX,%RBX; ADD %RAX,%RBX` and `XOR %RBX,%RCX; ADD %RAX,%RCX` respectively. The "Action" field in table is the address of a label inside the morphing function which performs actions specific to the instructions that were substituted. It is `NULL` if no specific action is necessary.

A complete list of equivalent instructions that are substituted by the morphing engine is in Appendix A.

6.3.2 Dead code insertion

Dead instruction insertion involves inserting instructions which do not result in any change in data, or the contents of general purpose registers. The sole purpose of adding these instructions is to increase the diversity of instructions.

However, the effect of such instructions on the `RFLAGS` register should be carefully considered as they might have adverse effects on control flow. Control flow instructions use bits in the `RFLAGS` register to decide which code path to take. If a dead code instruction which manipulates `RFLAGS` is inserted before a control flow instruction, it can have an adverse effect on the result of executing the next control flow instruction.

Examples for dead code instructions include `ADD $0x0,%RAX`, `SUB $0x0,%RBX`, `XOR $0x0,%RAX`, etc. The complete list of dead code instructions used by the worm is in Appendix B.

6.4 Functionality

A characteristic feature of metamorphic malware which carry their own engine is that the actual "payload" part of the worm will be much smaller than the overall size of the worm. In the case of our worm, the payload is benign and merely appends some text to the end of a temporary file. The worm's functionality is summarized by the Algorithm below:

```
Run ‘‘payload’’  // Actual intent of the worm

Open own binary image from disk by reading /proc/self/exe

Read worm data and book-keeping data from its ‘‘.data’’ section
```

```
Disassemble worm body excluding padding blocks:
    For each disassembled instruction:
        Add a unique label and store virtual address

Build symbol table:
    For each disassembled instruction INS:
        if control flow instruction
            add (INS.label, INS.target.label) to symbol table

Morph:
    Initialize substitution and dead code instruction tables
    For each disassembled instruction INS:
        if INS is a dead instruction:
            if INS is a control flow target
            (determined from symbol table)
                Change target to the next disassembled instruction
            Ignore dead instruction

        // Probability to insert dead code instruction = 0.33
        if adding new dead code instruction:
            choose dead code instruction randomly from table
                add new dead code instruction to morphed
                instruction list

        // Probability to morph an existing instruction
```

```
        // if possible, is 0.33
        If morphing:
            Check substitution table for a suitable entry
                if valid entry exists:
                    Add instructions to be substituted to
                    morphed instructions list
    Recalculate virtual addresses

Reassembly:
    Fix control flow in list of morphed instructions

Patch new binary:
    Replace padding blocks from benign binaries
    Create a new binary image
    Write binary to disk

Propogate:
    // This is for completeness only. A real worm uses exploits
    ‘‘rcp’’ or ‘‘scp’’ the new binary image to surrounding IP
    addresses
```

6.5 Implementation

The worm is implemented to work on Linux on the Intel x86_64 architecture. The programming language used to implement the worm is C. The compiler used to build the worm is GCC, version 4.6.2 build 20111027 and the resulting format of the

executable image of the worm is ELF64. This section explains implementation details such as layout of the worm's executable image, libraries used by the worm, etc.

6.5.1 Libraries used

The worm only links directly to libc and libdl. The libraries dynamically loaded during run-time are libbfd and libopdis. Libraries libc and libdl are part of the core of any Linux distribution. Libbfd is part of the GNU Binutils [13] package, and is usually found on most of the Linux distributions. Libopdis is an independent library licensed under GNU LGPL as of version 1.0.4 [16]. Libopdis extends the libopcodes library [13] by offering algorithms for linear and control-flow disassembly, instruction and operand objects that are suitable for analysis.

CHAPTER 7

Experiments

The effectiveness of the worm is evaluated using n-gram similarity [14], similarity using graph technique [18] and HMM based detection [30].

For the worm to be effective in evading signature based detection, the worm bodies in different generations of worm files must not be too similar to one another. At the same time, the worm files must be similar to benign files so that they are not easily distinguishable from benign files based on a similarity threshold [18].

An effective means of evading HMM based detection is to make the worms statistically similar to benign binaries. This achieved by using long sequences of instructions from benign executable files to fill the worm's padding blocks. This is in line with the HMM evasion technique [12] discussed in Section 5.4.

7.1 Test data

For each experiment, 100 generations of the worm are generated and 20 benign files are selected. The list of benign files, and their corresponding file IDs used in our test cases are shown in Table 7.1. From the 100 worms, 80 worms are chosen to train the HMM. The remaining 20 worms and benign files are scored using the trained HMM. The worm files in the test set are named MWOR_0, MWOR_1, ..., MWOR_19. The benign files are named BEN_0, BEN_1, ..., BEN_19.

The padding blocks of the MWOR files are randomly chosen blocks of code from one or more of the BEN files. Replacing the padding block randomly from the chosen benign file set in Table 7.1 is part of the worm's functionality.

Table 7.1: Mapping from Benign file ID to actual executable file

Benign file ID	Actual executable file
BEN_0	/usr/bin/as
BEN_1	/usr/bin/date
BEN_2	/usr/bin/dmesg
BEN_3	/usr/bin/file
BEN_4	/usr/bin/gcc
BEN_5	/usr/bin/size
BEN_6	/usr/bin/grep
BEN_7	/usr/bin/kill
BEN_8	/usr/bin/ld
BEN_9	/usr/bin/ldd
BEN_10	/usr/bin/mknod
BEN_11	/usr/bin/mount
BEN_12	/usr/bin/nasm
BEN_13	/usr/bin/nm
BEN_14	/usr/bin/objdump
BEN_14	/usr/bin/readelf
BEN_15	/usr/bin/rm
BEN_16	/usr/bin/sleep
BEN_17	/usr/bin/strip
BEN_18	/usr/bin/systemctl
BEN_19	/usr/bin/touch

As part of the experiment, both n-gram and graph technique are used to measure similarity between worms, between worms and benign files, and between benign files. An HMM classifier trained using worm files in the training set, is used to score benign files and worm files in the test set.

7.2 N-gram Similarity

The n-gram similarity technique [14] explained in Section 4.1 is used to measure similarity between opcode sequences extracted from different generations of the worm and from benign executable files. Since the objective here is to assess whether common signatures can be extracted from worm executable files, the

padding blocks are excluded from the assessment. When comparing these worm bodies to benign files, a representative sample sequence of instructions, of length equal to that of the worm body is chosen from the benign files.

Table 7.2 lists the similarity scores between consecutive generations of the worm. The average similarity is 19.09%. Similarly, Table 7.3 and Table 7.4 list the similarity scores between worms and benign files, and between worm files respectively. The average similarity between worms and benign files is 13.98%, while the average similarity between benign files is 26.35%.

Table 7.2: Similarity between MWOR files

File 1	File 2	Similarity	File 1	File 2	Similarity
MWOR_0	MWOR_1	0.209329	MWOR_10	MWOR_11	0.167043
MWOR_1	MWOR_2	0.139122	MWOR_11	MWOR_12	0.220303
MWOR_2	MWOR_3	0.199484	MWOR_12	MWOR_13	0.170271
MWOR_3	MWOR_4	0.21417	MWOR_13	MWOR_14	0.142834
MWOR_4	MWOR_5	0.222563	MWOR_14	MWOR_15	0.133796
MWOR_5	MWOR_6	0.526146	MWOR_15	MWOR_16	0.179309
MWOR_6	MWOR_7	0.206423	MWOR_16	MWOR_17	0.120562
MWOR_7	MWOR_8	0.225307	MWOR_17	MWOR_18	0.133473
MWOR_8	MWOR_9	0.133635	MWOR_18	MWOR_19	0.126372
MWOR_9	MWOR_10	0.15623			
Mean: 0.190862					
Variance: 0.007521					

The similarity between worm generations can also be visualized graphically as explained in Section 4. The similarity between the first and second generations of the worm is illustrated by the graph in Figure 7.1. Examples of the other graphs depicting the similarity between other consecutive pairs of worms are included in Appendix C.

The n-gram similarity between worms, is somewhat lower than the similarity between benign files. This can be attributed to the fact that, only the worm body is

Table 7.3: Similarity between MWOR and benign files

File 1	File 2	Similarity	File 1	File 2	Similarity
MWOR_0	BEN_0	0.203112	MWOR_10	BEN_10	0.234278
MWOR_1	BEN_1	0.118454	MWOR_11	BEN_11	0.130494
MWOR_2	BEN_2	0.178677	MWOR_12	BEN_12	0.125905
MWOR_3	BEN_3	0.105231	MWOR_13	BEN_13	0.141814
MWOR_4	BEN_4	0.17498	MWOR_14	BEN_14	0.068771
MWOR_5	BEN_5	0.121704	MWOR_15	BEN_15	0.050677
MWOR_6	BEN_6	0.133385	MWOR_16	BEN_16	0.123899
MWOR_7	BEN_7	0.169622	MWOR_17	BEN_17	0.118309
MWOR_8	BEN_8	0.152165	MWOR_18	BEN_18	0.181743
MWOR_9	BEN_9	0.13086	MWOR_19	BEN_19	0.131365
Mean: 0.139772					
Variance: 0.001732					

Table 7.4: Similarity between BEN files

File 1	File 2	Similarity	File 1	File 2	Similarity
BEN_0	BEN_1	0.223816	BEN_10	BEN_11	0.212141
BEN_1	BEN_2	0.176202	BEN_11	BEN_12	0.213142
BEN_2	BEN_3	0.309048	BEN_12	BEN_13	0.399767
BEN_3	BEN_4	0.248399	BEN_13	BEN_14	0.249333
BEN_4	BEN_5	0.196715	BEN_14	BEN_15	0.190627
BEN_5	BEN_6	0.227521	BEN_15	BEN_16	0.450179
BEN_6	BEN_7	0.199912	BEN_16	BEN_17	0.215491
BEN_7	BEN_8	0.240475	BEN_17	BEN_18	0.412275
BEN_8	BEN_9	0.248165	BEN_18	BEN_19	0.327552
BEN_9	BEN_10	0.265344			
Mean: 0.263479					
Variance: 0.006037					

considered for similarity tests, rather than the whole worm. The same is true in the case of worms versus benign files. The initial sections of the benign files, which are not morphed, result in a higher similarity between benign files, as opposed to worm versus benign files. However, low similarity of the worm body between different generations of the worm, helps evade signature based detection.

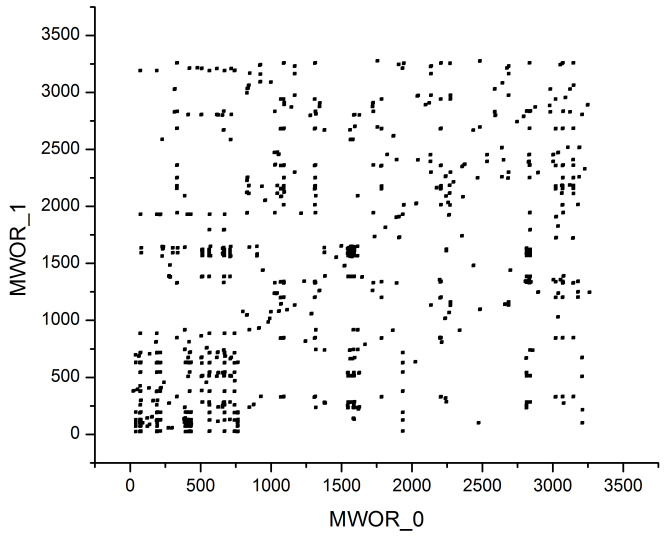

Figure 7.1: Similarity graph MWOR_0 vs MWOR_1

7.3 Similarity using graph technique

The graph technique to measure similarity [18] explained in Section 4.2 is used to measure similarity between complete worm executable files, including padding blocks. It is also used to compare the similarities between pairs benign files, and pairs of worm files and benign files.

