Spring 2012

Chi-Squared Distance and Metamorphic Virus Detection

Annie Hii Toderici
San Jose State University

CHI-SQUARED DISTANCE AND METAMORPHIC VIRUS DETECTION

A Thesis

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

Annie H. Toderici

May 2012

The Designated Thesis Committee Approves the Thesis Titled

CHI-SQUARED DISTANCE AND METAMORPHIC VIRUS DETECTION

by

Annie H. Toderici

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2012

Dr. Mark Stamp	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science
Dr. David Ross	Google

ABSTRACT

CHI-SQUARED DISTANCE AND METAMORPHIC VIRUS DETECTION

by Annie H. Toderici

Malware are programs that are designed with a malicious intent. Metamorphic malware change their internal structure each generation while still maintaining their original behavior. As metamorphic malware become more sophisticated, it is important to develop efficient and accurate detection techniques. Current commercial antivirus software generally try to scan for malware signatures within files and match them against a known set of signatures; therefore, they are not able to detect metamorphic malware that change their body from generation to generation, with each copy comprised of its own unique signature. Machine learning methods such as hidden Markov models (HMM) have shown promising results in detecting metamorphic malware. However, it is possible to exploit a weakness in HMMs and avoid detection by morphing and merging the malware with contents from normal files. As an alternative approach, we consider combining HMMs with the statistical framework of the chi-squared test to build a new detection method. This paper will present the experimental results of our proposed hybrid detector in metamorphic malware detection.

ACKNOWLEDGMENTS

I want to thank all the people who helped make the completion of this thesis possible. I thank my husband and my first reader, George, for pushing me beyond my comfort level and his encouragement throughout my whole graduate studies. Without his unconditional support, this thesis would not be possible. Another important person is my advisor, Dr. Mark Stamp. I would like to thank Dr. Stamp for his persistence, guidance, support, and advice in the process of developing this thesis. Moreover, I want to thank my graduate committee, Dr. Sami Khuri and Dr. David Ross, for their valuable time and assistance in finalizing my thesis.

I would like to thank Steve Urban, Michael Fang, and Sean O'Malley for proofreading my thesis and providing me with valuable corrections.

I also would like to acknowledge my remarkable family and friends for their constant encouragement and especially for my beloved father who had always believed in me and provided me with an opportunity for a higher education.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Over the past decade, computers have become a tool in our daily lives. People use email to write their correspondence instead of using traditional paper letters and the postal service. Driven by the advances in Internet and cloud computing technology, most businesses and organizations offer online services. Banks and financial organizations provide online access to their customers, allowing them to initiate transactions (for example, their customers can pay their bills through this system), and offer other services including depositing checks without having to go to the local branch of the bank. Government agencies such as the Department of Motor Vehicles now also allow drivers to renew driver licenses or car registrations on their websites [1]. Public transportation agencies such as the Bay Area Rapid Transit also rely on computers to transmit the route locations and schedules to their operators to make sure that they are safe [2].

This convenience of computers and the massive importance they play in our daily lives also drew an increase in malware (malicious software) attacks [3]. Malware writers are enticed by the potential profit they can obtain by controlling a massive number of computers and potentially extracting high value information from them (for example, credit card numbers, bank accounts, user names, and passwords to email accounts). Since our society relies so much on computers, malware poses a very serious threat to virtually all people using computers.

Due to the potential profit malware writers can make, the number of malicious software application has been rising speedily, and some of the attacks

caused extensive financial damage and interruption. According to McAfee, MyDoom is a massive spam-mailing malware that caused the most monetary damage of all time, an estimated $38 billion, and Conficker is a password-stealing botnet that caused an estimated of $9.1 billion in damages [4]. The most common types of attacks against users are designed to commit identity theft or credit card fraud. Sometimes they cause damage on the computer by overwriting data on the hard disk, encrypting important data, or blackmailing the user.

In order to prevent infections by computer viruses, or malware attacks, antiviral defense is generally applied. There are many antivirus defense systems based on algorithms such as signature detection, code emulation, heuristic code analysis, and machine learning. Signature detection is the most commonly used algorithm for detecting viruses [5]. It relies on a large database of known signatures for viruses and malware, and it matches them against all the files on the user's computer. If any one signature is matched to a file, it means that the file is likely to be infected by the corresponding malware such as a virus. Traditionally, this method has been very effective for detecting most naïve viruses. The weakness of the signature detection technique is that it cannot detect new or previously unknown variants of viruses.

Malware writers know this weakness and have been creating variations of viruses by employing a variety of code obfuscation methods [6]. The most common methods for code obfuscation are reordering instructions, renaming registers, making spaghetti code, substituting sets of equivalent instructions, and inserting junk code. Metamorphic viruses normally apply these methods. They are the most difficult type of virus to detect since they morph into a new copy at each infection.

Previous research has shown that machine learning methods such as hidden

Markov models (HMMs) can be used to detect metamorphic viruses [7]. However, the new metamorphic virus generator created by Lin [8] has been shown to be capable of defeating the HMM-based detection methods.

We studied the statistical properties of the χ^2 test (which is also written in literature as "chi-squared test") and its relevance to virus prediction tasks, as suggested by Filiol and Josse's theoretical framework [9]. We show that the χ^2 distance test, which is computed directly on instruction frequencies, can benefit virus detection and identification. We present our proposed hybrid virus detector based on the HMM and the χ^2 distance and its results. The goal of this project is to determine whether simple statistics can be used to predict the presence and absence of viruses.

The thesis is structured as follows: Chapter 2 gives background information on various computer viruses and the code obfuscation techniques. Chapter 3 provides an overview of the antivirus defense mechanisms. Chapter 4 introduces our proposed hybrid virus detector. Chapter 5 briefly discusses the performance evaluation techniques. Chapter 6 presents our experimental setup. Chapter 7 discusses the experimental results. The conclusion is presented in Chapter 8.

CHAPTER 2

Malware

Malicious software or malware is software designed for malicious purposes. Some malware may delete, overwrite, or steal user data. In general, this type of software can cause damage to the user's computer and may steal vital information. Since this is a broad definition, malware can be classified into categories such as viruses, worms, trojan horses, spyware, adware, or botnets. Since there is substantial overlap between these type of malware, we refer to them simply as "viruses" [5]. We can further classify viruses based on the way they try to conceal themselves from being detected by antivirus programs [10]. These categories are "encrypted," "polymorphic," and "metamorphic."

2.1 Encrypted Viruses

"Encrypted viruses" refer to those viruses that encrypt their body using a specified encryption algorithm but using different keys at every infection. Each encrypted virus has a decryption routine that usually remains the same, despite the fact that the keys change between infections. Therefore, it is possible to detect this class of viruses by analyzing the decryptor in order to obtain a reasonable signature. Figure 1 shows an encrypted virus example.

Encrypted viruses tend to use simple algorithms for encryption. Common variants use algorithms such as XORing the body of the virus with the encryption key. Despite its effort to encrypt its body, this type of viruses can be easily detected by signature detection. Listing 2.1 illustrates a simple encryption code written in assembly using the XOR function in the decryption loop with the key 0x55.

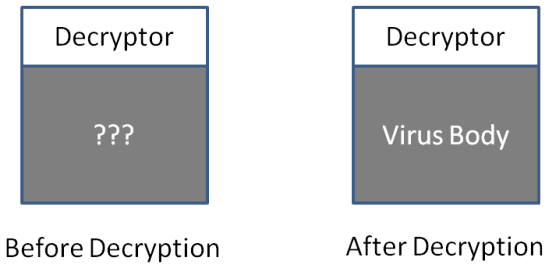

Figure 1: Encrypted virus before and after decryption.

```
decrypt_loop:

lodsd                   ; load a double word
xor eax,0x55555555      ; xor each byte with 0x55 (encrypt or decrypt)
stosd                   ; store back the word
sub ecx, 4              ; decrease the counter by 4 (bytes)
jnz decrypt_loop        ; as long as there are bytes left, continue
```

Listing 2.1: An example of a simple decryptor code.

2.2 Polymorphic Viruses

Polymorphic viruses are similar to encrypted viruses, but they obfuscate the decryption code by mutating it. Even though there are many variants of these decryptors, it is still possible to locate all the signatures of these decryptors. In addition, if a part of the code of a program looks "suspicious," we can test it by running it in a virtual machine and analyzing the effect it has on its own code segment. When the code finally decrypts itself, we can look for the signature for that decryptor or even the decrypted body of the virus. Therefore, polymorphic viruses can still be detected by signature detection, although it is time consuming to find the reliable signature patterns given the possible diversity of decryptor bodies and possibly the high cost of having to run the virus inside a virtual machine for cases in which the detection algorithm cannot determine whether the file is infected or not.

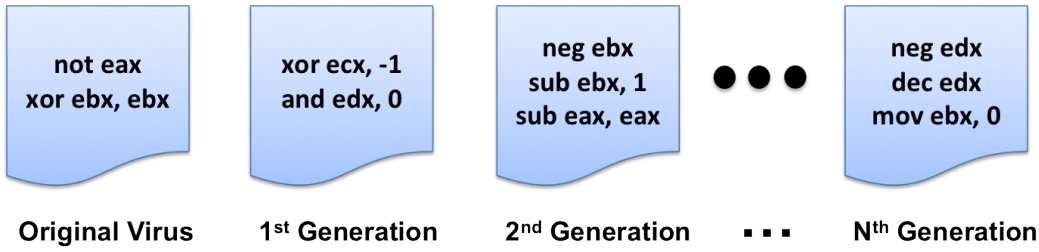

Figure 2: Multiple shapes of a metamorphic virus at each generation.

2.3 Metamorphic Viruses

Related to polymorphic viruses, metamorphic viruses change their internal structure on each generation. Unlike polymorphic viruses, the entire virus changes at each generation while still maintaining its original behavior. Figure 2 illustrates the multiple shapes of a metamorphic virus. Virus writers have tried various techniques when implementing metamorphic engines such as reordering subroutines, inserting garbage instructions, exchanging register usage, or substituting equivalent instructions. As metamorphic viruses become more sophisticated, it is important to develop efficient and accurate detection techniques. Current generation antivirus software applications generally try to scan for virus signatures within files to locate any known virus. This technique is used very often simply because it is not only effective in detecting known viruses but it is also very efficient. However, since metamorphic viruses change their body from generation to generation, each generation has its own unique signature. Therefore, current antivirus software program cannot detect variants of these viruses with ease.

Even though it is difficult to detect this type of virus, it is also difficult for malware writers to implement it correctly. The problem is that virus writers need to be able to make their malware mutate without arbitrarily increasing in size. Moreover, in the case of metamorphic viruses, the virus writers may need also to be

able to detect whether any particular file is infected so that the virus will not try to reinfect it again. All these issues are very difficult to address programmatically, without causing bugs in the code or potentially providing the antivirus programs an easy way of detecting the virus.

Many malware writers claim that they have implemented virus generators that are metamorphic. However, studies [7, 11] that analyzed such generators, have shown that only very few of them exhibit true metamorphic behavior. The studies included the following examples of generators that claimed to be metamorphic but were not: Phalcon/Skism Mass-Produced Code generator (PS-MPC), Second Generation virus generator (G2), Mass Code Generator (MPCGEN), and Virus Creation Lab for Win32 (VCL32) [12]. Out of all the generators that claimed to be metamorphic, only the Next Generation Virus Construction Kit (NGVCK) is indeed metamorphic [7, 11].

2.3.1 Virus Obfuscation Techniques

Code obfuscation techniques are generally applied when creating metamorphic viruses. These techniques when used in combination are able to create a vast number of distinct copies with equivalent code and behavior.

2.3.1.1 Register Renaming

Register renaming is one of the simplest techniques use in metamorphic generators. This is done by simply replacing the registers that the instructions use with different ones. See Figure 3 for an example. This technique does not fool the human eye when applied, since it is easy to spot by looking at the disassembled code, but it changes the binary bit pattern from the executable files and thus

generates a slightly different signature. Since the opcode sequences are not changed, it is still possible to detect this register renaming technique by using wildcard strings [13]. An example wildcard would need to be able to match sequences of `MOV Reg, 4`, `MOV Reg, Reg`, `SUB Reg, 1`. `Reg` represents the register used, and it can be `ax, bx, cx, eax, ebx`, etc.