Figure 7.2 illustrates the similarity between worm files, benign files, and worm and benign file pairs. The average similarity score for pairs of worm files is 0.592744. The average similarity score for pairs of worms and benign files is 0.565945. The average similarity score for pairs of benign files is 0.667563. As indicated by the similarity scores in Figure 7.2, it is clear that it is not possible to obtain a threshold that can be used to distinguish between the worm files and benign files.

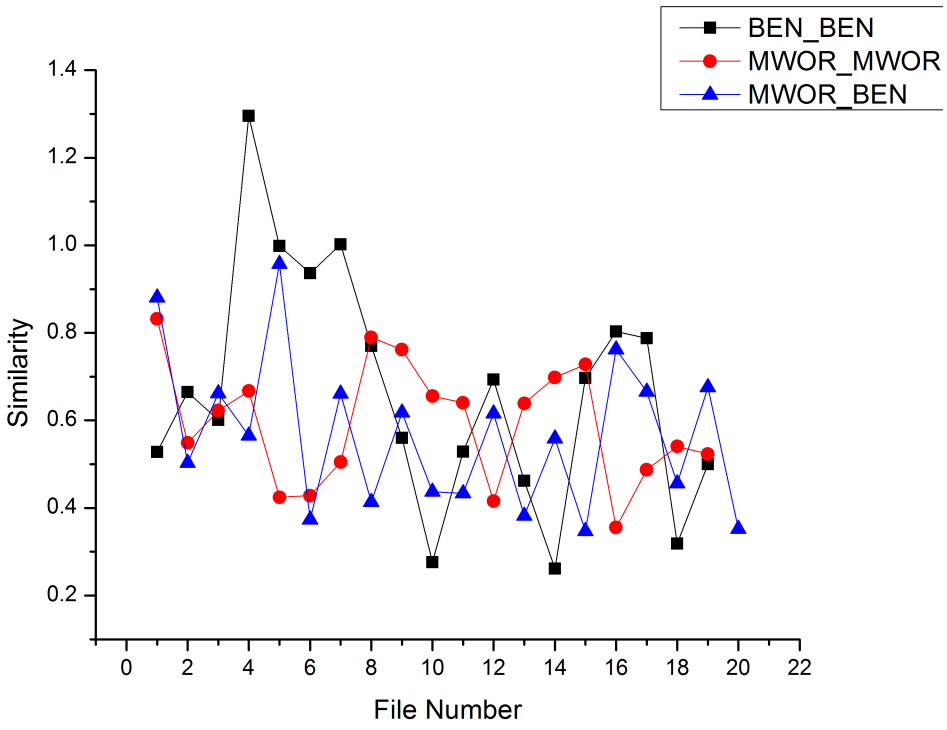

Figure 7.2: Similarity using graph technique

Table 7.5 lists the similarity scores between consecutive generations of the worm. Similarly, Table 7.6 and Table 7.7 list the similarity scores between worms and benign files, and between worm files respectively.

7.4 HMM

We now analyze the results of running the HMM detector on test data. As indicated by previous research [12, 30], the number of states in the HMM does not significantly impact the accuracy of classifier. Consequently, in this chapter, we will only consider HMMs with two hidden states. Additional results for HMMs with three states are presented in Appendix D.

The ratio of dead-code to worm-code, called the "padding-ratio", is the ratio

Table 7.5: Similarity using graph technique - MWOR files

File 1	File 2	Similarity	File 1	File 2	Similarity
MWOR_0	MWOR_1	0.831699	MWOR_10	MWOR_11	0.64038
MWOR_1	MWOR_2	0.548553	MWOR_11	MWOR_12	0.41586
MWOR_2	MWOR_3	0.622553	MWOR_12	MWOR_13	0.638838
MWOR_3	MWOR_4	0.667299	MWOR_13	MWOR_14	0.698381
MWOR_4	MWOR_5	0.424688	MWOR_14	MWOR_15	0.727753
MWOR_5	MWOR_6	0.428372	MWOR_15	MWOR_16	0.355481
MWOR_6	MWOR_7	0.504638	MWOR_16	MWOR_17	0.487425
MWOR_7	MWOR_8	0.78983	MWOR_17	MWOR_18	0.540405
MWOR_8	MWOR_9	0.761414	MWOR_18	MWOR_19	0.523221
MWOR_9	MWOR_10	0.655345			
Mean: 0.592744					
Variance: 0.017882					

Table 7.6: Similarity using graph technique - MWOR and BEN files

File 1	File 2	Similarity	File 1	File 2	Similarity
MWOR_0	BEN_0	0.880288	MWOR_10	BEN_10	0.433276
MWOR_1	BEN_1	0.502868	MWOR_11	BEN_11	0.615871
MWOR_2	BEN_2	0.661706	MWOR_12	BEN_12	0.381922
MWOR_3	BEN_3	0.565231	MWOR_13	BEN_13	0.558548
MWOR_4	BEN_4	0.957393	MWOR_14	BEN_14	0.34732
MWOR_5	BEN_5	0.373347	MWOR_15	BEN_15	0.761746
MWOR_6	BEN_6	0.660952	MWOR_16	BEN_16	0.665618
MWOR_7	BEN_7	0.413928	MWOR_17	BEN_17	0.455879
MWOR_8	BEN_8	0.618177	MWOR_18	BEN_18	0.675529
MWOR_9	BEN_9	0.437322	MWOR_19	BEN_19	0.35198
Mean: 0.565945					
Variance: 0.028684					

of number of dead code instructions in the worm to the number of instructions that correspond to the worm's functionality. For example, a worm with twice as much dead code as worm instructions will have a padding-ratio of 2.

We use an HMM with two states to score worms with padding ratio: 0.5, 1, 1.5, 2, 2.5, 3 and 4. Using these scores, we analyze the padding-ratio for which the HMM detector starts to falter.

Table 7.7: Similarity using graph technique - BEN files

File 1	File 2	Similarity	File 1	File 2	Similarity
BEN_0	BEN_1	0.527698	BEN_10	BEN_11	0.528616
BEN_1	BEN_2	0.665027	BEN_11	BEN_12	0.693015
BEN_2	BEN_3	0.601069	BEN_12	BEN_13	0.462323
BEN_3	BEN_4	1.295305	BEN_13	BEN_14	0.261346
BEN_4	BEN_5	0.998744	BEN_14	BEN_15	0.696717
BEN_5	BEN_6	0.936357	BEN_15	BEN_16	0.802703
BEN_6	BEN_7	1.00245	BEN_16	BEN_17	0.787582
BEN_7	BEN_8	0.770225	BEN_17	BEN_18	0.318344
BEN_8	BEN_9	0.559742	BEN_18	BEN_19	0.500018
BEN_9	BEN_10	0.276418			
Mean: 0.667563					
Variance: 0.068312					

Figure 7.3 shows the result of scoring worms and benign files that are part of the test data, using an HMM with two states. In this case, the generated worms contain half as much dead code as the instructions that constitute the core functionality of the worm.

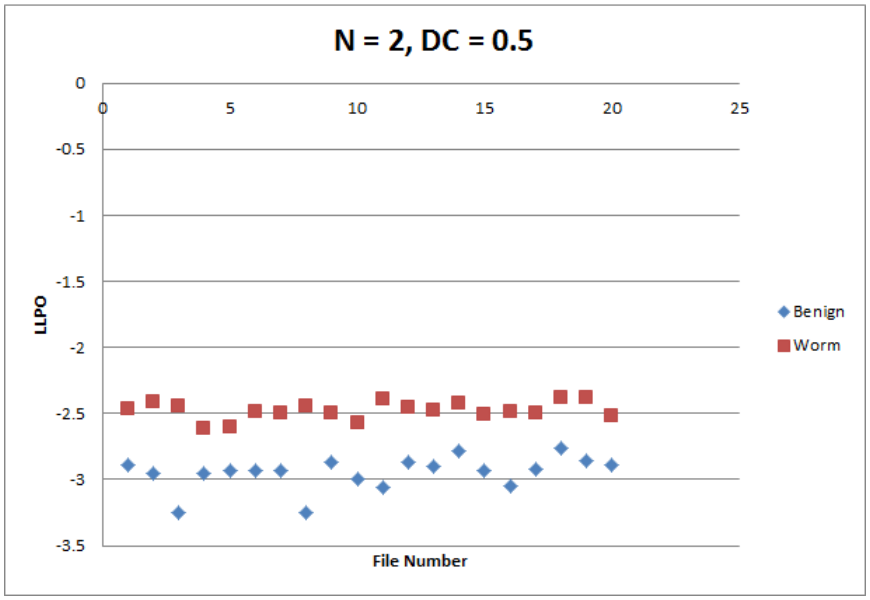

Figure 7.3: HMM with N = 2, padding-ratio: 0.5

The Log Likelihood Per Opcode (LLPO) scores for each of the MWOR files and BEN files are shown in Table 7.8.

Table 7.8: LLPO scores - padding-ratio: 0.5, N=2

File	LLPO	File	LLPO
BEN_0	-2.887989	MWOR_0	-2.462167
BEN_1	-2.953637	MWOR_1	-2.409931
BEN_2	-3.254619	MWOR_2	-2.447955
BEN_3	-2.955244	MWOR_3	-2.614725
BEN_4	-2.933179	MWOR_4	-2.604744
BEN_5	-2.93336	MWOR_5	-2.488717
BEN_6	-2.930817	MWOR_6	-2.491995
BEN_7	-3.248653	MWOR_7	-2.448593
BEN_8	-2.864609	MWOR_8	-2.501309
BEN_9	-2.993974	MWOR_9	-2.575815
BEN_10	-3.063865	MWOR_10	-2.388594
BEN_11	-2.868419	MWOR_11	-2.456711
BEN_12	-2.898413	MWOR_12	-2.471223
BEN_13	-2.784516	MWOR_13	-2.424988
BEN_14	-2.934695	MWOR_14	-2.502634
BEN_15	-3.044624	MWOR_15	-2.488669
BEN_16	-2.91717	MWOR_16	-2.493647
BEN_17	-2.758506	MWOR_17	-2.385327
BEN_18	-2.859302	MWOR_18	-2.381464
BEN_19	-2.890073	MWOR_19	-2.515267

Figure 7.4 shows the scores for a padding-ratio of 2.5. The increase in padding-ratio causes the LLPO scores of the worms to be closer to that of benign binaries.

The same test is repeated for other padding ratios. The results of the test are summarized in the ROC curve shown in Figure 7.5. The area under the curve (AUC) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one [5].

The AUC and standard error for each of the curves in the graph is shown in

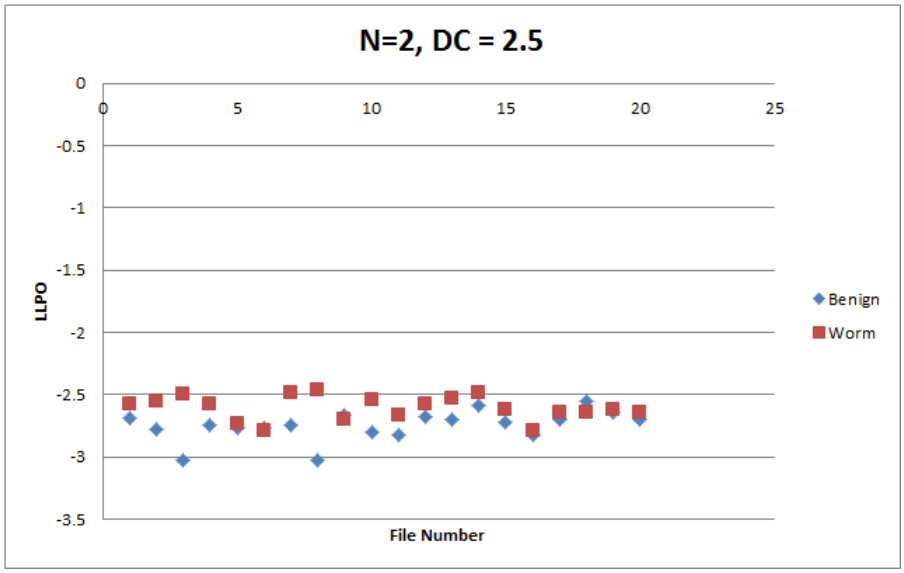

Figure 7.4: HMM with N = 2, padding-ratio 2.5

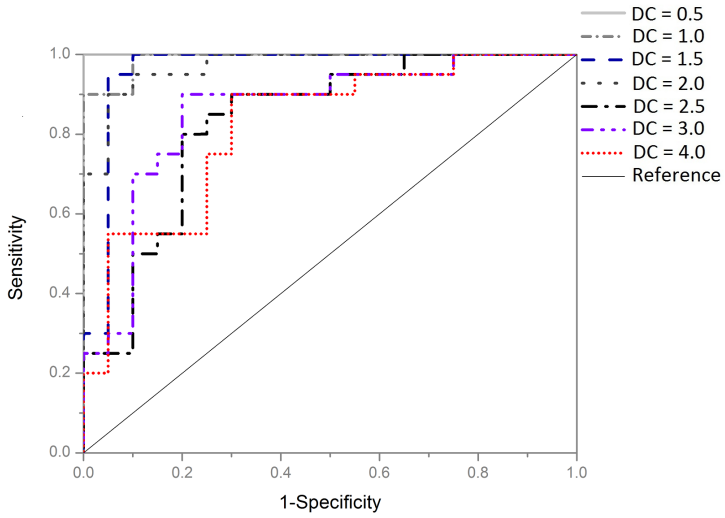

Figure 7.5: ROC Curves for different padding-ratios

Table 7.9. For a padding-ratio of 2.5, the area under the curve is 0.8325. At this point the it is safe to assume the HMM detector starts misclassifying files with some probability.

Table 7.9: ROC AUC statistics for different padding-ratios

Padding-ratio	AUC	Standard Error
0.5	1	0
1.0	0.99	0.0105
1.5	0.9625	0.03503
2.0	0.9725	0.02112
2.5	0.8325	0.06556
3.0	0.8575	0.06225
4.0	0.8225	0.06661

CHAPTER 8

Conclusion

The metamorphic worm described in this paper makes use of two morphing techniques: equivalent instruction substitution and dead instruction insertion. This is done in order to defeat signature-based detection. The worm also uses blocks of dead code from benign executable files to evade HMM detection. This also helps in making the worm executable files similar to benign executable files.

Results from the experiments show that it is not possible to obtain useful detection results using an HMM-based detector when the added dead code is more than 2.5 times the worm code. The HMM detector's performance is acceptable for padding-ratios up to 2.0. However, the probability of misclassification starts increasing for padding-ratios 2.5 and above, as indicated by the ROC curves.

We measured similarity using n-gram technique and graph technique between various combinations of benign executable files and worm files. The n-gram similarity between worm bodies in different generations of the worm, is sufficiently low to avoid extraction of a common signature, which can be used for signature-based detection.

The average similarity scores measured using graph technique, between worm file pairs, and between worm file and benign file pairs, are comparable to the similarity scores of benign file pairs. Therefore, the worms cannot be distinguished from benign files based on a similarity threshold.

CHAPTER 9

Future work

One of the main techniques used by the metamorphic worm described in this paper, is garbage instruction insertion. Use of garbage instructions is a proven technique to defeat the HMM detector [12]. However, the instructions are inserted randomly at feasible places and can be separated using more advanced dead-instruction finding tools. Further research can be done on such tools which can effectively detect functionally equivalent blocks of dead code.

Further research needs to be done on evaluating the effectiveness of HMM-based detectors, when compiler generated blocks of non-functional code are used for morphing.

The worm also uses simple morphing techniques. Further research can be carried out on morphing engines which use more advanced morphing techniques, while retaining the ability to be carried along with the malware.

LIST OF REFERENCES

[1] S. Attaluri, S. McGhee, and M. Stamp. Profile hidden markov models and metamorphic virus detection. *Journal in Computer Virology*, 5(2):151–169, 2009.

[2] J. Aycock. *Computer Viruses and Malware (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] P. Beaucamps. Advanced Metamorphic Techniques in Computer Viruses. In *International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE'07*, Venice, Italy, 2007.

[4] V. Bontchev. Possible virus attacks against integrity programs and how to prevent them. In *In Virus Bulletin Conference*, pages 131–141, 1992.

[5] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30:1145–1159, 1997.

[6] P. Desai. Towards an undetectable computer virus (2008). *Master's Projects. Paper 90*. `http://scholarworks.sjsu.edu/etd_projects/90`.

[7] T. M. Driller. Metamorphism in practice or "How I made MetaPHOR and what I've learnt", 2002. `http://vx.netlux.org/lib/vmd01.html`.

[8] E. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2:70–75, 2007.

[9] N. Idika and A. P. Mathur. A survey of malware detection techniques. *Purdue University*, page 48, 2007.

[10] E. Konstantinou, S. Wolthusen, and R. Holloway. Metamorphic virus: Analysis and detection, 2008.

[11] D. Lin. Hunting for undetectable metamorphic viruses (2009). *Master's Projects. Paper 18*. `http://scholarworks.sjsu.edu/etd_projects/18`.

[12] D. Lin and M. Stamp. Hunting for undetectable metamorphic viruses. *J. Comput. Virol.*, 7(3):201–214, Aug. 2011.

[13] F. Miller, A. Vandome, and J. McBrewster. *Gnu Binutils*. Alphascript Publishing, 2010.

[14] P. Mishra. Taxonomy of uniqueness transformations (2003). http://www.cs.sjsu.edu/faculty/stamp/students/FinalReport.doc.

[15] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, Jan. 1997.

[16] Opdis. libopcodes-based disassembler, 2010. http://mkfs.github.com/content/opdis/.

[17] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286, 1989.

[18] N. Runwal, R. M. Low, and M. Stamp. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8(1-2):37–52, 2012.

[19] Snakebyte. Next Generation Virus Construktion Kit (NGVCK), 2000. http://vx.netlux.org/vx.php?id=tn02.

[20] M. Stamp. *Information Security: Principles and Practice*. John Wiley & Sons, 2011.

[21] M. Stamp. A revealing introduction to hidden markov models, 2012. http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf.

[22] Symantec. What is the difference between viruses, worms, and trojans?, 2006. http://service1.symantec.com/support/nav.nsf/docid/1999041209131106.

[23] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[24] P. Szor and P. Ferrie. Hunting for metamorphic. In *In Virus Bulletin Conference*, pages 123–144, 2001.

[25] S. Venkatachalam. Detecting undetectable computer viruses (2010). *Master's Projects. Paper 156*. http://scholarworks.sjsu.edu/etd_projects/156.

[26] A. Venkatesan. Code obfuscation and virus detection (2008). *Master's Projects. Paper 116*. http://scholarworks.sjsu.edu/etd_projects/116.

[27] G. Wagener, R. State, and A. Dulaunoy. Malware behaviour analysis. *Journal in Computer Virology*, 4(4):279–287, 2008.

[28] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. The design space of metamorphic malware. In *Proceedings of the 2nd International Conference on Information Warfare*, 2007.

[29] W. Wong. Analysis and detection of metamorphic computer viruses (2006). *Master's Projects. Paper 153*. http://scholarworks.sjsu.edu/etd_projects/153.

[30] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[31] P. Zbitskiy. Code mutation techniques by means of formal grammars and automatons. *Journal in Computer Virology*, 5:199–207, 2009.

APPENDIX A

Equivalent instructions used by the worm

Table A.1: Equivalent instructions used by the worm

No.	Instruction	Equivalent instruction(s)

1	`MOV IMM, %REG`	`XOR %REG, %REG`
		`ADD IMM, %REG`
		OR
		`XOR %REG, %REG`
		`SUB -IMM, %REG`

2	`MOV %REG1, %REG2`	`XOR %REG2, %REG2` `ADD %REG1, %REG2`
3	`MOV %REG1, (%REG2)`	`MOV $0, (%REG2)` `ADD %REG1, (%REG2)`
4	`MOV IMM, (%REG)`	`MOV $0, (%REG2)` `ADD IMM, (%REG2)`
5	`XOR %REG, %REG`	`MOV $0, %REG`

APPENDIX B

Dead code instructions used by the worm

Table B.1: Dead code instructions used by the worm

No.	Instruction
1	`ADD $0, %RAX`
2	`ADD $0, %RBX`
3	`ADD $0, %RCX`
4	`ADD $0, %RDX`
5	`SUB $0, %RAX`
6	`SUB $0, %RBX`
7	`SUB $0, %RCX`
8	`SUB $0, %RDX`
9	`XOR $0, %RAX`
10	`XOR $0, %RBX`
11	`XOR $0, %RCX`
12	`XOR $0, %RDX`
13	`AND %RAX, %RAX`
14	`AND %RBX, %RBX`
15	`AND %RCX, %RCX`
16	`AND %RDX, %RDX`