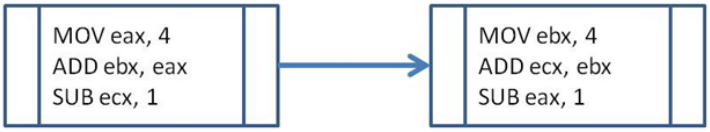

Figure 3: Register renaming example.

2.3.1.2 Equivalent Instruction Replacement

The instruction set for many modern Complex Instruction Set CPUs (CISC) have instructions that can either be substituted by an equivalent instruction with the same effect or with a sequence of other instructions that cause the same behavior. An example of such instruction-level equivalence is `MOV eax, 0` which is equivalent to `SUB eax, eax` or `XOR eax, eax`. Figure 4 illustrates an example where a single instruction is replaced with a sequence of instructions with an equivalent effect.

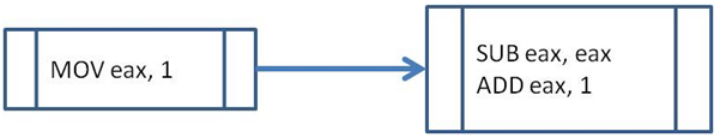

Figure 4: Equivalent instructions substitution.

2.3.1.3 Instruction Reordering

Instruction reordering is done by transposing instructions that do not depend on the output of previous instructions to generate a different but equivalent sequence of instructions. Due to the fact that now the instructions are reordered, the signature of the code is different, but since none of the instructions with dependencies were changed, the actual outcome of the code is identical to the original. Figure 5 shows that the last instruction can be transposed since it does not affect the outcome of the other instructions. However, the first instruction has to come before the second instruction; therefore, the ordering cannot be changed there. In addition, the figure also demonstrates that register renaming can be employed simultaneously with instruction reordering. This results in a powerful method for avoiding detection, since making wildcard signatures that are able to capture this behavior becomes much more difficult and computationally expensive.

Figure 5: Instruction reordering and equivalent instructions substitution.

2.3.1.4 Inserting Junk Code

Junk code is code that once executed will not affect the behavior of the program. Typically, junk code is inserted randomly throughout the body of the virus during the morphing process. The intention is that such junk code will confuse signature-based antiviruses. The most common example of junk code is randomly inserting the `NOP` instruction. This instruction simply does nothing in terms of affecting the CPU state, and modern detection algorithms usually have a rule to

remove or treat specially the `NOP` instructions encountered in executable files. However, there are other examples of junk code such as `MOV eax, eax`, `ADD eax, 0`, and `SUB eax, 0`.

It is possible to create relatively long sequences of instructions that, once chained together, have absolutely no effect except for making the detection of the "live" code more difficult. A jump instruction can be added right in front of this long sequences of junk instructions in order to completely skip it during runtime. Since the location of such code can be random, it can make signature-based approaches less likely to succeed, and multiple signatures must be used in order to properly detect such viruses.

CHAPTER 3

Malware Detection

Antiviruses and malware detection programs have been developed to combat these threats. Most antivirus programs are built with the assumption that the virus does not change its structure or code. With the constant evolution of virus creation, there are many areas of improvement, and both academia and antivirus software developers continually conduct research in order to lessen the threat from virus makers. This section discusses several detection approaches such as signature detection, heuristic detection, and the use of machine learning techniques.

3.1 Signature Detection

Commercial antivirus software typically uses signature detection to identify malicious files. A signature is created by analyzing the binary code of a virus body, and selecting a sequence of byte code that is unique to that virus [5]. This string of bytes extracted from the virus must be uniquely different from normal benign files. The signature has to be long enough that it would not appear in normal files, which would also almost guarantee that this signature will not be shared with other viruses either. An example of a signature extracted from the Chernobyl/CIH virus can be detected with the following sequence of bytes [14]:

```
E800  0000  005B  8D4B  4251  5050
0F01  4C24  FE5B  83C3  1CFA  8B2B
```

String matching algorithms are normally applied as the scanning mechanism in antiviruses. Algorithms such as Aho-Corasick, Veldman, and Wu-Manber are a

few string matching algorithms used [10]. The Aho-Corasick algorithm can only scan for exactly matching signatures, thus a slight variation will escape being detected [15]. Veldman and Wu-Manber proposed the use of wildcard search strings to help detecting these slight variants of the viruses [6].

After locating a malware signature, antivirus software vendors incorporate this signature into their database of virus signatures which they maintain and push out to their users. This has created a market for antivirus makers, since they now charge subscription fees for updating the virus signature database for the user.

Scanning files against a large database of virus signatures is simple and efficient. Users of antivirus software only need to click on the "start scanning" button or set a recurring scanning schedule, and the antivirus will do the work of scanning the hard-drive. However, this ease of use, and the relatively high speed of scanning comes at a cost: the virus database needs to be kept up to date. Moreover, the antivirus software is not able to detect all existing malware, because if none of the current signatures match a particular new malware, it will not be known until the antivirus makers acquire the signature for it. Thus, unknown viruses are very unlikely to be detected during a signature-based scan.

Antivirus software makers maintain the database of virus signatures by constantly searching the Internet for new viruses and generating suitable signatures for all viruses they find. This leads to a completely reactive scenario where the antivirus manufacturers are unable to combat a new virus until after it has caused damage. Fortunately, it is possible to find new viruses by simply reading the right internet forums where users trade new viruses. In addition, keeping contact with customers who found suspiciously behaving files is an invaluable source of information on new malware.

3.2 Heuristic Detection

Heuristic analysis can be used to detect unknown viruses and variants of known viruses. Heuristic analysis is either done using static or dynamic analysis. When encountering a suspicious program, static heuristic analysis will disassemble it for further study. The code is analyzed to see whether there are matching viral code patterns that are close to known viruses. The code will be evaluated and given a percentage of similarity to viruses, and if it surpasses a threshold, this program will be marked as infected [10].

In the case of dynamic heuristic analysis, the suspected program is executed under a virtual machine which is monitored for any viral behaviors. Once the program exhibits any viral behaviors, this program will be marked as infected. Examples of viral behavior include attempting to open other executable files with the intent of modifying their content, attempts to change the Master Boot Record (in case of boot sector viruses), or attempts at concealing themselves from the operating system. In addition, most modern antiviruses now provide programs which run in the background, and constantly monitor for suspicious behavior [10].

3.3 Machine Learning Techniques

Machine learning techniques have gained popularity in recent years. Tom Mitchell defines machine learning as the study of computer algorithms that improve through experiments [16]. Examples of such techniques are Naïve Bayes [17], decision trees [18], hidden Markov models [7, 11], and other statistical learning methods [9].

Back in 1986, Cohen [19] proposed an undecidability theorem of virus detection which indicates that it is impossible to find a single algorithm that can

detect all possible viruses. Chess and White [20] also further supported this idea by showing that there exist viruses which cannot be detected by any algorithm. Therefore, we are focusing on the detection of metamorphic viruses generated using the NGVCK virus and Lin's metamorphic generator [8]. Machine learning techniques such as the HMM and χ^2 statistical framework will be the main focus of this project, which we will introduce in the following two sections.

3.4 Hidden Markov Model

A hidden Markov model (HMM) is a statistical modeling method that has been used in speech recognition, bioinformatics, mouse gesture recognition, credit card fraud detection, and computer virus detection research. The HMM is one of the most popular models to use for sequential data. It is widely used because it is simple and it is computationally fast.

The popularity of HMMs within the virus detection community stems from the fact that programs can be represented as sequences of instructions. The CPU executes them one at a time, which effectively means that programs can be treated as time series: the type of data that HMMs excel on.

A hidden Markov model is a stochastic model which assumes that the underlying data can be expressed as a Markov process with hidden states. In essence, it can be viewed as a variant of a probabilistic state machine, where the transition probabilities from one state to another depend only on the current state.

A Markov model assumes that sequential data can be modeled based solely on the current state, with absolutely no memory. What happens next in the sequence depends only on the current state with nothing in the past influencing the state transition. In a typical Markov chain, the states are fully observed. In the case of

Table 1: The list of symbols and their meanings.

Symbol	Description
T	The length of the observation sequence
N	The number of states in the model
M	The number of distinct observation symbols
$\mathcal{O}$	The observation sequence $\mathcal{O} = \{\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}\}$
Q	The set of states of the Markov process
V	The set of observation symbols
A	The state transition probability matrix
B	The observation probability matrix
π	The initial state distribution
λ	The hidden Markov model, as defined by its parameters A, B, π, is depicted as $\lambda = (A, B, \pi)$

hidden Markov model, as its name implies, the states are never observed, as they are "hidden." We can only estimate these states while observing sequences of data. HMM will choose the sequence of states that jointly maximizes the probability of the entire observation sequence [21].

3.4.1 Notation

In order to mathematically describe the HMM, we will use the notation summarized in Table 1. The main components for the HMM are the state transition probability matrix (A), the observation probability matrix (B) which gives the likelihood of observation given the state, the initial state distribution (π), observation sequence (i.e. $\mathcal{O} = \{\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3\}$), and the hidden states (i.e. $X = \{x_0, x_1, x_2, x_3\}$). The first three elements define the HMM model (λ) such that $\lambda = (A, B, \pi)$. Matrices A and B are row stochastic, so each row must sum up to 1. π_{x_0} is the probability of starting in the first state (x_0). a_{x_0,x_1} is the probability of transitioning from state x_0 to state x_1. Finally, $b_{x_0}(\mathcal{O}_0)$ is the probability of observing $\mathcal{O}_0$ at state X_0.

Figure 6 shows a graphical representation of a generic HMM. The Markov process gives the transition between states. However, in the case of hidden Markov model, the transition between states is not observed, as the states are "hidden." The state transitions are placed above the dashed line, in order to indicate the fact that we do not "know" what the transitions should be. The transitions are "learned" by analyzing the observations (below the dashed line). Only the observation sequence is available for study while the state transition sequence of the Markov process is hidden behind the dashed line. The Markov process is estimated by knowing the initial state x_0 and the state-to-state transition probabilities obtained from the matrix A. The observation sequence is related to the states of the Markov process by the matrix B, which describes the probability of observing a particular symbol in a given state.

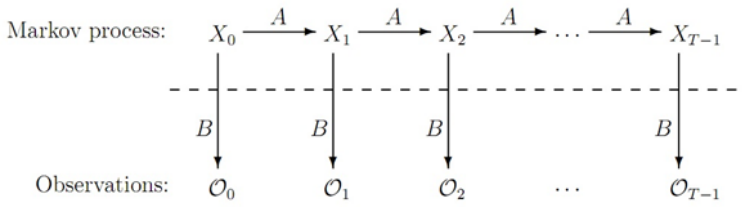

Figure 6: Markov Process. Reprinted with permission from [21].

HMM is used to solve three problems:

1. For the first problem, we want to find the probability of this observed sequence $\mathcal{O}$ happening under a given model λ. An example use for this is computing the probability of a file being a virus. The forward algorithm, or α-pass, is used to determine the likelihood of the observed sequence.

2. The second problem intends to find the hidden state sequence from the sequence of observations and a given model λ. The Viterbi algorithm and

backward algorithm, or β-pass, are used to find the optimal state sequence. An example application of this, assuming a HMM with two hidden states would be to determine the most "virus-like" areas of a file, where "virus-like" would be associated with the model being in a specific state. In practice, however, it is enough to simply determine the probability of a file being a virus.

3. The last problem finds the model λ that maximizes the probability of a given observation sequence $\mathcal{O}$. The Baum-Welch algorithm can provide an estimation for the model parameters given the observations.

Here, we are interested in the application of these algorithms and to ensure consistency across our research, we utilized the HMM library from [21].