APPENDIX C

Similarity Graphs

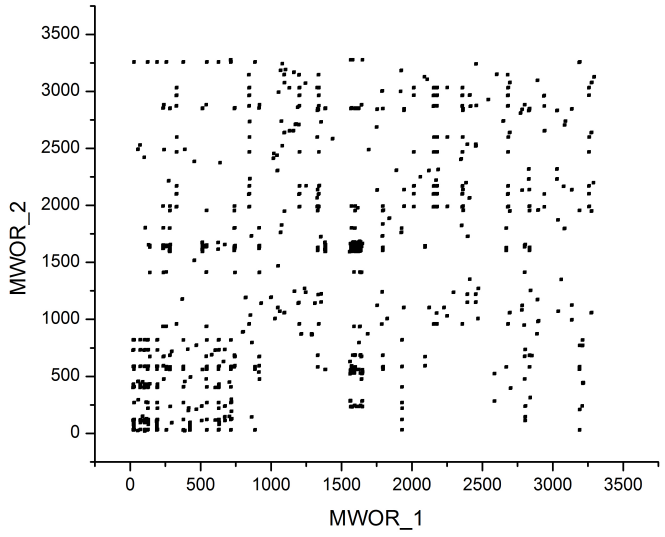

Figure C.1: Similarity graph - MWOR_1 vs MWOR_2

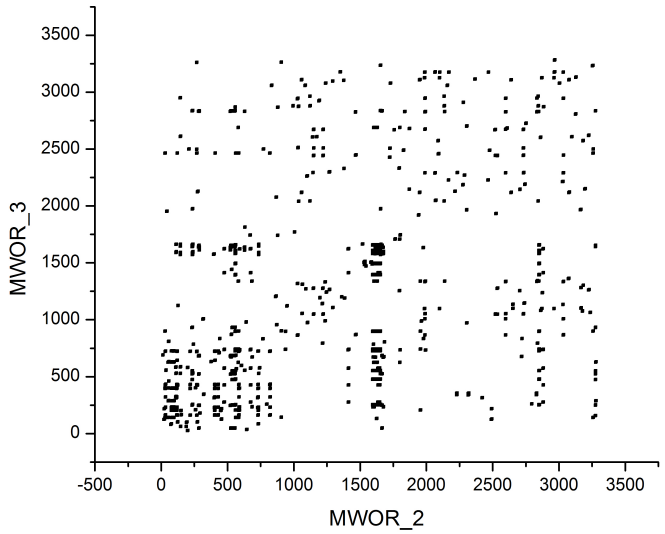

Figure C.2: Similarity graph - MWOR_2 vs MWOR_3

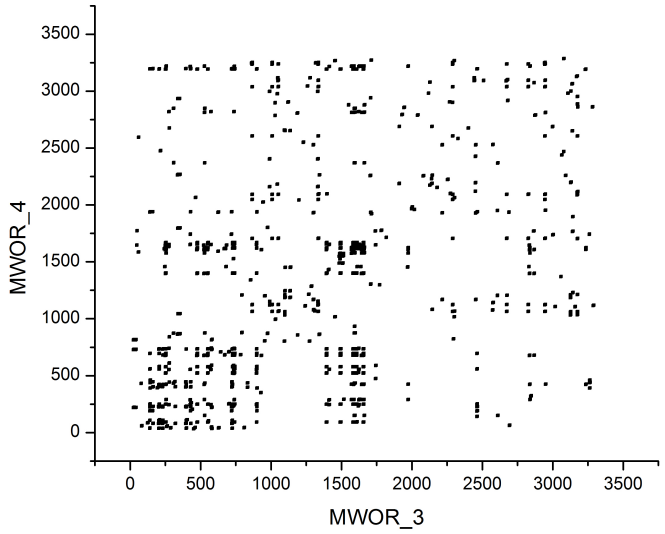

Figure C.3: Similarity graph - MWOR_3 vs MWOR_4

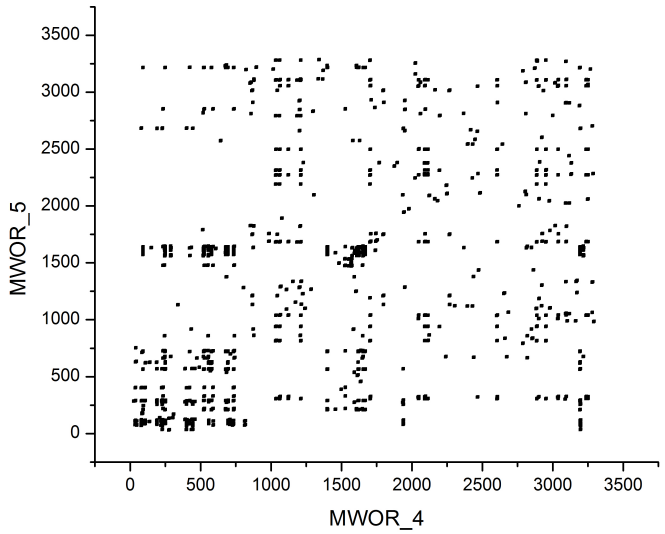

Figure C.4: Similarity graph - MWOR_4 vs MWOR_5

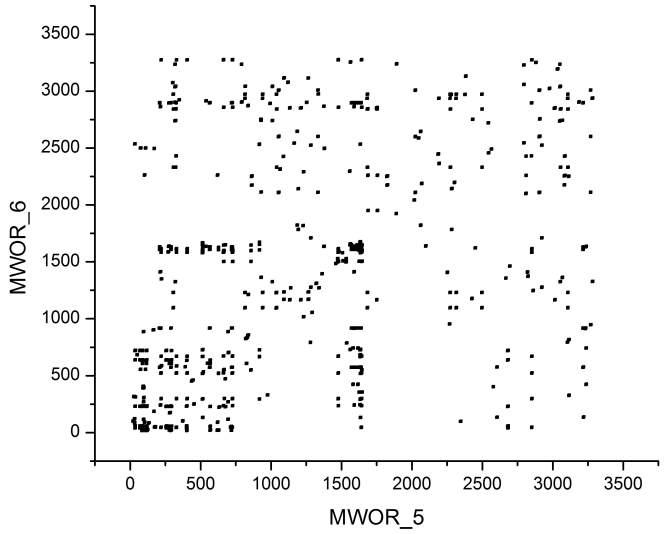

Figure C.5: Similarity graph - MWOR_5 vs MWOR_6

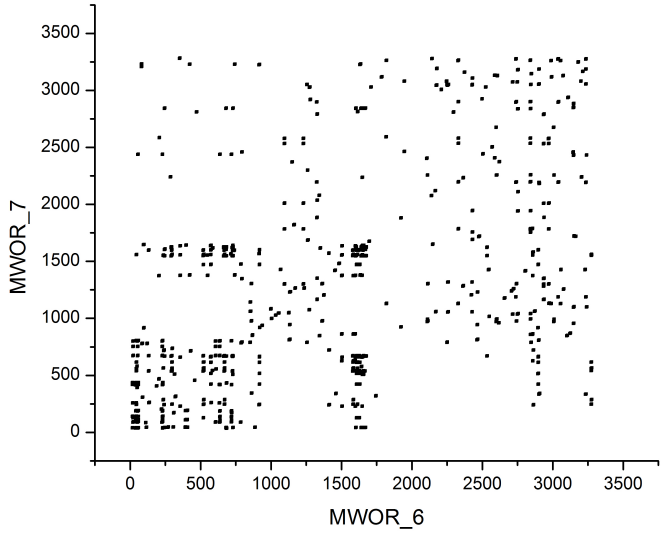

Figure C.6: Similarity graph - MWOR_6 vs MWOR_7

APPENDIX D

Additional HMM results

- HMM parameters: N = 3, M = 131
- Worm to padding ratio: 2.0
- LLPO scores: Table D.1
- Lowest MWOR file LLPO: -2.555177
- Highest BEN file LLPO: -2.479824
- Graph: Figure D.1

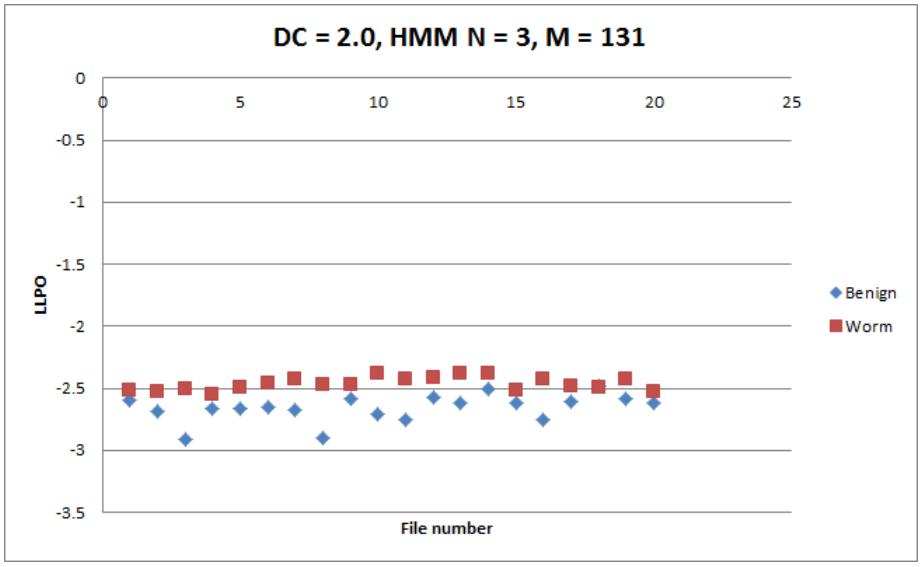

Figure D.1: HMM with N = 3, worm-padding ratio 2.0

(1) HMM parameters: N = 3, M = 129

(2) Worm to padding ratio: 3.0

(3) LLPO scores: Table D.2

(4) Lowest MWOR file LLPO: -2.541051

(5) Highest BEN file LLPO: -2.451643

(6) Graph: Figure D.2

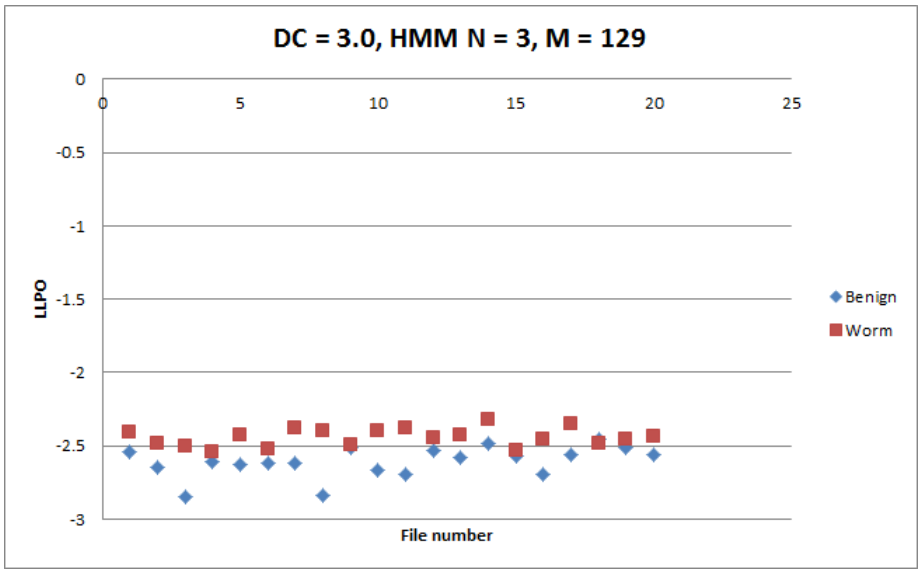

Figure D.2: HMM with N = 3, worm-padding ratio 3.0

(1) HMM parameters: N = 3, M = 131

(2) Worm to padding ratio: 4.0

(3) LLPO scores: Table D.3

(4) Lowest MWOR file LLPO: -2.718673

(5) Highest BEN file LLPO: -2.451643

(6) Graph: Figure D.3

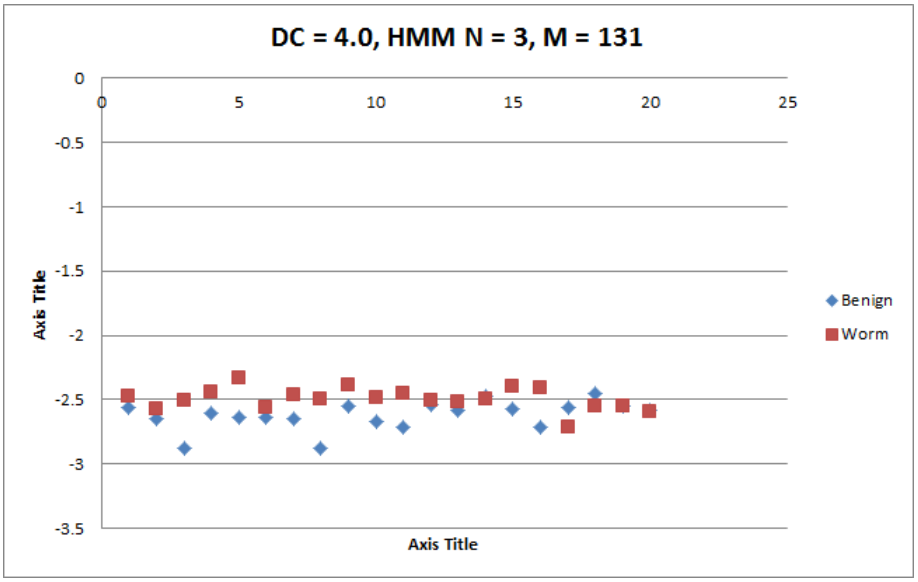

Figure D.3: HMM with N = 3, worm-padding ratio 4.0

Table D.1: LLPO scores - Worm-padding ratio: 2.0, N=3

File	LLPO	File	LLPO
File	LLPO	File	LLPO
BEN_0	-2.599386	MWOR_0	-2.512852
BEN_1	-2.684883	MWOR_1	-2.529261
BEN_2	-2.907455	MWOR_2	-2.499819
BEN_3	-2.657581	MWOR_3	-2.555177
BEN_4	-2.661491	MWOR_4	-2.492141
BEN_5	-2.655631	MWOR_5	-2.456433
BEN_6	-2.674456	MWOR_6	-2.425267
BEN_7	-2.900707	MWOR_7	-2.470162
BEN_8	-2.585162	MWOR_8	-2.47152
BEN_9	-2.709333	MWOR_9	-2.37629
BEN_10	-2.756476	MWOR_10	-2.427578
BEN_11	-2.573857	MWOR_11	-2.412057
BEN_12	-2.613086	MWOR_12	-2.383274
BEN_13	-2.508351	MWOR_13	-2.381367
BEN_14	-2.615704	MWOR_14	-2.511594
BEN_15	-2.748886	MWOR_15	-2.42333
BEN_16	-2.603163	MWOR_16	-2.477987
BEN_17	-2.479824	MWOR_17	-2.49241
BEN_18	-2.579598	MWOR_18	-2.427912
BEN_19	-2.613378	MWOR_19	-2.533057

Table D.2: LLPO scores - Worm-padding ratio: 3.0, N=3

File	LLPO	File	LLPO
BEN_0	-2.544323	MWOR_0	-2.402998
BEN_1	-2.64357	MWOR_1	-2.486666
BEN_2	-2.843443	MWOR_2	-2.504983
BEN_3	-2.608455	MWOR_3	-2.541051
BEN_4	-2.625266	MWOR_4	-2.4218
BEN_5	-2.619258	MWOR_5	-2.517544
BEN_6	-2.62047	MWOR_6	-2.382393
BEN_7	-2.84105	MWOR_7	-2.395489
BEN_8	-2.515048	MWOR_8	-2.49441
BEN_9	-2.669118	MWOR_9	-2.393952
BEN_10	-2.689387	MWOR_10	-2.380322
BEN_11	-2.526972	MWOR_11	-2.444773
BEN_12	-2.576147	MWOR_12	-2.421168
BEN_13	-2.478839	MWOR_13	-2.325241
BEN_14	-2.567223	MWOR_14	-2.527484
BEN_15	-2.690124	MWOR_15	-2.453113
BEN_16	-2.558649	MWOR_16	-2.353091
BEN_17	-2.451643	MWOR_17	-2.486506
BEN_18	-2.515152	MWOR_18	-2.45853
BEN_19	-2.560068	MWOR_19	-2.434684

Table D.3: LLPO scores - Worm-padding ratio: 4.0, N=3

File	LLPO	File	LLPO
BEN_0	-2.563521	MWOR_0	-2.478913
BEN_1	-2.652861	MWOR_1	-2.569835
BEN_2	-2.87828	MWOR_2	-2.502876
BEN_3	-2.606329	MWOR_3	-2.440894
BEN_4	-2.639205	MWOR_4	-2.326988
BEN_5	-2.633219	MWOR_5	-2.557039
BEN_6	-2.650461	MWOR_6	-2.465314
BEN_7	-2.878112	MWOR_7	-2.497504
BEN_8	-2.54517	MWOR_8	-2.391017
BEN_9	-2.665605	MWOR_9	-2.486907
BEN_10	-2.717854	MWOR_10	-2.448817
BEN_11	-2.543789	MWOR_11	-2.511273
BEN_12	-2.579349	MWOR_12	-2.522425
BEN_13	-2.477433	MWOR_13	-2.495827
BEN_14	-2.569425	MWOR_14	-2.392941
BEN_15	-2.714744	MWOR_15	-2.411323
BEN_16	-2.562329	MWOR_16	-2.718673
BEN_17	-2.455398	MWOR_17	-2.545033
BEN_18	-2.546387	MWOR_18	-2.552821
BEN_19	-2.581043	MWOR_19	-2.588134

APPENDIX E

Selected HMM Models

Table E.1: HMM matrices, N = 2 Worm-padding ratio: 2.0

HMM Parameters		
N = 2, M = 131, T = 790441		
π :		
	0.0000000000	1.0000000000
A :		
	0.8874284727	0.1125715273
	0.0616794013	0.9383205987
B :		
adc	0.0000962463	0.0000060150
add	0.0223048733	0.0987331347
addsd	0.0001501143	0.0000000000
addss	0.0001358177	0.0000000000
and	0.0117147309	0.0357255629
bsf	0.0000500381	0.0000000000
bsr	0.0000030630	0.0000061549
bswap	0.0000000000	0.0000019583
bt	0.0001501143	0.0000000000
call	0.0497101214	0.0723558852
cdqe	0.0001440348	0.0065772642
clc	0.0000000000	0.0000058748
cld	0.0000000000	0.0000039166
cmova	0.0001165807	0.0000908293
cmovae	0.0002358939	0.0000000000
cmovb	0.0002823578	0.0000000000
cmovbe	0.0001608367	0.0000000000
cmove	0.0035955949	0.0000000000
cmovg	0.0001858558	0.0000000000
cmovge	0.0000929279	0.0000000000
cmovl	0.0000655234	0.0000013070
cmovle	0.0000857796	0.0000000000
cmovne	0.0018192423	0.0000000000
cmovns	0.0000718093	0.0000331119
cmovs	0.0001574846	0.0000018366
cmp	0.1287337339	0.0000000000
cpuid	0.0000986223	0.0000223376
cvtsi2sd	0.0002358939	0.0000000000
cvtsi2ss	0.0002501905	0.0000000000
cvttsd2si	0.0000142966	0.0000000000
cvttss2si	0.0000786313	0.0000000000

Continued on Next Page...

Table E.1 – Continued

HMM Parameters		
cwde	0.0000000000	0.0000567900
dec	0.0000000000	0.0002898249
div	0.0003694867	0.0002401287
divsd	0.0000714830	0.0000000000
divss	0.0000107224	0.0000000000
enter	0.0000000000	0.0000391655
fcmovnb	0.0000000000	0.0000058748
fild	0.0000008404	0.0000112892
fisttp	0.0000084592	0.0000345307
fld	0.0000142966	0.0000000000
fmul	0.0000000000	0.0000176245
fstp	0.0000076758	0.0000075441
fsub	0.0000000000	0.0000019583
fucomip	0.0000071483	0.0000000000
fxch	0.0000071483	0.0000000000
hlt	0.0000000000	0.0000391655
icebp	0.0000000000	0.0000372072
idiv	0.0000000000	0.0001351211
imul	0.0007768793	0.0012937144
in	0.0000144056	0.0000371475
inc	0.0001104452	0.000088311
ja	0.0083456402	0.0000000000
jae	0.0042246453	0.0000000000
jb	0.0049823651	0.0000000000
jbe	0.0064549149	0.0000000000
je	0.1199091576	0.0000000000
jg	0.0046928589	0.0000000000
jge	0.0009828912	0.0000000000
jl	0.0025805363	0.0000000000
jle	0.0076379585	0.0000000000
jmp	0.0383346104	0.0411913252
jne	0.0723836853	0.0000000000
jnp	0.0000107224	0.0000000000
jns	0.0011747288	0.0000025974
jo	0.0000030633	0.0000120296
jp	0.0000107224	0.0000000000
js	0.0035705758	0.0000000000
ldmxcsr	0.0000000000	0.0000019583
lea	0.0212980591	0.0262488989
leave	0.0000128848	0.0005765067
lock	0.0000000000	0.0000744145
lods	0.0000000000	0.0000019583
loop	0.0000008308	0.0000073779
loope	0.0001644109	0.0000000000
loopne	0.0000026619	0.0000690395
mov	0.2512854820	0.4841948314
movabs	0.0009057861	0.0002184908
movapd	0.0000536122	0.0000000000
movaps	0.0000019070	0.0019023996

Continued on Next Page...

Table E.1 – Continued

HMM Parameters		
movs	0.0000000000	0.0000469986
movsd	0.0007255524	0.0000000000
movss	0.0000929279	0.0000000000
movsx	0.0016784643	0.0004472465
movsxd	0.0014446895	0.0218715853
movzx	0.0252677456	0.0034297606
mul	0.0000422668	0.0000473400
mulsd	0.0000643347	0.0000000000
mulss	0.0001143728	0.0000000000
neg	0.0004151083	0.0003600451
nop	0.0265848542	0.0158442338
not	0.0005018279	0.0011761312
or	0.0046880928	0.0027716139
out	0.0000030457	0.0000218306
pop	0.0000000000	0.0222518925
push	0.0003771502	0.0203160941
rep	0.0003791506	0.0013158023
repnz	0.0000408011	0.0006473755
repz	0.0043506278	0.0000014721
ret	0.0007408776	0.0118332996
retf	0.0000000000	0.0000019583
rex	0.0000060341	0.0000064853
rol	0.0000000000	0.0004797777
ror	0.0000000000	0.0003485732
sar	0.0000172454	0.0019762433
sbb	0.0013464675	0.0000024991
seta	0.0008113320	0.0000000000
setae	0.0000178707	0.0000000000
setb	0.0007219783	0.0000000000
setbe	0.0000679088	0.0000000000
sete	0.0032238833	0.0000000000
setg	0.0001572626	0.0000000000
setge	0.0000571864	0.0000000000
setl	0.0000571864	0.0000000000
setle	0.0000428898	0.0000000000
setne	0.0030308792	0.0000000000
sets	0.0000071483	0.0000000000
shl	0.0000581400	0.0207474139
shr	0.0018454119	0.0026822080
sldt	0.0000035741	0.0000000000
stmxcsr	0.0000000000	0.0000019583
stos	0.0001143245	0.0000744409
sub	0.0106511885	0.0303884087
subsd	0.0000071483	0.0000000000
subss	0.0000393156	0.0000000000
test	0.1085183416	0.0000000000
ucomisd	0.0000428898	0.0000000000
ucomiss	0.0001965782	0.0000000000
xchg	0.0030175557	0.0010706439

Continued on Next Page...

Table E.1 – Continued		
HMM Parameters		
xor	0.0272213666	0.0694342484
xorpd	0.0000428898	0.0000000000

Table E.2: HMM matrices, N = 3 Worm-padding ratio: 2.0

HMM Parameters			
N = 3, M = 131, T = 790441			
π :			
	0.0000000000	0.0000000000	1.000000000
A :			
	0.8612819278	0.1372622966	0.0014557756
	0.0000012331	0.7801870820	0.2198116849
	0.5001390860	0.4998609140	0.0000000000
B :			
adc	0.0000828834	0.0000105444	0.0000000000
add	0.0234782893	0.0912150386	0.1570662990
addsd	0.0001345659	0.0000000000	0.0000000000
addss	0.0001217501	0.0000000000	0.0000000000
and	0.0085665320	0.0350832625	0.0589494082
bsf	0.0000000000	0.0000000000	0.0001617223
bsr	0.0000128158	0.0000000000	0.0000000000
bswap	0.0000000000	0.0000025526	0.0000000000
bt	0.0000580231	0.0000000000	0.0002759692
call	0.0104884193	0.0033310490	0.5345897198
cdqe	0.0001669943	0.0083739393	0.0007661987
clc	0.0000000000	0.0000076578	0.0000000000
cld	0.0000000000	0.0000051052	0.0000000000
cmova	0.0002531120	0.0000000000	0.0000000000
cmovae	0.0002070472	0.0000035163	0.0000000000
cmovb	0.0002531120	0.0000000000	0.0000000000
cmovbe	0.0000776469	0.0000530054	0.0000000000
cmove	0.0029079862	0.0002511113	0.0000000000
cmovg	0.0001323610	0.0000272827	0.0000000000
cmovge	0.0000833027	0.0000000000	0.0000000000
cmovl	0.0000532171	0.0000061011	0.0000000000
cmovle	0.0000707234	0.0000049168	0.0000000000
cmovne	0.0014605853	0.0001356193	0.0000000000
cmovns	0.0000701355	0.0000385690	0.0000000000
cmovs	0.0001431256	0.0000008383	0.0000000000
cmp	0.1070266756	0.0000000000	0.0301889020
cpuid	0.0000287684	0.0000000000	0.0003467898
cvtsi2sd	0.0002114607	0.0000000000	0.0000000000
cvtsi2ss	0.0002242765	0.0000000000	0.0000000000
cvttsd2si	0.0000128158	0.0000000000	0.0000000000
cvttss2si	0.0000704869	0.0000000000	0.0000000000
cwde	0.0000000000	0.0000740255	0.0000000000
dec	0.0000099010	0.0003105508	0.0002685659
div	0.0000336322	0.0000000000	0.0024894014
divsd	0.0000582197	0.0000046682	0.0000000000
divss	0.0000096118	0.0000000000	0.0000000000
enter	0.0000000000	0.0000510520	0.0000000000
fcmovnb	0.0000000000	0.0000076578	0.0000000000

Continued on Next Page...