3.4.2 Example HMM

An example from [21] will be used to illustrate the concepts of HMM. Suppose we are meteorologists working to determine the average annual temperature of a particular location over a series of years. Let's assume that the years of interest occurred before the invention of thermometers. Given this, there is no way of knowing exactly what the temperature was during the time of interest. However, let's say that researchers found evidence of the temperature being related to the growth of the trees. In this case, we can estimate the temperature based on the observation of tree ring sizes instead. The temperature will be either hot (H) or cold (C), and the tree ring sizes are either small (S), medium (M), or large (L).

Suppose that researchers have evidence which gives them the temperature transition probabilities and relationship between the temperature and the tree ring sizes. For the temperature to transition from a hot year to a cold year, the

probability is 0.3, and thus there is a higher chance of 0.7 for the temperature to remain hot. The probability for the next year to stay hot is therefore 0.7. For a cold year to transition to a hot year, the probability is 0.4, and again, there is a higher chance of 0.6 to remain cold for another year. Equation 1 is the matrix representation of the temperature transition probability which we have described thus far. This is the state-to-state transition matrix A:

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{cc} H & C \end{array} \\ \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \end{array} \tag{1}$$

In a hot year, the probabilities of the small, medium, and large tree ring sizes are 0.1, 0.4, and 0.5 respectively. In a cold year, the probabilities are 0.7, 0.2, and 0.1 for the small, medium, and large tree ring sizes respectively. Table 2 shows the matrix for the observation-to-state transition probabilities B.

The initial state probability distribution for a hot year is 0.6 and 0.4 for a cold year. Now the goal is to observe a four year period of tree ring sizes to determine the most likely state sequence of the Markov process. In this four year period, the tree ring sizes are S, M, S, L. To simplify the observation notation, S is represented as 0, M as 1, and L as 2. Using this notation, the observation sequence $\mathcal{O}$ is $\{0, 1, 0, 2\}$. Table 2 summarizes the parameters for this particular HMM:

Let's set $X = \{x_0, x_1, x_2, x_3\}$ to be a generic state sequence of length four. The probability of the state sequence X can be calculated using the formula given in Equation 3, where $\mathcal{O}_k$ refers to the k^{th} element of $\mathcal{O}$.

Normally,

$$P(X) = \pi_{x_0} P(x_1|x_0) P(x_2|x_0, x_1) P(x_3|x_0, x_1, x_2) \tag{2}$$

Table 2: Parameter values and their explanations.

Parameter and Value	Explanation
$\mathcal{O} = \{0, 1, 0, 2\}$	The observation sequence for this model
$V = \{0, 1, 2\}$	The observation symbols of tree ring sizes, corresponding to S, M, and L
$Q = \{H, C\}$	The states in the model of states are H and C
$T = 4$	The length of the observation sequence of $\mathcal{O} = \{0, 1, 0, 2\}$
$N = 2$	The number of states in the model with states of either H or C
$M = 3$	The number of observation symbols with tree ring sizes of S, M, and L
$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$	The state transition probabilities
$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$	The observation probabilities
$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$	The initial state transition probabilities

However, under the HMM assumptions, this equation becomes:

$$P(X) = \pi_{x_0} b_{x_0}(\mathcal{O}_0) a_{x_0,x_1} b_{x_1}(\mathcal{O}_1) a_{x_1,x_2} b_{x_2}(\mathcal{O}_2) a_{x_2,x_3} b_{x_3}(\mathcal{O}_3) \tag{3}$$

To find out how likely the state sequence of HHCC is to occur, we can use the formulation from Equation 3. Substituting the symbols for their values, we obtain:

$$P(HHCC) = (0.6)(0.1)(0.7)(0.4)(0.3)(0.7)(0.6)(0.1) = 0.000212 \tag{4}$$

Given the observation sequence, in order to find the optimal hidden states sequence, we can use the Viterbi algorithm and the backward algorithm. The most probable symbol is chosen at each position based on Table 3. Thus, the optimal state sequence for $\mathcal{O} = \{0, 1, 0, 2\}$ is CHCH.

Table 3: Probabilities for Hot/Cold given the four observations.

	Element			
	0	1	2	3
P(H)	0.188170	0.519432	0.228878	0.803979
P(C)	0.811830	0.480568	0.771122	0.196021

3.5 Chi-Squared Distance

Statistical analysis is commonly used by professionals and researchers to evaluate statements or claims. The two major areas of inferential statistics are estimation of parameters and hypothesis testing. We will illustrate what these terms mean with the following example.

A study is commissioned for compiling home prices at location A. After gathering data on the prices at that location, a histogram of the prices is plotted. The histogram graph is simply summarizing the possible price ranges, by counting the number of houses within each possible price bucket. This graph exhibits a bell-shaped curve demonstrating the normal distribution. The bell shape is centered at the average (or mean) house price. The majority of the prices should be close to the average, with a few "outlier" values for prices which either cost a very large sum of money, or for those houses which are in foreclosure and cost very little.

Now, let us suppose the same study is conducted at another location, B. After gathering the housing prices, a graph is also plotted using data received. If location B is from a similar area to A, this graph should yield a similar bell-shaped graph. Statistical testing will be applied to see whether these two graphs are significantly similar or different from each other.

Since each graph is assumed to follow a normal distribution, it is useful to summarize the data before comparing it. The parameters of this distribution are

estimated from the data at location A, and from the data at location B. Using a statistical test, we can determine whether the estimated parameters are sufficiently close or far apart in order to draw a conclusion.

3.5.1 Notation

Suppose that X is a statistical variable coming from a distribution under observation. The goal is to estimate the main characteristics of X's probability distribution P. Let $X_1, X_2, \ldots, X_n$ be a random sample of elements from this distribution. These samples reveal some information about the unknown parameter θ that are used to estimate the probability law P. The function that depends on $X_1, X_2, \ldots, X_n$ which is used to estimate θ is denoted by $f(X_1, X_2, \ldots, X_n)$, and it is called an estimator function. An estimator function is used to compute the probability distribution on every sample. The notation $\theta_n^* = f(X_1, X_2, \ldots, X_n)$ will be used to measure an estimate for an unknown feature θ of P.

The parameter space from which θ is drawn is completely arbitrary, and it depends on the problem and the choice of f. For example, it may be equal to the set of natural numbers, $\mathbb{N}$, or the k-dimensional real numbers, $\mathbb{R}^k$. To further illustrate this idea with a typical usage scenario, if θ represents the parameters of a normal distribution, it will have two dimensions, in which the first dimension would correspond to the mean, μ, and the second to the variance, σ. For the purpose of malware analysis, the parameter space is restricted to all k-dimensional vectors with dimensions coming from the set of natural numbers. The form of P is assumed to be known.

The statistical testing is used to decide which hypotheses will be likely belong to the observation samples $(X_1, X_2, \ldots, X_n)$. In statistical testing, one or more

initial hypotheses are proposed which will either be kept or rejected after testing on the likelihood of these hypotheses with respect to the probabilistic law of X.

There can be many hypotheses but we will only consider two. The initial hypothesis is denoted $\mathcal{H}_0$ and is referred to as the *null hypothesis*. The *alternative hypothesis* is denoted as $\mathcal{H}_1$. The tests which we are interested in will either accept or reject these two hypotheses.

In order to build a test for a given sample, an estimator E is used as the decision threshold to decide whether to keep or reject the null hypothesis. The observed value e will be computed on this sample and compared with the estimator value, E. This threshold will have two sets of disjoint values with acceptance region A and rejection region B.

There are two types of errors associated with any detection problem:

- The type I error, denoted as α, represents the false positive probability rate. This error indicates that the probability of falsely rejecting the null hypothesis.
- The type II error, β, represents the false negative probability rate. This error indicates the non-detection rate where the null hypothesis is kept, while the alternative hypothesis is actually the correct one.

In an antiviral context, the type I error is more important than the type II error, and it is usually difficult to determine the type II error due to the unknown number of non-detections.

3.5.2 Statistical Testing on Malware Detection

In order to formalize the virus detection framework, we will discuss some notation proposed by Chess and White [20]. An antivirus uses a detection algorithm D, which it applies on a program p. The goal of the algorithm is to determine whether p is infected by a particular virus V. $D(p)$ should return true if and only if the program p is infected by the virus V. Filiol and Josse further expand on the ideas from Chess and White's idea by providing a *statistical framework* for describing the detection process [9].

In the case of malware analysis, we will be considering the analysis of instruction frequencies. We are interested in modelling the behavior of a compiler which produces "clean" programs. The *spectrum* of instructions of such a compiler is obtained by analyzing the instruction frequencies of all possible programs produced by such a compiler. However, this is impossible to compute in an exact form in practice. An approximation is to compute the instruction frequencies from a large set of programs that are known to not be malware, and that are compiled with the compiler in question.

Most compilers normally only use a small subset of all possible instructions, whereas malware will consider the whole set or larger set of these instructions. Knowing this fact, we can develop an estimator function which should be able to determine whether a particular program has been compiled with the known compiler, or whether it is malware.

The formal definition of the spectrum is:

$$spec(C) = (I_i, n_i)_{1 \leq i \leq c} \tag{5}$$

In Equation 5, C represents a particular compiler, I_i represents the i^{th}

instruction, and n_i is i^{th} instruction's frequency. c denotes the total number of unique instructions that may ever be output by the compiler.

In our experiments, the spectrum of the compiler represents the expected value to be observed in the normal set of files. As such, in any normal file, for instruction i we expect to observe a frequency of n_i. In the case of an 80x86 compiler, at the end of 2011, the total number of possible instructions was 501 [22].

When trying to determine whether a particular file comes from a compiler with a known spectrum, or whether the file is in fact malware, the first step we take is to compute the instruction frequencies observed in this file. We denote the observed frequency for instruction i by $\hat{n}_i$.

There are a few important formulae to consider, such as the null hypothesis, estimator function, and the decision threshold. The null hypothesis $\mathcal{H}_0$ (Equation 6) states that the frequencies are the same for the observed and expected file.

$$\mathcal{H}_0 : \hat{n}_i = n_i, \quad 1 \leq i \leq c \tag{6}$$

$$\mathcal{H}_1 : \hat{n}_i \neq n_i, \quad 1 \leq i \leq c \tag{7}$$

If the frequencies are the same, then the suspected file is likely to be uninfected, and the null hypothesis will be accepted. However, if the frequencies are significantly different, then the alternative hypothesis (Equation 7) will be accepted which means that the suspected file is in fact infected with a virus.

Since there is more than one instruction from a compiler or from a file, these simple hypotheses will have to be extended to consider more instructions. Filiol and Josse [9] purposed an estimator function to set up these hypotheses.

One choice for the estimator function, is D_2, is also known as the Pearson's χ^2

statistic test, and is expressed as follows:

$$\mathrm{D}^2 = \sum_{i=1}^{c} \frac{(\hat{n}_i - n_i)^2}{n_i} \tag{8}$$

Pearson's χ^2 statistic test is commonly used to check whether the difference between the expected and observed data is significant. The decision threshold is obtained by comparing the estimator value given by D^2 with the $\chi^2(\alpha, c-1)$ distribution with $c-1$ degrees of freedom and a type I error rate of α. Typically, the type I error rate is set at 0.05, which means that the test tolerates no more than 5% of the files being falsely classified as infected when they are normal files.

The updated null hypothesis and the alternative hypothesis are:

$$\begin{cases} \mathcal{H}_0 & if \quad D^2 \leq \chi^2(\alpha, c-1) \\ \mathcal{H}_1 & if \quad D^2 > \chi^2(\alpha, c-1) \end{cases} \tag{9}$$

The estimator D^2 computes the Pearson's χ^2 distance between the spectrum of the compiler and the current testing file. This value can be compared with the χ^2 value for the given number of instructions and the desired false positive rate (in our case, it is set to 5%).

The main goal of the χ^2 statistic test is to determine whether the distribution agrees with (or "fits") some expected distribution.