Table E.2 – Continued

HMM Parameters			
fild	0.0000000000	0.0000153156	0.0000000000
fisttp	0.0000010288	0.0000152854	0.0001581495
fld	0.0000128158	0.0000000000	0.0000000000
fmul	0.0000000000	0.0000229734	0.0000000000
fstp	0.0000071958	0.0000095827	0.0000000000
fsub	0.0000032039	0.0000000000	0.0000000000
fucomip	0.0000064079	0.0000000000	0.0000000000
fxch	0.0000064079	0.0000000000	0.0000000000
hlt	0.0000640790	0.0000000000	0.0000000000
icebp	0.0000000000	0.0000484994	0.0000000000
idiv	0.0000000000	0.0000000000	0.0007970598
imul	0.0006132673	0.0015983244	0.0006981240
in	0.0000167522	0.0000453633	0.0000000000
inc	0.0000835692	0.0000559938	0.0003232233
ja	0.0074812232	0.0000000000	0.0000000000
jae	0.0037870689	0.0000000000	0.0000000000
jb	0.0044663063	0.0000000000	0.0000000000
jbe	0.0057863337	0.0000000000	0.0000000000
je	0.1074893184	0.0000000000	0.0000000000
jg	0.0042067863	0.0000000000	0.0000000000
jge	0.0008466299	0.0000274515	0.0000000000
jl	0.0023132519	0.0000000000	0.0000000000
jle	0.0068468411	0.0000000000	0.0000000000
jmp	0.0656506094	0.0116690931	0.0773728756
jne	0.0648863953	0.0000000000	0.0000000000
jnp	0.0000096118	0.0000000000	0.0000000000
jns	0.0007665269	0.0002316631	0.0000000000
jo	0.0000001864	0.0000177197	0.0000000000
jp	0.0000096118	0.0000000000	0.0000000000
js	0.0032007460	0.0000000000	0.0000000000
ldmxcsr	0.0000032039	0.0000000000	0.0000000000
lea	0.0159713702	0.0362829569	0.0018943892
leave	0.0005994922	0.0000000000	0.0012809520
lock	0.0000051874	0.0000397825	0.0002402252
lods	0.0000000000	0.0000025526	0.0000000000
loop	0.0000000000	0.0000102104	0.0000000000
loope	0.0000000000	0.0000000000	0.0005313732
loopne	0.0000042639	0.0000884966	0.0000000000
mov	0.2228118157	0.6235781612	0.0430640170
movabs	0.0007338803	0.0002167839	0.0005893452
movapd	0.0000480592	0.0000000000	0.0000000000
movaps	0.0031142394	0.0000000000	0.0000000000
movs	0.0000000000	0.0000002543	0.0002760874
movsd	0.0002950718	0.0002830931	0.0000000000
movss	0.0000833027	0.0000000000	0.0000000000
movsx	0.0013002940	0.0007457659	0.0000000000
movsxd	0.0012559968	0.0284969571	0.0001975253
movzx	0.0224119768	0.0019835049	0.0121157674
mul	0.0000000000	0.0000918937	0.0000000000

Continued on Next Page...

Table E.2 – Continued

HMM Parameters			
mulsd	0.0000576711	0.0000000000	0.0000000000
mulss	0.0001025264	0.0000000000	0.0000000000
neg	0.0004132444	0.0003309115	0.0004780444
nop	0.0487322115	0.0008141748	0.0000000000
not	0.0001655860	0.0010675424	0.0031316466
or	0.0051077514	0.0014539539	0.0065058417
out	0.0000000000	0.0000306312	0.0000000000
pop	0.0364064838	0.0000000000	0.0000000000
push	0.0335773960	0.0000000000	0.0000000000
rep	0.0002909919	0.0012972864	0.0020672262
repnz	0.0000000000	0.0008088005	0.0002904841
repz	0.0017800704	0.0000000000	0.0076519334
ret	0.0200243835	0.0000002422	0.0000000000
retf	0.0000000000	0.0000025526	0.0000000000
rex	0.0000030712	0.0000000000	0.0000466851
rol	0.0000042868	0.0005870880	0.0001578656
ror	0.0000031970	0.0003717528	0.0003623200
sar	0.0000170744	0.0023766679	0.0008963376
sbb	0.0007386885	0.0003763670	0.0000000000
seta	0.0007272966	0.0000000000	0.0000000000
setae	0.0000160197	0.0000000000	0.0000000000
setb	0.0006471979	0.0000000000	0.0000000000
setbe	0.0000608750	0.0000000000	0.0000000000
sete	0.0028381179	0.0000413051	0.0000000000
setg	0.0001409738	0.0000000000	0.0000000000
setge	0.0000512632	0.0000000000	0.0000000000
setl	0.0000512632	0.0000000000	0.0000000000
setle	0.0000384474	0.0000000000	0.0000000000
setne	0.0027169496	0.0000000000	0.0000000000
sets	0.0000064079	0.0000000000	0.0000000000
shl	0.0002439713	0.0264287036	0.0020933850
shr	0.0004962522	0.0025269726	0.0085614955
sldt	0.0000032039	0.0000000000	0.0000000000
stmxcsr	0.0000032039	0.0000000000	0.0000000000
stos	0.0001597281	0.0000000000	0.0002327241
sub	0.0128469226	0.0288366179	0.0368650112
subsd	0.0000064079	0.0000000000	0.0000000000
subss	0.0000352434	0.0000000000	0.0000000000
test	0.0972783254	0.0000000000	0.0000000158
ucomisd	0.0000384474	0.0000000000	0.0000000000
ucomiss	0.0001762172	0.0000000000	0.0000000000
xchg	0.0043416162	0.0000916833	0.0000000000
xor	0.0259037597	0.0879810038	0.0060168932
xorpd	0.0000384474	0.0000000000	0.0000000000

Table E.3: HMM matrices, N = 2 Worm-padding ratio: 3.0

HMM Parameters		
N = 2, M = 129, T = 1054041		
π :		
	1.0000000000	0.0000000000
A :		
	0.8798421953	0.1201578047
	0.0679607327	0.9320392673
B :		
adc	0.0000623653	0.0000048299
add	0.0227135425	0.0838085640
addsd	0.0001785745	0.0000000000
addss	0.0001024178	0.0000000000
and	0.0101517075	0.0283808283
bsf	0.0000105044	0.0000000000
bsr	0.0000000000	0.0000118827
bswap	0.0000000000	0.0000089120
bt	0.0002074616	0.0000000000
call	0.0507969274	0.0820009550
cdqe	0.0002427649	0.0051029690
clc	0.0000000000	0.0000044560
cld	0.0000000000	0.0000029707
cmova	0.0001001587	0.0001899159
cmovae	0.0002337226	0.0000000000
cmovb	0.0002704879	0.0000000000
cmovbe	0.0003256359	0.0000000000
cmove	0.0036213866	0.0000000000
cmovg	0.0001654441	0.0000000000
cmovge	0.0001129221	0.0000000000
cmovl	0.0000459651	0.0000022233
cmovle	0.0001127475	0.0000015841
cmovne	0.0019932070	0.0000000000
cmovns	0.0001055527	0.0000115949
cmovs	0.0001024178	0.0000000000
cmp	0.1283557032	0.0000000000
cpuid	0.0000394959	0.0000103383
cvtsi2sd	0.0002888706	0.0000000000
cvtsi2ss	0.0001864528	0.0000000000
cvttsd2si	0.0000105044	0.0000000000
cvttss2si	0.0000604002	0.0000000000
cwde	0.0000000000	0.0000430749
dec	0.0000000000	0.0002257716
div	0.0003908249	0.0003998185
divsd	0.0000919134	0.0000000000
divss	0.0000078783	0.0000000000
enter	0.0000000000	0.0000297068
fcmovnb	0.0000000000	0.0000044560
fild	0.0000000000	0.0000089120

Continued on Next Page...

Table E.3 – Continued

HMM Parameters		
fisttp	0.0000056207	0.0000265277
fld	0.0000000000	0.0000059414
fmul	0.0000000000	0.0000133681
fstp	0.0000000000	0.0000089120
fsub	0.0000000000	0.0000014853
fucomip	0.0000000000	0.0000029707
fxch	0.0000000000	0.0000029707
hlt	0.0000000000	0.0000519869
icebp	0.0000000000	0.0000282215
idiv	0.0000000000	0.0000891204
imul	0.0005759725	0.0010778717
in	0.0000104706	0.0000282406
inc	0.0001209957	0.0000622738
ja	0.0092202237	0.0000000000
jae	0.0052837055	0.0000000000
jb	0.0059297252	0.0000000000
jbe	0.0069827898	0.0000000000
je	0.1258372769	0.0000000000
jg	0.0040993361	0.0000000000
jge	0.0011607345	0.0000000000
jl	0.0021849120	0.0000000000
jle	0.0070221812	0.0000000000
jmp	0.0264328267	0.0563145109
jne	0.0731735441	0.0000000000
jnp	0.0000052522	0.0000000000
jns	0.0011003343	0.0000000000
jo	0.0000022305	0.0000091358
jp	0.0000131305	0.0000000000
js	0.0039522748	0.0000000000
lea	0.0261441439	0.0277409236
leave	0.0000073980	0.0004429029
lock	0.0000000000	0.0000564429
lods	0.0000000000	0.0000014853
loop	0.0000002381	0.0000058067
loope	0.0001208004	0.0000000000
loopne	0.0000016139	0.0000525594
mov	0.2556700975	0.4798947462
movabs	0.0008012547	0.0003503732
movapd	0.0000656524	0.0000000000
movaps	0.0000000000	0.0013724539
movs	0.0000000000	0.0000415895
movsd	0.0001582972	0.0003471558
movss	0.0000682785	0.0000000000
movsx	0.0019355024	0.0004529893
movsxd	0.0015610365	0.0177372853
movzx	0.0241815988	0.0039893369
mul	0.0000261068	0.0001545625
mulsd	0.0000814090	0.0000000000
mulss	0.0000840351	0.0000000000

Continued on Next Page...