The following steps are needed to perform the Pearson χ^2 statistic test given a compiler, and a file to be tested:

1. Set up the null hypothesis $\mathcal{H}_0$ and the alternative hypothesis $\mathcal{H}_1$
2. Choose a significance level (for example $\alpha = 0.05$)
3. Compute the estimator value D^2 given the frequencies in the file to be tested and the compiler's spectrum

Table 4: Summary of notations.

Symbol	Meaning
F	An input file, which comprises of instructions $X_1, X_2, \ldots, X_n$
D	The detection procedure, which decides whether an input file F is infected or not
N	The number of assembly instructions in the code
E	The estimator which measures the frequency of instructions
$\mathcal{H}_0$	The null hypothesis, which assumes the file is an uninfected file
$\mathcal{H}_1$	The alternative hypothesis, assumes the file is infected
Θ	The theoretical mean value of E
α	Type I error rate
β	Type II error rate

4. Evaluate the acceptance or rejection of the null hypothesis $\mathcal{H}_0$ by using Equation 9.

3.5.3 Example

Let's consider a simplified problem involving only three instructions with three frequencies. Instruction I_{i_1} has frequency n_{i_1}, instruction I_{i_2} has frequency n_{i_2}, and instruction I_{i_3} has frequency n_{i_3}.

For this example, the spectrum from the compiler C is listed in Table 5.

Table 5: Example compiler spectrum.

Instruction	Opcode	Frequency n_i
i_1	MOV	7
i_2	PUSH	10
i_3	POP	3

The observation samples for this compiler are (`MOV, PUSH, POP`). The corresponding frequencies are $(7, 10, 3) \in \mathbb{N}^3$. Since we represent the distribution of interest as a histogram over three instructions, the parameter space of the set θ is therefore $\mathbb{N}^3$.

Table 6: Example frequencies of instructions in a program.

Instruction	Opcode	Frequency n_i
i_1	MOV	6
i_3	POP	11

Suppose that we have a file which might be infected. Given the instructions and the frequencies in Table 6, we would like to perform the χ^2 test to see the likelihood of it being uninfected.

The null hypothesis $\mathcal{H}_0$ is that the file is uninfected if the estimator function D^2 yields a score less than or equal to the χ^2 value:

$$\mathrm{D}^2 = \sum_{i=1}^{c} \frac{\left(\hat{n}_i - n_i\right)^2}{n_i} \leq \chi^2\left(\alpha, c-1\right) \tag{10}$$

When computing estimator D^2, we use all of the frequency counts for each instruction. However, since χ^2 is a probability distribution, the frequencies will be normalized before performing this test. Normalization is done by dividing the count of an instruction by the total number of instructions. For example, if there are three `MOV` instructions out of a total of ten instructions, then the normalized value is 0.3. The normalized values for the compiler's spectrum, $spec(C)$ are (`MOV`, 0.35), (`PUSH`, 0.5), and (`POP`, 0.15). The normalized values for the file are (`MOV`, 0.353) and (`POP`, 0.647).

$$D^2 = \frac{(0.353 - 0.35)^2}{0.35} + \frac{(0.0 - 0.5)^2}{0.5} + \frac{(0.647 - 0.15)^2}{0.15} \cong 2.1467$$

We will now test the claim that the observed frequency counts agree with the claimed distribution. The value we compare $D^2 = 2.1467$ against is $\chi^2(0.05, 2) = 5.991$. Since $D^2 \leq \chi^2(0.05, 2)$, we accept the null hypothesis claiming that this file is uninfected and reject the alternative hypothesis.

CHAPTER 4

Hybrid Virus Detection

We propose a new detection method by combining the HMM detector and the CSD estimator which we refer to as the hybrid method.

Our experiments show that the HMM detector performs extremely well in detecting viruses morphed with dead code, while the CSD estimator performs better in detecting viruses morphed with subroutine code from normal files. We are interested to find a model that can combine the advantages of both. We tested two possible approaches to combine the scores from the two algorithms: (1) an additive model (Equation 11), and (2) a multiplicative model (Equation 12).

The additive model is motivated by the fact that in a probabilistic scenario, we want to capture detections from either detector. The multiplicative model is motivated by the case in which we want to emphasize high scores when both detectors yield high scores. Probabilistically, the additive model corresponds to an "OR," whereas the multiplicative model corresponds to an "AND" (assuming the two algorithms provide independent scores):

$$P_{add}(X|virus) = (P_{HMM}(X|virus) + P_{\chi^2}(X|virus)) \div 2 \tag{11}$$

$$P_{mul}(X|virus) = P_{HMM}(X|virus) \cdot P_{\chi^2}(X|virus) \tag{12}$$

We ran experiments and evaluated the performance for both the additive, and the multiplicative models. The better of these two is the multiplicative model, and in the rest of this section we will discuss the details that make it work equally as well or better than the best of HMM and CSD-based detection algorithms.

The HMM allows us to compute $P_{HMM}(X|virus)$ since it is a probabilistic model (see Equation 3 for an example). We write the probability of X being a virus as a conditional probability in order to emphasize that the HMM was trained on virus files. Equivalently, we can compute the probabilities when the training is done on normal files.

The CSD is simply a score, and we can't use it directly to compute a probability. However, the CSD is directly related to the χ^2 distribution, and its cumulative distribution function (CDF). As such, we can write:

$$\begin{aligned} P_{\chi^2}(X|virus) &= P(Y < D^2) \\ P(Y < D^2) &= CDF_{\chi^2}(D^2) \\ &= \frac{1}{\Gamma(k/2)}\gamma(k/2, D^2/2) \end{aligned} \tag{13}$$

In Equation 13, Γ is the Gamma function, k represents the degrees of freedom, which in our case is equivalent to the number of unique instructions encountered in the training phase. $\gamma(k, z)$ is the lower incomplete Gamma function:

$$\Gamma(z) = \int_0^\infty t^{z-1} \cdot e^{-t} \mathrm{d}t \tag{14}$$

$$\gamma(s, x) = \int_0^x t^{s-1} \cdot e^{-t} \mathrm{d}t \tag{15}$$

Equation 12 makes the assumption that the scores from the HMM and CSD are statistically independent. However, since both are trying to detect the same thing, it is very likely that the independence assumption is not true. Moreover, a probability given by the HMM may be more reliable in the case in which we deal with files that have a lot of dead code, whereas the probability given by the CSD may be more reliable when dealing with files that have been morphed with

subroutine code. Hence, we need to find appropriate weights to combine these two scores.

We begin by looking at the log-likelihood for the multiplicative model:

$$\log P_{mul}(X|virus) = \log P_{HMM}(X|virus) + \log P_{\chi^2}(X|virus) \tag{16}$$

In this form, it is possible to assign weights to the HMM and CSD independently, yielding $\log P_{hyb}$ (or the log-likelihood for our method):

$$\log P_{hyb}(X|virus) = w_1 \cdot \log P_{HMM}(X|virus) + w_2 \cdot \log P_{\chi^2}(X|virus) \tag{17}$$

The final probability $P_{hyb}(X|virus)$ is:

$$P_{hyb}(X|virus) = P^{w_1}_{HMM}(X|virus) \cdot P^{w_2}_{\chi^2}(X|virus) \tag{18}$$

The values of w_1 and w_2 were obtained by running a grid search over the set of values between 0 and 1, using a logarithmic scale. The best values were found to be: $w_1 = 10^{-8}$ and $w_2 = 10^{-9}$.

CHAPTER 5

Performance Evaluation

5.1 Cross-Validation

When the amount of data is limited, researchers have to make use of it as efficiently as possible. Nonetheless, in order to compute meaningful statistics, a large amount of data is necessary. This leads to a dilemma: how to obtain meaningful results given a relatively small dataset used for both training and testing.

Researchers have used cross-validation, or rotation estimation [23] in order to alleviate this problem. The basic idea is that since training requires as much data as possible, it would be beneficial to use most of the data possible for training, while leaving a small fraction for testing. Of course, just doing this will yield inaccurate estimates of the performance of the algorithm, since too little data is used for testing.

However, if the training data and the testing data are selected such that they do not intersect, and the experiment is repeated many times on different subsets of data, the accuracy of the performance estimation can be greatly improved.

Typically, researchers use five-fold cross-validation, where 80% of the data is used for training, 20% is used for testing, and the experiment is repeated five times. Each such selection (80:20) is called a "fold." For each fold, the evaluation performance is recorded. All folds are used for estimating the method's performance by computing the mean and standard deviation of the algorithm's performance. If the algorithm has multiple operating points, then a graph can be plotted showing the various operating points as a function of the algorithm's parameter.

For each point on the graph, it is possible to also display the error bars (standard deviation). The generic cross-validation approach has a drawback: if the folds don't contain the same distribution of classes as the original data, a biased error will be computed, which is not desirable. For example, if the real data has a 5:1 ratio of positive to negative classes, and the folds somehow manage to capture a 10:1 ratio due to a sampling problem, then the error estimates will all be incorrect. Therefore, in practice, an alternative method is used, called "stratified cross-validation."

This method ensures that the same proportion of classes is kept for each fold. The key insight for applying stratified cross-validation is to make the splits based on each class, as opposed to basing the split on the entire data. This allows the algorithm to ensure an unbiased split of data.

Even stratified cross-validation is not perfect. The results can be inaccurate if the training and testing sets contain very similar variants of the malware. As a result, there are two distinct tasks in which cross-validation can be used: (1) determining whether a variant of a malware can be detected, in which case it is desirable to have (hopefully dissimilar) variants in both training and testing; (2) determining whether the algorithm can generalize and detect previously unseen malware, while keeping a relatively low false positive rate.

5.2 Evaluation Metrics

The ideal goal of the antivirus defense is to build a detection mechanism that can identify all malware without misclassification, to yield no false positives and no false negatives. The ongoing battle between the antivirus industry and the malware writers is making this situation unreachable as in [9]. Malware writers create a virus

and then antivirus writers detect it. Once the malware writers find out, they modify it to defeat the previous detection, and the process just keeps repeating. Unless there are no more malware, antivirus writers have to continue their efforts to detect and to eradicate malware.

5.2.1 Accuracy Measure

There are four possible outcomes for detection: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). A detection is considered a true positive when a virus is correctly classified as a virus, whereas it is a true negative when a normal file is correctly marked as non-malicious. TP and TN are the desirable outcome from any detection. False positives occur when a normal file is mistakenly classified as a virus. On the other hand, false negatives occur when a virus file is not detected as a virus, and thus passes the scan as a normal file. For antivirus makers, the goal is to have false positives as low as possible. Users tend to be unhappy or even angry if an important normal file is identified as malicious, but tend to be more forgiving for false negatives. In any case, both false positives and false negatives will reduce user satisfaction. Table 7 shows the four possible outcomes for detection.

Table 7: Possible outcomes for detection.

		Predicted Class	
		Virus	Normal
Actual Class	Virus	True Positive	False Negative
	Normal	False Positive	True Negative

The true positive rate (TPR) is the number of viruses correctly identified. It is calculated based on the number of true positives obtained from the total number

of all viruses tested [24]:

$$\text{TPR} = \frac{TP}{TP + FN} \tag{19}$$

The false positive rate (FPR) is the number of false positives obtained from the total number of normal files tested.

$$\text{FPR} = \frac{FP}{FP + TN} \tag{20}$$

The overall success rate, also know as the *accuracy rate*, is the total number of correct classifications obtained from the total virus files and normal files used on testing.

$$\text{Accuracy Rate} = \frac{TP + TN}{TP + TN + FP + FN} \tag{21}$$

Finally, the error rate is one minus this accuracy rate.

$$\text{Error Rate} = 1 - \frac{TP + TN}{TP + TN + FP + FN} \tag{22}$$

5.2.2 Mean Maximum Accuracy

Since we performed five-fold cross-validation, we have a total of five models for this test. Each of these folds will give an accuracy rate, and with these values we need to evaluate the performance of the algorithms tested. Normally, a single value is easier for comparing the performance. We propose to use the mean of the values obtained from the five-fold cross-validation, an operation called mean maximum accuracy rate (MMA):

$$\text{MMA} = \frac{1}{5} \sum_{i=1}^{5} Accuracy_i \tag{23}$$

5.2.3 Receiver Operating Characteristic

The receiver operating characteristic (ROC) was developed for applications in signal detection theory [25]. ROC curves have gained popularity in the machine learning community as a tool for evaluating the performance of various algorithms [26, 27].

Typically, a ROC curve is represented as a two-dimensional plot, in which the X-axis is assigned to the false positive rate, and the Y-axis is assigned to the true positive rate. Sometimes the false negative rate is used instead of the true positive rate. In the context of virus detection, the Y-axis represents the true detection rate corresponding to the true positive rate, and the X-axis represents the false detection rate (i.e., a non-virus file identified as a virus by the algorithm).

The true positive and false positive rates are commonly represented in percentage format. The true positive rate is obtained by using the true positive count divided by the sum of true positive and false negative counts, and then multiplied by 100 to get a value which represents a percentage. Figure 7 illustrates an example of some ROC curves.

An algorithm that performs random classification with 50% accuracy will generate a diagonal line in the ROC space. Anything that lies on the top left portion of the graph represents a better classification with higher true positive rate. The blue lines with the diamond-shaped markers from Figure 7 indicates such ideal classification with 100% true positive rate with 0% false positive rate. The line with red squared-shape markers correspond to an algorithm that achieves 78% true positive rate with 10% false positive rate. The triangle-shaped markers show that this algorithm is performing poorly in this experiment. We will present the ROC

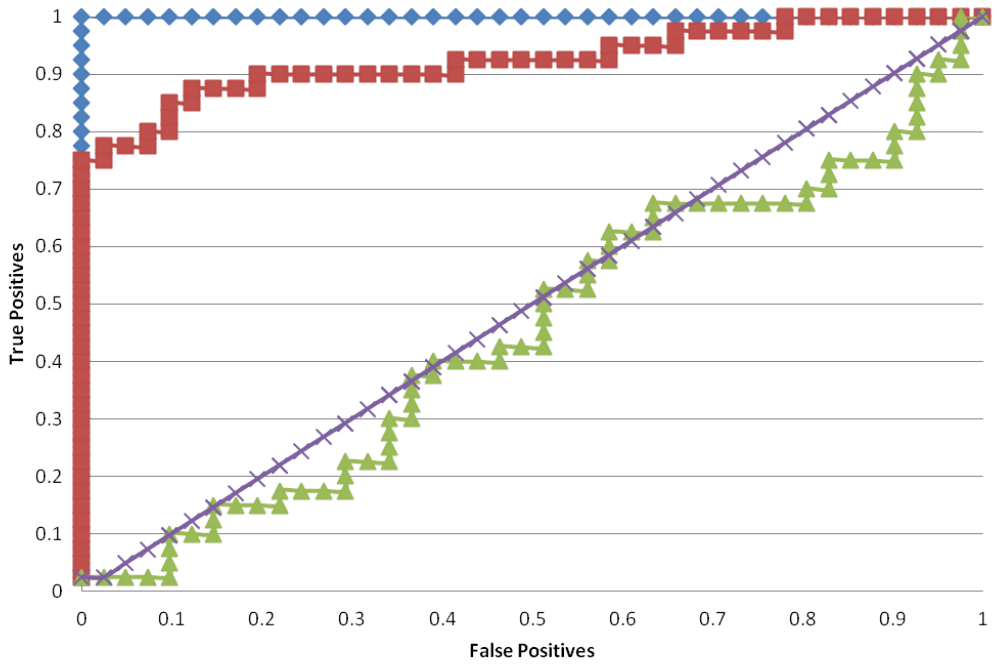

Figure 7: Example ROC curves.

curves for the HMM detector, the CSD estimator, and our proposed method to compare their performance on the metamorphic virus detection problem.

CHAPTER 6

Experimental Setup

6.1 Datasets

The datasets used in this project are 200 base NGVCK family viruses and 40 Cygwin utility files obtained from [7, 12, 28]. These 40 Cygwin utility files will be used to represent the benign program files. We disassembled these executable files by using IDA (Interactive Disassembler) Pro [29] to generate assembly files from which 80x86 instructions have been extracted for experiments. Figure 8 illustrates this process.

These virus and normal assembly files will be used as the input data to create morphed versions of the virus by either employing dead code insertion and/or subroutine copying from the normal files. Lin's metamorphic virus generator was modified to enable automatically generation of various morphed variants of these 200 virus files with these 40 normal files [8]. Figure 9 shows the basic steps taken to generate the various metamorphic versions of the new virus.

The parameter for the metamorphic virus generator is set to generate increments of 10% dead code and 10% subroutine code until the maximum of 40%.

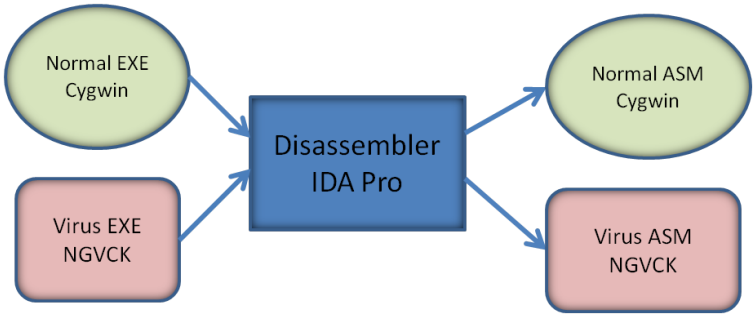

Figure 8: Process for obtaining assembly files.

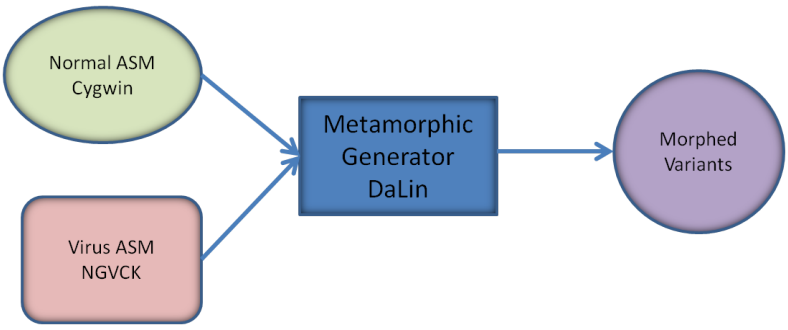

Figure 9: Process for generating metamorphic viruses.

Some example parameters are: 0% dead code with 10% subroutine code, 0% dead code with 20% subroutine code, 10% dead code with 10% subroutine code. There are 25 combinations of possible parameter values, each having 200 files, resulting in a total of 5000 files.

6.2 Data Processing

The first step is to separate the files into five groups for the five-fold cross-validation experiments. One set is used for validation during testing phase, while the remaining four subsets are used for training. From the 200 virus files, we selected 160 files used for the training, and the remaining 40 files will be used for testing purposes. In these five folds, each fold will have a set of different training file built from 160 virus files and 40 virus files used for testing.

The assembly files were read by our program and then converted to a format compatible with both the χ^2 statistical framework that we developed for this thesis, called the chi-squared distance estimator (CSD) and the HMM-based detector. The processing of these input assembly files requires several steps:

- Concatenate the randomly selected files to create one training file.
- Each of these assembly files is read in line by line to check the validity of each

line. Blank lines or labels, and lines that start with DATA or CODE, are skipped. In addition, if a line is solely composed of a comment (meaning that it starts with a semicolon), it is promptly removed. Ultimately, the line will be saved for additional processing.

- For each saved line, we need to remove those lines which do not contain valid instructions. For this purpose, we use the list of all known 80x86 instructions, and any line that does not contain a valid instruction is simply removed.
- For all files within the training set of one fold, we build a dictionary by computing a unique index for each instruction, from zero to N. N is the total number of unique instructions encountered within the training partition.
- Using the dictionary containing the unique mapping between an instruction and an index, we represent all files within a fold as a sequence of indices corresponding to their respective instructions.

The database of instructions was created by analyzing the Core / Pentium / Phenom documentation from both AMD and Intel. We present the complete list of allowed instructions in Appendix A.

In order to train the hidden Markov model (HMM), the training files are concatenated together within each fold. This results in a rather large file, containing the instructions from 160 training files. It is important to note that the mapping from an instruction to an index is used for both the training files, and also the testing files. In particular, it is possible that in the testing files, there may be some instructions which have not been seen during the training. There are two ways to handle this: (1) those instructions which miss an index could be ignored; or (2) we can assign a special index (N+1) to represent this fact. This required tuning on the

HMM, which is done by assigning a very low probability of ever encountering this symbol, will allows detection of "suspicious" files. For example, if the training is done on normal files (as opposed to virus files), and the estimation on a virus file, it is possible that the virus writer is using some exotic instructions. Thus, these instructions should also influence the final score, making it less likely to be a normal file.

We call the mapping between an instruction and an index "an alphabet." The first line contains the number of observation symbols (unique instructions). The following lines contain the ordered list of instructions (ordered in the chronological appearance in the training set).

We use the same training input files and testing input files for HMM and the CSD experiments. We implemented a Java program to create the input files for both the HMM and CSD frameworks as we described. The training observation sequence and all the testing observation sequences will need to be processed into a format that the frameworks can understand. Each fold has one alphabet file that is used to build the rest of the input files. The instruction (which we will also refer to as opcode) from these observation sequences is remapped into numbers by using the alphabet file. To further emphasize the remapping process, the first opcode will now have index 0, and the second opcode will have index 1, and so forth. Any opcode not in this alphabet file will receive a large number, equal to the size of the alphabet, as its index. For example, if there are 10 opcodes in the alphabet file, the indices are ranging from 0 to 9, then 10 indicates that the opcode does not appear in the training file. All unseen opcodes are grouped together using the same index. This is intended to make sure all opcodes are considered when building the validation input files.

6.3 HMM-based Detector Framework

Once all the input files are created, we pass the training input files as an argument into the HMM program to build the model file. The maximum iterations for HMM training is set at 800 and the program terminates either when it converges or when the maximum iterations are reached. We train the HMM model using the HMM detector from [7] to ensure consistency amongst the training and testing performed. After HMM is trained with the input files, the output model files are used for computing the log-likelihoods per opcode on the rest of the validation input files.

The HMM detector will compute the log likelihood instead of raw probability to avoid the issue of underflow. Probability usually yields results between 0 and 1, thus when floating point numbers are very close to zero, and are multiplied with similarly small numbers, the outcome is an underflow which will eventually result in a "NaN". Therefore, log likelihood is used whenever we deal with probability problems. We will abbreviate the HMM log likelihood score as HMMLL.

The evaluation of the HMM is very fast. The reason for this operation's speed is that only a single pass over the data is necessary to compute the result. We compute the log-likelihood for all the test files, then we write these scores into one score file per fold.

6.4 CSD Estimator Framework

The CSD estimator framework is significantly faster when compared to HMM training. Since the CSD estimator uses calculations that are based on the frequency of opcodes, it requires very little memory and only one single pass over the training data.

The CSD estimator determines how far a file is from the expected distribution of instructions. The E value, which is the expected file, can be obtained using the same training file created for HMM training. It is used as the base comparison file to see how other files are similar to it. The O symbol, which stands for the observed file, is the testing file for HMM. All the instructions from the expected file will be saved along with the total number of instructions from this file. We normalized these counts by dividing each instruction with the total number of all instructions. Each of the observed files is also normalized, just like the expected file, before calculating the CSD value. The same expected file is used on one fold of cross-validation set, while the observed files are the testing datasets. All of the instruction frequencies from the expected files are retrieved and used to compare with the instruction frequencies from each of the testing files.

For every instruction in the expected file it computes its contribution to the D^2 value. If there are instructions from the observed file which do not exist in the expected file, their D^2 contribution will be set to zero since dividing by zero will yield an undefined value. However, if that instruction is not found in the observed file, then the value is equal to the count from the expected file. The final CSD score, which is equal to the D^2 value, will be used in the experiments.