Table E.3 – Continued		
HMM Parameters		
neg	0.0002820293	0.0005088850
nop	0.0204647588	0.0207444864
not	0.0003899404	0.0011400180
or	0.0034646895	0.0030073217
out	0.0000024414	0.0000164432
pop	0.0000000000	0.0241991554
push	0.0002968426	0.0227583229
rep	0.0005232092	0.0010483014
repnz	0.0000000000	0.0005867092
repz	0.0054341312	0.0000144358
ret	0.0001471499	0.0123535208
retf	0.0000000000	0.0000014853
rex	0.0000000050	0.0000074239
rol	0.0000000000	0.0004010417
ror	0.0000000000	0.0002643905
sar	0.0000069096	0.0016210536
sbb	0.0018810120	0.0000129568
seta	0.0010425602	0.0000000000
setae	0.0000262610	0.0000000000
setb	0.0009322641	0.0000000000
setbe	0.0000446436	0.0000000000
sete	0.0033640292	0.0000000000
setg	0.0001549397	0.0000000000
setge	0.0000315132	0.0000000000
setl	0.0000367653	0.0000000000
setle	0.0000236349	0.0000000000
setne	0.0024370173	0.0000000000
sets	0.0000078783	0.0000000000
shl	0.0000661565	0.0172875844
shr	0.0019120854	0.0025902700
sldt	0.0000026261	0.0000000000
stos	0.0000490186	0.0000287176
sub	0.0099482600	0.0266941826
subsd	0.0000052522	0.0000000000
subss	0.0000288871	0.0000000000
test	0.1157478154	0.0000000000
ucomisd	0.0000315132	0.0000000000
ucomiss	0.0001444353	0.0000000000
xchg	0.0025051041	0.0014601974
xor	0.0234487579	0.0726254926
xorpd	0.0000065047	0.0000096890

Table E.4: HMM matrices, N = 3 Worm-padding ratio: 3.0

HMM Parameters			
N = 3, M = 129, T = 1054041			
π :			
	1.0000000000	0.0000000000	0.0000000000
A :			
	0.8401558381	0.1184121543	0.0414320076
	0.0871060187	0.8805817832	0.0323121981
	0.0544288643	0.0613836061	0.8841875296
B :			
adc	0.0000686927	0.0000000000	0.0000149139
add	0.0280699091	0.0174225388	0.1893056150
addsd	0.0002008214	0.0000000000	0.0000000000
addss	0.0001151770	0.0000000000	0.0000000000
and	0.0112359283	0.0138288670	0.0508137537
bsf	0.0000118130	0.0000000000	0.0000000000
bsr	0.0000000000	0.0000000000	0.0000319014
bswap	0.0000000000	0.0000129127	0.0000000000
bt	0.0002333073	0.0000000000	0.0000000000
call	0.0325833308	0.1354266373	0.0023525599
cdqe	0.0002378837	0.0047539025	0.0049387804
clc	0.0000000000	0.0000000000	0.0000119630
cld	0.0000000000	0.0000000000	0.0000079753
cmova	0.0001009336	0.0000000000	0.0005256663
cmovae	0.0002628398	0.0000000000	0.0000000000
cmovb	0.0003041854	0.0000000000	0.0000000000
cmovbe	0.0003276320	0.0000281084	0.0000000000
cmove	0.0040670800	0.0000000010	0.0000073716
cmovg	0.0001792889	0.0000049308	0.0000000000
cmovge	0.0001269900	0.0000000000	0.0000000000
cmovl	0.0000497627	0.0000046268	0.0000000000
cmovle	0.0001186605	0.0000000000	0.0000152347
cmovne	0.0021607049	0.0000588935	0.0000000000
cmovns	0.0001138984	0.0000012441	0.0000353103
cmovs	0.0001151770	0.0000000000	0.0000000000
cmp	0.1443463177	0.0000000000	0.0000000000
cpuid	0.0000336036	0.0000163412	0.0000120764
cvtsi2sd	0.0003248582	0.0000000000	0.0000000000
cvtsi2ss	0.0002096812	0.0000000000	0.0000000000
cvttsd2si	0.0000118130	0.0000000000	0.0000000000
cvttss2si	0.0000679249	0.0000000000	0.0000000000
cwde	0.0000000000	0.0000025685	0.0001108831
dec	0.0000485819	0.0000246204	0.0004949083
div	0.0003450119	0.0003707240	0.0005140745
divsd	0.0001033640	0.0000000000	0.0000000000
divss	0.0000088598	0.0000000000	0.0000000000
enter	0.0000000000	0.0000000000	0.0000797534
fcmovnb	0.0000000000	0.0000023113	0.0000076803

Continued on Next Page...

Table E.4 – Continued

HMM Parameters			
fild	0.0000000000	0.0000000000	0.0000239260
fisttp	0.0000088875	0.0000000000	0.0000677530
fld	0.0000118130	0.0000000000	0.0000000000
fmul	0.0000000000	0.0000017796	0.0000325916
fstp	0.0000065920	0.0000000000	0.0000150251
fsub	0.0000000000	0.0000000000	0.0000039877
fucomip	0.0000059065	0.0000000000	0.0000000000
fxch	0.0000059065	0.0000000000	0.0000000000
hlt	0.0000000000	0.0000730641	0.0000041878
icebp	0.0000000000	0.0000000000	0.0000757657
idiv	0.0000000000	0.0000000000	0.0002392601
imul	0.0006387832	0.0005153713	0.0019508906
in	0.0000123661	0.0000000000	0.0000750190
inc	0.0001379581	0.0000300376	0.0001089787
ja	0.0103688836	0.0000000000	0.0000000000
jae	0.0059419521	0.0000000000	0.0000000000
jb	0.0066684532	0.0000000000	0.0000000000
jbe	0.0078527090	0.0000000000	0.0000000000
je	0.1415141447	0.0000000000	0.0000000000
jg	0.0046100334	0.0000000000	0.0000000000
jge	0.0013053394	0.0000000000	0.0000000000
jl	0.0024571094	0.0000000000	0.0000000000
jle	0.0078970079	0.0000000000	0.0000000000
jmp	0.0221411910	0.0608761139	0.0486305594
jne	0.0822895390	0.0000000000	0.0000000000
jnp	0.0000059065	0.0000000000	0.0000000000
jns	0.0011172046	0.0000824689	0.0000095080
jo	0.0000025507	0.0000000000	0.0000244696
jp	0.0000147663	0.0000000000	0.0000000000
js	0.0044416529	0.0000000000	0.0000040482
lea	0.0207468819	0.0374799817	0.0167145260
leave	0.0000048021	0.0000000000	0.0011938042
lock	0.0000000000	0.0000041588	0.0001438256
lods	0.0000000000	0.0000000000	0.0000039877
loop	0.0000000000	0.0000000000	0.0000159507
loope	0.0001358498	0.0000000000	0.0000000000
loopne	0.0000020400	0.0000000000	0.0001408015
mov	0.2143872020	0.5913503857	0.2914029817
movabs	0.0008978493	0.0001832596	0.0006054355
movapd	0.0000738314	0.0000000000	0.0000000000
movaps	0.0027288090	0.0000000000	0.0000000000
movs	0.0000000000	0.0000602594	0.0000000000
movsd	0.0001029955	0.0005576681	0.0000000000
movss	0.0000767847	0.0000000000	0.0000000000
movsx	0.0019227582	0.0004187492	0.0007830236
movsxd	0.0019069202	0.0036603345	0.0406323382
movzx	0.0262046980	0.0025548983	0.0073121624
mul	0.0000280977	0.0000086801	0.0004005715
mulsd	0.0000915510	0.0000000000	0.0000000000

Continued on Next Page...

Table E.4 – Continued			
HMM Parameters			
mulss	0.0000945042	0.0000000000	0.0000000000
neg	0.0002366660	0.0005907692	0.0003802532
nop	0.0195140607	0.0202251175	0.0229434617
not	0.0004292780	0.0000000007	0.0030730658
or	0.0036517954	0.0013136726	0.0059697802
out	0.0000054794	0.0000000000	0.0000404534
pop	0.0000000000	0.0000000000	0.0649671000
push	0.0006618863	0.0000000000	0.0606559482
rep	0.0005787808	0.0014835884	0.0000783925
repnz	0.0000000000	0.0000000000	0.0015751292
repz	0.0059393660	0.0000000000	0.0002706657
ret	0.0002012314	0.0000000000	0.0331170357
retf	0.0000000000	0.0000000000	0.0000039877
rex	0.0000000000	0.0000000000	0.0000199383
rol	0.0000000000	0.0000000000	0.0010766706
ror	0.0000000000	0.0000000000	0.0007098050
sar	0.0000230304	0.0000793049	0.0041844685
sbb	0.0020508079	0.0000000000	0.0001219327
seta	0.0011724428	0.0000000000	0.0000000000
setae	0.0000295326	0.0000000000	0.0000000000
setb	0.0010484061	0.0000000000	0.0000000000
setbe	0.0000502054	0.0000000000	0.0000000000
sete	0.0037831216	0.0000000000	0.0000000000
setg	0.0001742421	0.0000000000	0.0000000000
setge	0.0000354391	0.0000000000	0.0000000000
setl	0.0000413456	0.0000000000	0.0000000000
setle	0.0000265793	0.0000000000	0.0000000000
setne	0.0027312174	0.0000000000	0.0000126987
sets	0.0000088598	0.0000000000	0.0000000000
shl	0.0004156036	0.0006030007	0.0448336949
shr	0.0022834459	0.0001579169	0.0064816637
sldt	0.0000029533	0.0000000000	0.0000000000
stos	0.0000513155	0.0000217031	0.0000420285
sub	0.0106269804	0.0145519773	0.0454590987
subsd	0.0000059065	0.0000000000	0.0000000000
subss	0.0000324858	0.0000000000	0.0000000000
test	0.1301007285	0.0000000000	0.0000904740
ucomisd	0.0000354391	0.0000000000	0.0000000000
ucomiss	0.0001624291	0.0000000000	0.0000000000
xchg	0.0026466208	0.0013329123	0.0016807297
xor	0.0161896499	0.0898088944	0.0423157229
xorpd	0.0000064034	0.0000147027	0.0000000000

Table E.5: HMM matrices, N = 2 Worm-padding ratio: 4.0

HMM Parameters		
N = 2, M = 131, T = 1317049		
π :		
	0.9985471245	0.0014528755
A :		
	0.4967575341	0.5032424659
	0.4597537055	0.5402462945
B :		
adc	0.0000367951	0.0000375784
add	0.0547993781	0.0579618186
addsd	0.0000661023	0.0000572977
addss	0.0000206925	0.0000275895
and	0.0190965142	0.0186388023
bsf	0.0000024473	0.0000021230
bsr	0.0000055805	0.0000065252
bswap	0.0000111953	0.0000101133
bt	0.0000948431	0.0000949696
call	0.0731853855	0.0677838613
cdqe	0.0028501174	0.0030568097
clc	0.0000023837	0.0000021811
cld	0.0000014840	0.0000015501
cmova	0.0000823310	0.0001064005
cmovae	0.0000736147	0.0000736814
cmovb	0.0001471997	0.0001459370
cmovbe	0.0000909783	0.0000868770
cmove	0.0014361536	0.0014165634
cmovg	0.0000714001	0.0000757046
cmovge	0.0000405580	0.0000414053
cmovl	0.0000240547	0.0000230649
cmovle	0.0000500548	0.0000574291
cmovne	0.0008673460	0.0008029283
cmovns	0.0000718731	0.0000810842
cmovs	0.0000347324	0.0000321982
cmp	0.0515371734	0.0485093581
cpuid	0.0000189153	0.0000190427
cvtsi2sd	0.0001117871	0.0000867545
cvtsi2ss	0.0000409571	0.0000439466
cvttsd2si	0.0000034505	0.0000026595
cvttss2si	0.0000153175	0.0000150649
cwde	0.0000227074	0.0000213900
dec	0.0001377207	0.0001269911
div	0.0003169465	0.0003540926
divsd	0.0000369084	0.0000273043
divss	0.0000031685	0.0000029171
enter	0.0000144419	0.0000158648
fcmovnb	0.0000021302	0.0000024126
fild	0.0000041303	0.0000049443

Continued on Next Page...