6.4.1 Bigram Frequencies Test

This test is similar to building the single instruction (unigram) frequency table on Section 6.2. Two consecutive instructions are connected with an underline (_) to build a new alphabet, for example, a `JMP` follows by a `MOV` will be represented as "`jmp_mov`". In this experiment, we build two different sets of input files: the alphabet in which (1) the ordering of the pair instructions sequence does matter; (2)

the order of instruction pairs does not matter. For example, the alphabet key built for the first case will yield "`jmp_mov`" for instructions pair of `MOV` and `JMP` regardless of the ordering. However, in the second case, the alphabet "`jmp_mov`" represents two consecutive instructions of `JMP` followed by `MOV`, and "`mov_jmp`" will represent the instruction pair of `MOV` followed by `JMP`.

6.5 Threshold and Evaluation Framework

Our framework will consider all possible scores obtained from the HMM detector and CSD estimator as thresholds. Depending on the type of training file, the score which is set as the threshold will consider all the scores that are below it as corresponding to a virus file and everything else as a normal file. Then, the program will calculate the TP, TN, FP and FN from all of the thresholds. These four metrics will be used to obtain the ROC curves and accuracy scores.

The ROC curve is built by getting the TPR and FPR using the Equations 19 and 20. These values will be written to a "roc" file which is used to plot the ROC curve. The next step is computing the accuracy using Equation 21. For all possible thresholds, we obtained all the corresponding accuracy rates and wrote them to a "score" file.

The final assessment will be the evaluation framework. Our mean maximum implementation will go through each of these score files, and then return the maximum accuracy rate. Since we have five-fold cross-validation, we have five maximum rates - one for each fold. We then compute and compare the mean values for easier and clearer evaluation (only need to compare two numbers to see which one has a higher score). Finally, we will use these mean maximum scores to determine the performance of these two detection algorithms in Chapter 7.

CHAPTER 7

Results

In this section, we present the results of our experiments using the hidden Markov model-based detector (HMM detector), chi-squared distance-based estimator (CSD estimator), and our proposed hybrid virus detector. We used the Receiver Operating characteristic (ROC) curve and the mean maximum accuracy (MMA) rate to evaluate the performance of the three detectors. See Appendix C for the ROC curves for all tests performed.

For consistency, we used the same set of data for all three detectors. Table 8 summarizes the five types of models we set up for these experiments. Moreover, we will present the results of the HMM model trained on two hidden states since previous studies [7, 11] show that the number of hidden states have minor impact on the performance of HMM. The improvement is negligible while training a model requires much longer time when using more hidden states. The bigrams' approach achieves similar or lower scores as the unigram approach, therefore, we will mainly consider the hybrid model using the unigram CSD. The MMA scores for the bigrams are listed in Appendix D. Our baseline consists of the performance of the various detectors on the unmorphed viruses.

7.1 Training Set: Original Viruses

We used 80% of the 200 original NGVCK viruses without any morphing as training data. For testing purposes, we used the remaining 20% of the files, and all of the normal files. See Appendix B for discussion of the HMMLL and CSD scores. Table 9 contains the MMA scores obtained using the HMM detector, the CSD

Table 8: Files setup per parameter pair for each experiment.

Training Dataset
Normal
Original viruses
Viruses morphed with 10% dead code
Viruses morphed with 10% subroutine code
Viruses morphed with 10% dead code and 10% subroutine code

estimator, and the hybrid model on all the test sets. The score which is in bold (per row) indicates that the detector performs the best among the other detectors tested in that category.

HMM: The MMA rates for the HMM detector ranged from 77.25% to 99.75%. The HMM detector performed best when the test set was the unmorphed viruses. The lowest score occurred when the testing was done with 40% subroutine code morphing. Viruses which were morphed with dead code alone were not very effective, since the HMM still achieved an accuracy rate of more than 90.25%. The ROC curve confirmed this finding as illustrated in Figure C.14. On the other hand, as we increased the percentage of the subroutine code, it was much harder for HMM to correctly detect the viruses. When the morphing algorithm used 40% subroutine code, the MMA decreased by 22.50%. However, when the algorithm used 40% dead code, the MMA only decreased by 9.50%. See Figure C.15 for the ROC curves.

The overall average MMA for the HMM detector was 84.09%, with an error rate of 15.91%.

CSD: The MMA scores for the CSD estimator ranged from 71.46% to 99.50%. The best MMA rate occurred when trying to detect unmorphed viruses, while the worst rate occurred when the viruses were morphed with 40% dead code and 40% subroutine code. Both morphing with dead code and subroutine code contributed to

the lower accuracy rate for the CSD estimator. When using 40% subroutine code for morphing, the performance dropped by 23.66% compared to the baseline. The dead code variants caused a performance drop of 21.68%. The ROC in Figure C.16 illustrates the performance of virus detection with varying amounts of subroutine code embedded, while Figure C.17 shows the viruses with dead code embedded.

The overall average MMA rate was 79.43% which was almost 5% lower when compared to the performance of the HMM detector.

Hybrid Model: The hybrid model performed better than both the HMM and CSD with an overall average accuracy rate of 84.94%. Our hybrid model achieved 2.16% better accuracy rate than the HMM, and a 6.82% improvement over the CSD estimator. These results show that depending on the morphing parameters, our proposed hybrid approach either outperformed, or at worst, was on par with the best of the HMM and CSD methods.

7.2 Training Set: Viruses Morphed with 10% Dead Code

The training dataset consists of virus files which were already morphed with 10% dead code. Table 10 summarizes the MMA scores obtained by the HMM detector, the CSD estimator, and the hybrid model. See the ROC curves for the HMM detector in Figures C.18 and C.19. The ROC curves for the CSD estimator are shown in Figures C.21 and C.20.

HMM: The HMM detector performed best when the testing files were variants of the virus morphed only with dead code. There was a clear distinction between these virus variants and the normal files with MMA rates of more than 99.50%, and this model even achieved 100% accuracy rate when the comparison virus set was morphed with more than 30% dead code and no subroutine code. This implies that

Table 9: MMA for all detectors. Training on unmorphed viruses.

Dead Code	Subroutine	HMM	CSD	Hybrid
0%	0%	**99.75%**	99.50%	**99.75%**
	10%	87.50%	85.81%	**88.96%**
	20%	82.00%	81.58%	**84.20%**
	30%	79.75%	80.35%	**81.69%**
	40%	77.25%	76.84%	**78.94%**
10%	0%	92.00%	90.81%	**93.48%**
	10%	86.00%	83.87%	**88.47%**
	20%	83.00%	79.60%	**84.96%**
	30%	78.50%	76.90%	**83.46%**
	40%	77.50%	74.89%	**80.19%**
20%	0%	91.50%	86.39%	**92.98%**
	10%	85.50%	82.66%	**88.47%**
	20%	82.75%	78.91%	**85.71%**
	30%	80.25%	75.91%	**82.20%**
	40%	77.75%	74.41%	**79.95%**
30%	0%	90.75%	81.91%	**92.22%**
	10%	86.50%	79.16%	**88.47%**
	20%	82.25%	76.47%	**85.71%**
	30%	81.50%	74.16%	**82.20%**
	40%	79.50%	73.21%	**82.20%**
40%	0%	90.25%	77.82%	**91.47%**
	10%	85.75%	76.00%	**89.22%**
	20%	83.00%	73.97%	**84.96%**
	30%	81.25%	73.23%	**84.21%**
	40%	80.50%	71.46%	**82.20%**
Average		84.09%	79.43%	**86.25%**

the obfuscation technique used on Lin's metamorphic generator applied many similar junk instructions throughout the morphing process. The performance weakened when the test viruses were morphed with an increasing amounts of subroutine code and the lowest score occurred when the viruses were morphed with 40% of subroutine code. Therefore, the higher percentage of the dead code in the viruses variants, the easier it is for HMM to detect them. On the other hand, higher percentage of subroutine code has a negative impact on the HMM detector's performance.

The overall average MMA for the HMM detector was 88.12%, with an error rate of 11.88%.

CSD: The MMA scores of the CSD estimator for this experiment ranged from 71.50% to 95.75%. The best accuracy rate occurred when the test set was the same variant of virus. It performed notably weaker when the test sets consisted of only variants morphed with subroutine code, and the lowest accuracy occurred for viruses that were morphed with 40% subroutine. However, it obtained MMA rates over 92.00% on variants which were only morphed with dead code.

In general, the HMM detector performed better than CSD in this experimental setup since the average MMA rate for HMM was 88.12% while the rate for CSD was 82.72%.

Hybrid Model: The new model we proposed performed better than both the HMM and CSD estimator in this test setup as well. The overall average accuracy rates for the new model was 89.40%, which was an improvement of 1.28% compared to HMM detector and 6.68% improvement compared to CSD.

Table 10: MMA for all detectors. Training on viruses with 10% dead code.

Dead Code	Subroutine	HMM	CSD	Hybrid
0%	0%	**99.50%**	92.00%	99.24%
	10%	87.75%	80.75%	**88.46%**
	20%	81.75%	76.00%	**83.95%**
	30%	80.00%	73.50%	**81.18%**
	40%	76.75%	71.50%	**77.18%**
10%	0%	**99.75%**	95.75%	99.24%
	10%	**90.75%**	86.75%	90.47%
	20%	85.75%	80.00%	**86.46%**
	30%	81.75%	77.25%	**83.46%**
	40%	80.00%	75.25%	**81.20%**
20%	0%	**99.75%**	94.00%	98.49%
	10%	90.25%	87.25%	**91.73%**
	20%	86.25%	81.75%	**88.21%**
	30%	82.75%	78.25%	**86.71%**
	40%	81.00%	76.00%	**84.20%**
30%	0%	**100.00%**	93.00%	98.24%
	10%	**93.50%**	87.25%	93.47%
	20%	88.75%	82.75%	**91.22%**
	30%	84.00%	78.00%	**87.46%**
	40%	83.00%	77.75%	**86.96%**
40%	0%	**100.00%**	92.50%	97.99%
	10%	93.25%	88.00%	**93.47%**
	20%	87.75%	82.75%	**90.97%**
	30%	85.25%	81.50%	**89.22%**
	40%	83.75%	78.50%	**88.46%**
Average		88.12%	82.72%	**89.40%**

7.3 Training Set: Viruses Morphed with 10% Subroutine Code

In this experiment, the HMM model was built from virus variants that were morphed randomly with 10% subroutine code from the normal files. For the full list of MMA rates of all detectors, see Table 11.

HMM: When the training set consisted of virus variants morphed with 10% subroutine code, the HMM yielded a high number of false negatives. The metamorphic virus generator made by Lin was designed to "confuse" HMM-type detectors by making the morphed viruses look like "normal files," and as such, it is expected that HMM will do poorly. The morphed files produced by this generator had scores that were close to those of normal files [8]. The HMM detector obtained MMA rates as low as 57%, while the best rate was only 71.25%. The HMM performed slightly better as the percentage of dead code increased but worse when the percentage of subroutine code increased. The ROC curves also show that HMM was doing poorly as illustrated in Figures C.23 and C.22.

The overall average MMA rate for the HMM detector was merely 63.46%.

CSD: The CSD estimator performed surprising well when the virus was morphed with normal files. The MMA results ranged from 86.75% to 96.25%. These results showed that the CSD estimator performed better than HMM when applying this obfuscation technique of morphing subroutine code from normal files. The ROC curves are shown in Figures C.25 and C.24.

The CSD estimator has an overall average MMA rate of 89.74% while HMM only achieves a rate of 63.46%.

Hybrid Model: Similarly, the new model performed well under this experiment. The overall average MMA rate was 91.23%, which was higher than HMM's 63.46%

Table 11: MMA for all detectors. Training on viruses with 10% subroutine code.