Table E.5 – Continued

HMM Parameters		
fisttp	0.0000141685	0.0000161146
fld	0.0000035138	0.0000026015
fmul	0.0000069576	0.0000067201
fstp	0.0000040950	0.0000049765
fsub	0.0000007945	0.0000007271
fucomip	0.0000013759	0.0000016489
fxch	0.0000015861	0.0000014568
hlt	0.0000444187	0.0000335194
icebp	0.0000148737	0.0000140174
idiv	0.0000676179	0.0000718953
imul	0.0011402858	0.0010969722
in	0.0000174705	0.0000174567
inc	0.0001124096	0.0001079798
ja	0.0030456559	0.0028665454
jae	0.0018727266	0.0018124722
jb	0.0019311099	0.0017097344
jbe	0.0025497522	0.0024056995
je	0.0523560313	0.0460758585
jg	0.0017213198	0.0014582503
jge	0.0005566059	0.0005100005
jl	0.0007592538	0.0007927095
jle	0.0027505774	0.0024459807
jmp	0.0414677057	0.0538642017
jne	0.0288874261	0.0282174391
jnp	0.0000016647	0.0000013851
jns	0.0004435686	0.0004272941
jo	0.0000053730	0.0000052619
jp	0.0000050735	0.0000040826
js	0.0014141650	0.0012070882
ldmxcsr	0.0000035979	0.0000039777
lea	0.0279573494	0.0268732118
leave	0.0002199441	0.0002465663
lock	0.0000309610	0.0000269260
lods	0.0000005859	0.0000009177
loop	0.0000026285	0.0000034104
loope	0.0000345714	0.0000352511
loopne	0.0000245244	0.0000299006
mov	0.3930410177	0.3871306459
movabs	0.0004675989	0.0004867047
movapd	0.0000221904	0.0000233152
movaps	0.0010165383	0.0012797668
movs	0.0000502519	0.0000427197
movsd	0.0003220612	0.0002607909
movss	0.0000226994	0.0000228502
movsx	0.0012568459	0.0011110782
movsxd	0.0111392407	0.0106787844
movzx	0.0133626910	0.0112336983
mul	0.0000375296	0.0000427190
mulsd	0.0000254538	0.0000232397

Continued on Next Page...

Table E.5 – Continued		
HMM Parameters		
mulss	0.0000172847	0.0000190794
neg	0.0003653912	0.0003723104
nop	0.0196748718	0.0254666308
not	0.0011966786	0.0010381879
or	0.0038876450	0.0037187921
out	0.0000088417	0.0000093576
pop	0.0122520965	0.0199183607
push	0.0142035825	0.0186425878
rep	0.0008245629	0.0008115026
repnz	0.0005120233	0.0004257781
repz	0.0023332268	0.0019337110
ret	0.0071281524	0.0098289848
retf	0.0000008168	0.0000007067
rex	0.0000042149	0.0000034140
rol	0.0005261586	0.0004695289
ror	0.0001090077	0.0001604875
sar	0.0010894266	0.0012872768
sbb	0.0007413612	0.0006841036
seta	0.0004151954	0.0004009098
setae	0.0000139731	0.0000133872
setb	0.0003812304	0.0003825399
setbe	0.0000194379	0.0000214711
sete	0.0015104261	0.0013356327
setg	0.0000983457	0.0000917697
setge	0.0000185478	0.0000179254
setl	0.0000474728	0.0000467115
setle	0.0000150363	0.0000153217
setne	0.0011659902	0.0011330593
sets	0.0000036357	0.0000024902
shl	0.0089551704	0.0096462016
shr	0.0020773486	0.0024057586
sldt	0.0000007113	0.0000008031
stmxcsr	0.0000032987	0.0000042510
stos	0.0000627205	0.0000560284
sub	0.0212834226	0.0197007569
subsd	0.0000014974	0.0000015378
subss	0.0000070480	0.0000080904
test	0.0465648092	0.0429759322
ucomisd	0.0000119182	0.0000094528
ucomiss	0.0000456744	0.0000410899
xchg	0.0020071457	0.0019729912
xor	0.0547352782	0.0503183741
xorpd	0.0000144939	0.0000129114

Table E.6: HMM matrices, N = 3 Worm-padding ratio: 4.0

HMM Parameters			
N = 3, M = 131, T = 1317049			
π :			
	0.0000000000	0.0000000000	1.0000000000
A :			
	0.8440445378	0.1369401603	0.0190153020
	0.0020971899	0.5243338837	0.4735689264
	0.2700737983	0.4832011537	0.2467250480
B :			
adc	0.0000768630	0.0000154039	0.0000008589
add	0.0219442537	0.0418087140	0.1391768850
addsd	0.0001499633	0.0000000000	0.0000000000
addss	0.0000592448	0.0000000000	0.0000000000
and	0.0056939193	0.0163670891	0.0457075428
bsf	0.0000000000	0.0000000080	0.0000097197
bsr	0.0000148112	0.0000000000	0.0000000000
bswap	0.0000000000	0.0000229204	0.0000105699
bt	0.0002314242	0.0000000000	0.0000000012
call	0.0080288331	0.0040484733	0.2803971111
cdqe	0.0002383820	0.0065649974	0.0022401434
clc	0.0000000000	0.0000062093	0.0000002920
cld	0.0000000000	0.0000041118	0.0000002368
cmova	0.0002314248	0.0000000000	0.0000000000
cmovae	0.0001795857	0.0000000000	0.0000000000
cmovb	0.0003573199	0.0000000000	0.0000000000
cmovbe	0.0001314191	0.0000964306	0.0000026736
cmove	0.0030921788	0.0003939283	0.0000752610
cmovg	0.0001584909	0.0000240330	0.0000004247
cmovge	0.0000999755	0.0000000000	0.0000000000
cmovl	0.0000486198	0.0000094885	0.0000009475
cmovle	0.0001271497	0.0000041919	0.0000011607
cmovne	0.0017783231	0.0002729279	0.0000310218
cmovns	0.0001012792	0.0000981419	0.0000009787
cmovs	0.0000588429	0.0000258632	0.0000003125
cmp	0.1218090644	0.0000000000	0.0000000003
cpuid	0.0000073423	0.0000095105	0.0000537751
cvtsi2sd	0.0002406818	0.0000000000	0.0000000000
cvtsi2ss	0.0001036783	0.0000000000	0.0000000000
cvttsd2si	0.0000074056	0.0000000000	0.0000000000
cvttss2si	0.0000370280	0.0000000000	0.0000000000
cwde	0.0000000000	0.0000599484	0.0000029365
dec	0.0000000004	0.0002381938	0.0002023263
div	0.0000138980	0.0000022417	0.0014093084
divsd	0.0000765486	0.0000012175	0.0000002694
divss	0.0000074056	0.0000000000	0.0000000000
enter	0.0000000000	0.0000407058	0.0000029950
fcmovnb	0.0000000000	0.0000062804	0.0000001839

Continued on Next Page...

Table E.6 – Continued

HMM Parameters			
fild	0.0000005389	0.0000011504	0.0000167704
fisttp	0.0000002058	0.0000051380	0.0000567070
fld	0.0000074056	0.0000000000	0.0000000000
fmul	0.0000000000	0.0000184894	0.0000010865
fstp	0.0000041018	0.0000074992	0.0000008759
fsub	0.0000018514	0.0000000000	0.0000000000
fucomip	0.0000037028	0.0000000000	0.0000000000
fxch	0.0000037028	0.0000000000	0.0000000000
hlt	0.0000000000	0.0000787735	0.0000456841
icebp	0.0000000000	0.0000391792	0.0000020718
idiv	0.0000000000	0.0000063730	0.0002887542
imul	0.0006122794	0.0017069258	0.0011072843
in	0.0000088156	0.0000369855	0.0000029363
inc	0.0000978800	0.0000107109	0.0002825873
ja	0.0071982375	0.0000000000	0.0000000000
jae	0.0044896415	0.0000000000	0.0000000000
jb	0.0044266939	0.0000000001	0.0000000000
jbe	0.0060337079	0.0000000000	0.0000000000
je	0.1196614145	0.0000000316	0.0000000005
jg	0.0038620174	0.0000000000	0.0000000000
jge	0.0012977429	0.0000001003	0.0000000009
jl	0.0018939807	0.0000000000	0.0000000000
jle	0.0063077547	0.0000127123	0.0000000677
jmp	0.0644172341	0.0134160245	0.0715801504
jne	0.0695847451	0.0000000782	0.0000000024
jnp	0.0000037028	0.0000000000	0.0000000000
jns	0.0007647518	0.0003380045	0.0000049546
jo	0.0000001955	0.0000142735	0.0000006654
jp	0.0000111084	0.0000000000	0.0000000000
js	0.0031610721	0.0000264531	0.0000006679
ldmxcsr	0.0000092570	0.0000000000	0.0000000000
lea	0.0163598218	0.0486691714	0.0143699143
leave	0.0004008389	0.0000010553	0.0002951972
lock	0.0000006591	0.0000008555	0.0001208145
lods	0.0000000000	0.0000020111	0.0000001866
loop	0.0000000000	0.0000084117	0.0000001877
loope	0.0000002271	0.0000000000	0.0001488236
loopne	0.0000000000	0.0000641447	0.0000192646
mov	0.1835068386	0.6817903164	0.3080046846
movabs	0.0006096951	0.0002853643	0.0005383271
movapd	0.0000555396	0.0000000027	0.0000000000
movaps	0.0028141258	0.0000000000	0.0000000000
movs	0.0000000000	0.0001099191	0.0000307737
movsd	0.0000677987	0.0005323671	0.0003110500
movss	0.0000555420	0.0000000000	0.0000000000
movsx	0.0019932330	0.0006513406	0.0005616580
movsxd	0.0011773337	0.0251075494	0.0063304003
movzx	0.0252138044	0.0017584942	0.0054858204
mul	0.0000000000	0.0001081539	0.0000075057

Continued on Next Page...

Table E.6 – Continued			
HMM Parameters			
mulsd	0.0000591981	0.0000000356	0.0000000277
mulss	0.0000444336	0.0000000000	0.0000000000
neg	0.0000958148	0.0003869207	0.0008204500
nop	0.0448083873	0.0111497760	0.0015285884
not	0.0001164287	0.0011825577	0.0027570567
or	0.0043755194	0.0019436983	0.0056111448
out	0.0000000007	0.0000242027	0.0000021314
pop	0.0396439971	0.0000000000	0.0000000000
push	0.0402901351	0.0000000000	0.0000000000
rep	0.0003720701	0.0012545017	0.0009346186
repnz	0.0000000000	0.0010618879	0.0003806675
repz	0.0051787613	0.0000000000	0.0000025438
ret	0.0204153817	0.0001521154	0.0004823938
retf	0.0000000000	0.0000020846	0.0000000748
rex	0.0000000000	0.0000001596	0.0000159770
rol	0.0000000000	0.0012890875	0.0001617761
ror	0.0000000000	0.0002182096	0.0002489281
sar	0.0000000000	0.0025773822	0.0011779151
sbb	0.0009545611	0.0008967517	0.0000037268
seta	0.0009942010	0.0000000000	0.0000000000
setae	0.0000333252	0.0000000000	0.0000000000
setb	0.0009312535	0.0000000000	0.0000000000
setbe	0.0000499878	0.0000000000	0.0000000000
sete	0.0034602637	0.0000000002	0.0000000000
setg	0.0002314230	0.0000000021	0.0000000000
setge	0.0000443831	0.0000000573	0.0000000013
setl	0.0001147867	0.0000000000	0.0000000000
setle	0.0000370280	0.0000000000	0.0000000000
setne	0.0028010475	0.0000001220	0.0000000220
sets	0.0000074056	0.0000000000	0.0000000000
shl	0.0000000080	0.0219096285	0.0064945917
shr	0.0006075657	0.0012384284	0.0066612662
sldt	0.0000018514	0.0000000000	0.0000000000
stmxcsr	0.0000092570	0.0000000000	0.0000000000
stos	0.0000570179	0.0000700050	0.0000466967
sub	0.0114616550	0.0201788585	0.0366383704
subsd	0.0000037028	0.0000000000	0.0000000000
subss	0.0000185140	0.0000000000	0.0000000000
test	0.1089427232	0.0000000001	0.0000471196
ucomisd	0.0000259196	0.0000000000	0.0000000000
ucomiss	0.0001055297	0.0000000000	0.0000000000
xchg	0.0041833547	0.0006868617	0.0001250157
xor	0.0182797050	0.0888314225	0.0569142674
xorpd	0.0000188458	0.0000161514	0.0000008157