Dead Code	Subroutine	HMM	CSD	Hybrid
0%	0%	62.25%	96.25%	**97.24%**
	10%	60.50%	92.25%	**94.97%**
	20%	59.25%	90.00%	**92.47%**
	30%	58.25%	88.50%	**90.72%**
	40%	57.00%	86.75%	**88.71%**
10%	0%	61.75%	95.00%	**97.23%**
	10%	60.50%	92.50%	**96.22%**
	20%	59.50%	90.00%	**93.72%**
	30%	58.25%	88.75%	**92.22%**
	40%	59.00%	88.50%	**91.22%**
20%	0%	65.75%	92.00%	**95.23%**
	10%	64.25%	90.25%	**94.23%**
	20%	64.00%	89.50%	**92.72%**
	30%	62.25%	88.75%	**91.96%**
	40%	59.75%	88.25%	**90.97%**
30%	0%	68.75%	89.50%	**91.96%**
	10%	68.00%	89.75%	**92.21%**
	20%	67.00%	90.25%	**92.46%**
	30%	64.75%	88.50%	**90.71%**
	40%	63.75%	88.25%	**90.96%**
40%	0%	71.25%	87.50%	**90.45%**
	10%	70.25%	88.75%	**91.46%**
	20%	69.25%	88.75%	**91.46%**
	30%	66.00%	87.75%	**90.96%**
	40%	65.25%	87.25%	**89.71%**
Average		63.46%	89.74%	**92.48%**

by 29.02% and higher than CSD's 89.86% by 2.74%. In this experiment, it outperformed the two detectors for every possible morphing parameter combination.

7.4 Training Set: Viruses Morphed with 10% Dead Code and 10% Subroutine Code

In this experiment we examined the scenario in which the training is done on virus variants which have been morphed with 10% dead code and 10% subroutine code from random normal files. See Table 12 for comprehensive MMA results.

HMM: The HMM detector performed poorly as long as the model was trained on virus variants which were morphed with subroutine code from normal files. The previous experiments also confirmed the finding that Lin's method of inserting subroutine code from normal files is very effective in confusing the HMM detector. For this experiment, the highest MMA rate was 78.25% when the viruses were morphed with 40% dead code, while the lowest was 57.00% when the viruses were morphed with 40% subroutine code. The ROC curves also show that HMM detector performs poorly under this experimental setup in Figure C.26 and Figure C.27.

The HMM detector only achieved an overall average MMA rate of 66.35%, with an error rate of 33.65%.

CSD: The CSD estimator has higher MMA rates when compared to the HMM detector. The CSD estimator performed best when the viruses were morphed with 10% dead code, achieving a MMA rate of 97.75%. The lowest MMA rate occurred when viruses were morphed with 40% subroutine code, with a rate of 83.50%. This experiment again showed that the CSD estimator performed better in distinguishing viruses morphed with subroutine code. Figures C.28 and C.29 show the ROC curves for this experiment.

The overall average MMA rate of the CSD estimator was 91.45%.

Hybrid Model: The new model performed 0.16% better than the CSD estimator with an overall average MMA rate of 91.61%. The differences between the best scores and our model were less than 2.30%. However, compared to the HMM, the hybrid model has an average MMA rate which is higher by 25.26%.

Table 12: MMA for all detectors. Training on viruses with 10% dead code and 10% subroutine code.

Dead Code	Subroutine	HMM	CSD	Hybrid
0%	0%	64.50%	90.00%	**90.73%**
	10%	61.50%	**90.25%**	90.22%
	20%	60.25%	**87.75%**	87.71%
	30%	58.00%	**86.25%**	84.71%
	40%	57.00%	**83.50%**	81.20%
10%	0%	65.00%	**97.75%**	97.73%
	10%	63.00%	**95.50%**	95.47%
	20%	62.00%	**92.75%**	91.72%
	30%	60.50%	**91.00%**	90.22%
	40%	60.50%	**88.50%**	87.71%
20%	0%	69.25%	96.25%	**96.98%**
	10%	67.00%	94.00%	**94.48%**
	20%	65.00%	92.25%	**92.47%**
	30%	63.75%	90.75%	**90.97%**
	40%	60.75%	**89.50%**	88.97%
30%	0%	74.25%	95.00%	**95.23%**
	10%	73.50%	93.00%	**94.97%**
	20%	71.50%	92.75%	**93.47%**
	30%	67.75%	89.75%	**90.97%**
	40%	67.50%	89.75%	**90.97%**
40%	0%	78.25%	93.75%	**93.97%**
	10%	75.75%	93.50%	**94.48%**
	20%	73.50%	91.00%	**92.72%**
	30%	70.25%	91.50%	**91.71%**
	40%	68.50%	90.25%	**90.71%**
Average		66.35%	91.45%	**91.61%**

7.5 Training Set: Normal Files

This experiment was performed using 32 randomly selected normal files for training. The testing phase used the remaining eight normal files not used during training, and 40 variants of viruses. Table 13 summarizes the MMA scores performed by HMM detector, CSD estimator, and the proposed hybrid detector.

HMM: According to the MMA, the HMM detector performed extremely well under this experimental setup. The MMA scores ranged from 97.50% to 99.17%. We concluded that the HMM detector trained with normal files yielded an excellent detection rate and low false positive rate. Figures C.30 and C.31 show the ROC curves for this experiment.

The HMM detector achieved an overall average MMA rate of 98.32%, and an error rate of 1.68%.

CSD: The MMA scores obtained from this test were reasonably good with scores ranging from 87.50% up to 98.33%. The virus morphed with 40% dead code and 40% subroutine code had the lowest MMA score of 87.50%. The highest MMA score was obtained when compared with viruses that had no morphing, achieving a MMA rate of 98.33%. Figures C.32 and C.33 show the ROC curves for this experiment.

The CSD estimator had an overall average MMA score of 90.74%, which was 7.62% lower than the HMM detector's score of 98.32%.

Hybrid Model: In this particular experiment, our hybrid model achieved an overall average MMA rate of 98.26%. It outperformed the CSD estimator by 7.56% but performed worse than the HMM detector by 0.06%. The results show that our hybrid model achieved the same level of accuracy with the HMM detector for every possible morphing parameter combination except in three cases. This difference was

Table 13: MMA for all detectors. Training on normal files.

Dead Code	Subroutine	HMM	CSD	Hybrid
0%	0%	**99.17%**	98.33%	**99.17%**
	10%	**98.75%**	93.33%	**98.75%**
	20%	**98.75%**	91.67%	98.33%
	30%	**98.33%**	91.25%	**98.33%**
	40%	**98.33%**	91.67%	**98.33%**
10%	0%	**99.17%**	95.00%	**99.17%**
	10%	**98.75%**	91.25%	**98.75%**
	20%	**98.33%**	90.42%	**98.33%**
	30%	**98.33%**	90.42%	**98.33%**
	40%	**98.33%**	90.42%	**98.33%**
20%	0%	**98.75%**	92.92%	98.33%
	10%	**98.33%**	89.58%	97.92%
	20%	**98.33%**	90.42%	**98.33%**
	30%	**98.33%**	87.92%	**98.33%**
	40%	**98.33%**	88.75%	**98.33%**
30%	0%	**98.33%**	92.08%	**98.33%**
	10%	**98.33%**	89.58%	**98.33%**
	20%	**98.33%**	89.17%	**98.33%**
	30%	**97.92%**	88.33%	**97.92%**
	40%	**97.92%**	88.33%	**97.92%**
40%	0%	**97.92%**	91.25%	**97.92%**
	10%	**97.92%**	90.00%	**97.92%**
	20%	**97.50%**	89.58%	**97.50%**
	30%	**97.50%**	88.33%	**97.50%**
	40%	**97.92%**	87.50%	**97.92%**
Average		**98.32%**	90.70%	98.26%

not large enough to conclude that this detector performed significantly worse than the HMM, since there were only 8 normal files in the testing set, and the difference was smaller than $\frac{1}{(8\times5)}$.

CHAPTER 8

Conclusion

In this project, we made use of the frequency analysis of instructions and statistical methods in developing a detection model inspired by previous research [7, 9, 11]. Our goal was to detect metamorphic viruses created by Lin's metamorphic virus generator. We explored its weaknesses by analyzing the virus variants' ease of detection using both the HMM detector and the CSD estimator.

We found that the HMM detector performed well in detecting virus variants with dead code morphed from Lin's generator. The CSD estimator did better in detecting virus variants that had subroutine code embedded by Lin's generator. With these two attributes in mind, we set out to create a hybrid model to better detect these viruses. We designed a hybrid detection model which uses probabilistic scores from both the HMM and the CSD detectors. Since the two underlying algorithms don't perform equally well, we optimized weights for each of the detectors by performing a grid search.

The results showed that the proposed hybrid model was able to beat the scores of both the HMM detector and CSD estimator when the training files are virus variants. However, it performed at the same level as the HMM alone when the training is done on normal files.

Future work would include the investigation of more statistical models and evaluation of their performance in a similar setup as presented in this paper. Another possible extension would be to analyze other metamorphic viruses, and use a significantly larger number of normal files.

LIST OF REFERENCES

[1] "California department of motor vehicles." September 2011. [Online]. Available: http://dmv.ca.gov/portal/home/dmv.htm

[2] "BART service restored following computer problems." August 2011. [Online]. Available: http://www.bart.gov/news/articles/2011/news20110808a.aspx

[3] T. Dirro, P. Greve, R. Kashyap, D. Marcus, F. Paget, C. Schmugar, J. Shah, and A. Wosotowsky, "McAfee threats report: Second quarter 2011," McAfee, Inc, 2821 Mission College Blvd, Santa Clara, Ca 95054, Tech. Rep., 2011.

[4] F. Coulter and K. Eichorn, "A good decade for cybercrime," McAfee, Inc, 2821 Mission College Blvd, Santa Clara, Ca 95054, Tech. Rep., 2011.

[5] M. Stamp, *Information Security: Principles and Practice*, 2nd ed. Wiley, 2011.

[6] P. Ször, *The Art of Computer Virus research and defense*. Addition Wesley Professional, 2005.

[7] W. Wong, "Analysis and detection of metamorphic computer viruses," Master's thesis, San Jose State University, 2006.

[8] D. Lin, "Hunting for undetectable metamorphic viruses," Master's thesis, San Jose State University, 2010.

[9] E. Filiol and S. Josse, "A statistical model for undecidable viral detection," *Journal in Computer Virology*, vol. 3, no. 2, pp. 65–74, 2007.

[10] J. Aycock, *Computer Viruses and Malware*. Springer Science+Business Media, LLC, 2006.

[11] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[12] "Vx heavens." November 2011. [Online]. Available: http://www.vx.netlux.org/

[13] P. Ször and P. Ferrie, "Hunting for metamorphic," in *Virus Bulletin*, September 2001, pp. 123–144.

[14] R. Wang, "Flash in the pan?" in *Virus Bulletin Conference*. Virus Analysis Library, July 1998.

[15] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," in *Communications of the ACM*, vol. 18, 1975, pp. 333–340.

[16] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[17] M. Schultz, E. Eskin, and E. Zadok, "Data mining methods for detection of new malicious executables," in *International Conference on Data Mining*, 2001, pp. 38–49.

[18] J. Kolter and M. Maloof, "Learning to detect and classify malicious executables in the wild," *The Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, December 2006.

[19] F. Cohen, "Computer viruses," Ph.D. dissertation, University of Southern California, 1986.

[20] D. Chess and S. White, "An undetectable computer virus," in *Virus Bulletin Conference*, September 2000.

[21] M. Stamp, "A revealing introduction to hidden Markov models," San Jose State University, Tech. Rep., January 2004. [Online]. Available: www.cs.sjsu.edu/faculty/RUA/HMM.pdf

[22] *Intel 64 and IA-32 Intel Architecture Software Developer's Manual: Volume 2*, Intel, October 2011.

[23] S. Geisser, *Predictive Inference: An Introduction*. New York: Chapman and Hall, 1993.

[24] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., J. Gray, Ed. Morgan Kaufmann Publishers, 2005.

[25] J. Egan, *Signal Detection Theory and ROC Analysis*. Academic Press, Inc, 1975.

[26] C. Cortes and M. Mohri, "AUC optimization vs. error rate minimization," in *NIPS*, S. Thrun, L. K. Saul, and B. Scholkopf, Eds., vol. 16. The MIT Press, 2004, pp. 313–320.

[27] K. Ataman, W. N. Street, and Y. Zhang, "Learning to rank by maximizing AUC with linear programming," in *International Joint Conference on Neural Networks*, 2006, pp. 123–129.

[28] "Cygwin." September 2011. [Online]. Available: http://www.cygwin.com

[29] "Interactive dissassembler." 2011. [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml

APPENDIX A

80x86 Opcodes

Table A.14: Instruction set used to build the dictionary for data processing.

aaa	aad	aas	adc	add
addpd	addps	addsd	addss	addsubpd
addsubps	and	andnpd	andnps	andpd
andps	arpl	bound	bsf	bsr
bswap	bt	btc	btr	bts
call	cbw	cdq	clc	cld
clflush	cli	clts	cmc	cmovcc
cmp	cmppd	cmpps	cmps	cmpsb
cmpsd	cmpss	cmpsw	cmpxchg	cmpxchg8b
comisd	comiss	cpuid	cvtdq2pd	cvtdq2ps
cvtpd2dq	cvtpd2pi	cvtpd2ps	cvtpi2pd	cvtpi2ps
cvtps2dq	cvtps2pd	cvtps2pi	cvtsd2si	cvtsd2ss
cvtsi2sd	cvtsi2ss	cvtss2sd	cvtss2si	cvttpd2dq
cvttpd2pi	cvttps2dq	cvttps2pi	cvttsd2si	cvttss2si
cwd	cwde	daa	das	dec
div	divpd	divps	divsd	divss
emms	enter	f2xm1	fabs	fadd
faddp	fbld	fbstp	fchs	fclex
fcmovcc	fcom	fcomi	fcomip	fcomp
fcompp	fcos	fdecstp	fdiv	fdivp
fdivr	fdivrp	ffree	fiadd	ficom
ficomp	fidiv	fidivr	fild	fimul
fincstp	finit	fist	fistp	fisttp
fisub	fisubr	fld	fld1	fldcw
fldenv	fldl2e	fldl2t	fldlg2	fldln2
fldpi	fldz	fmul	fmulp	fnclex
fninit	fnop	fnsave	fnstcw	fnstenv
fnstsw	fpatan	fprem	fprem1	fptan
frndint	frstor	fsave	fscale	fsin
fsincos	fsqrt	fst	fstcw	fstenv
fstp	fstsw	fsub	fsubp	fsubr
fsubrp	ftst	fucom	fucomi	fucomip
fucomp	fucompp	fwait	fxam	fxch
fxrstor	fxsave	fxtract	fyl2x	fyl2xp1

Continued on Next Page...

haddpd	haddps	hlt	hsubpd	hsubps
icebp	idiv	imul	in	inc
ins	insb	insd	insw	int 3
int n	into	invd	invlpg	iret
iretd	ja	jae	jb	jbe
jc	jcxz	je	jecxz	jg
jge	jl	jle	jmp	jna
jnae	jnb	jnbe	jnc	jne
jng	jnge	jnl	jnle	jno
jnp	jns	jnz	jo	jp
jpe	jpo	js	jz	jcc
lahf	lar	lddqu	ldmxcsr	lds
lea	leave	les	lfence	lfs
lgdt	lgs	lidt	lldt	lmsw
lock	lods	lodsb	lodsd	lodsw
loop	loopcc	lsl	lss	ltr
maskmovdqu	maskmovq	maxpd	maxps	maxsd
maxss	mfence	minpd	minps	minsd
minss	monitor	mov	movapd	movaps
movd	movddup	movdq2q	movdqa	movdqu
movhlps	movhpd	movhps	movlhps	movlpd
movlps	movmskpd	movmskps	movntdq	movnti
movntpd	movntps	movntq	movq	movq2dq
movs	movsb	movsd	movshdup	movsldup
movss	movsw	movsx	movupd	movups
movzx	mul	mulpd	mulps	mulsd
mulss	mwait	neg	not	or
orpd	orps	out	outs	outsb
outsd	outsw	packssdw	packsswb	packuswb
paddb	paddd	paddq	paddsb	paddsw
paddusb	paddusw	paddw	pand	pandn
pause	pavgb	pavgw	pcmpeqb	pcmpeqd
pcmpeqw	pcmpgtb	pcmpgtd	pcmpgtw	pextrw
pinsrw	pmaddwd	pmaxsw	pmaxub	pminsw
pminub	pmovmskb	pmulhuw	pmulhw	pmullw
pmuludq	pop	popa	popad	popaw
popf	popfd	por	prefetchh	psadbw
pshufd	pshufhw	pshuflw	pshufw	pslld
pslldq	psllq	psllw	psrad	psraw
psrld	psrldq	psrlq	psrlw	psubb
psubd	psubq	psubsb	psubsw	psubusb

Continued on Next Page...

psubusw	psubw	punpckhbw	punpckhdq	punpckhqdq
punpckhwd	punpcklbw	punpckldq	punpcklqdq	punpcklwd
push	pusha	pushad	pushf	pushfd
pxor	rcl	rcpps	rcpss	rcr
rdmsr	rdpmc	rdtsc	rep	repe
repne	repnz	repz	ret	retf
retn	rol	ror	rsm	rsqrtps
rsqrtss	sahf	sal	sar	sbb
scas	scasb	scasd	scasw	seta
setae	setb	setbe	setc	sete
setg	setge	setl	setle	setna
setnae	setnb	setnbe	setnc	setne
setng	setnge	setnl	setnle	setno
setnp	setns	setnz	seto	setp
setpe	setpo	sets	setz	setcc
sfence	sgdt	shl	shld	shr
shrd	shufpd	shufps	sidt	sldt
smsw	sqrtpd	sqrtps	sqrtsd	sqrtss
stc	std	sti	stmxcsr	stos
stosb	stosd	stosw	str	sub
subpd	subps	subsd	subss	sysenter
sysexit	test	ucomisd	ucomiss	ud2
unpckhpd	unpckhps	unpcklpd	unpcklps	verr
verw	wait	wbinvd	wrmsr	xadd
xchg	xlat	xlatb	xor	xorpd
xorps				

APPENDIX B

Analysis of HMM and CSD Scores

B.1 Training on Virus Files

When the HMM is trained on virus data, the HMMLL scores for viruses are always higher than the scores of the normal files. In general, all the five-folds validation results show a clear separation between the two types with the exception of one file from one of the test sets. The resulting scores in Figure B.10 show that the virus files are different from normal files with all the viruses in this test set scored higher than -5.74, while the normal files scored below -6.60.

The CSD scores are the opposite of HMMLL scores. When training on virus files, and comparing the CSD score between a virus file and a normal file, the score for the virus should be lower than for the normal file. The lower the score produced by CSD, the more similar the compared file is to the training set. Example scores from the CSD estimator is plotted in Figure B.11. For the first fold of this experiment, using base viruses as training, the normal files have CSD scores above 0.59, while all the unmorphed virus files have scores below 0.43. Our CSD estimator performs fairly well in distinguishing the virus variants and normal files in this setup.

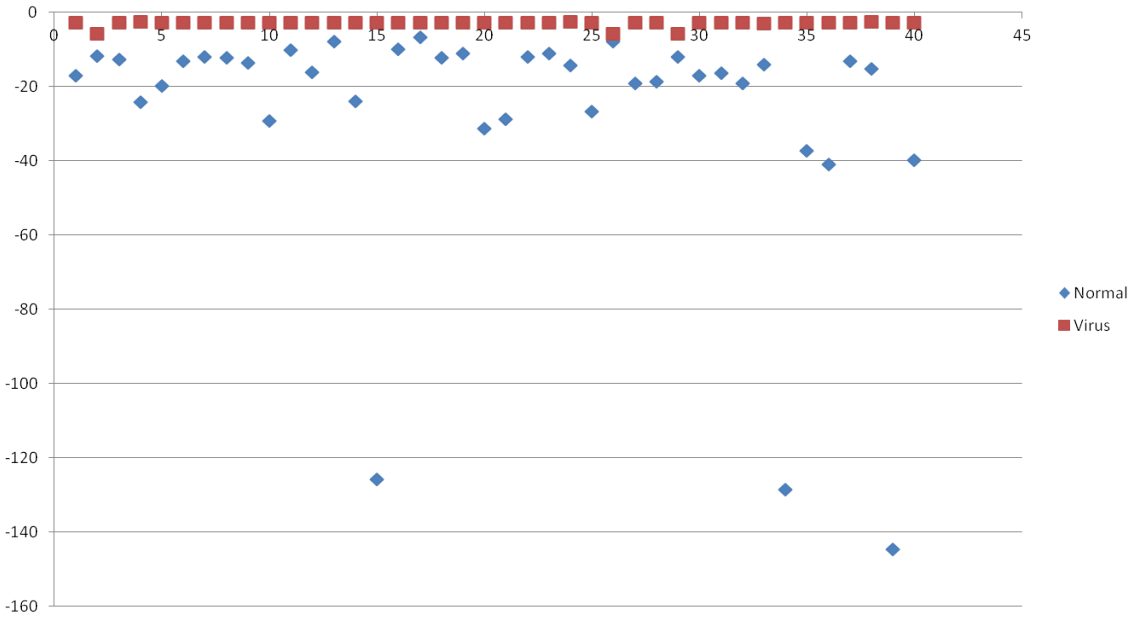

Figure B.10: HMMLL scores for base viruses generated by NGVCK.

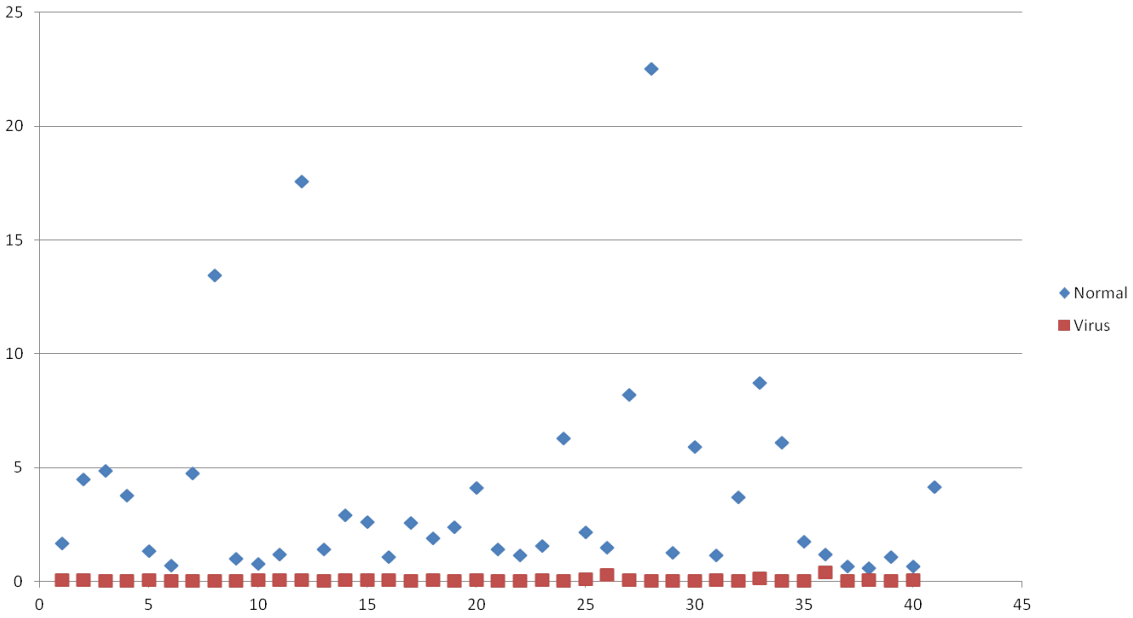

Figure B.11: The CSD scores for the base virus.

B.2 Training on Normal Files

In this experiment, when applying the HMM, the resulting scores for the normal files are higher than the virus files. All eight of the normal files score above -2.45, while all the virus files score below -17.55. Figure B.12 depicts the HMMLL computed on the first fold of the validation set with base viruses. From the figure, it is clear that the virus files are different from normal files.

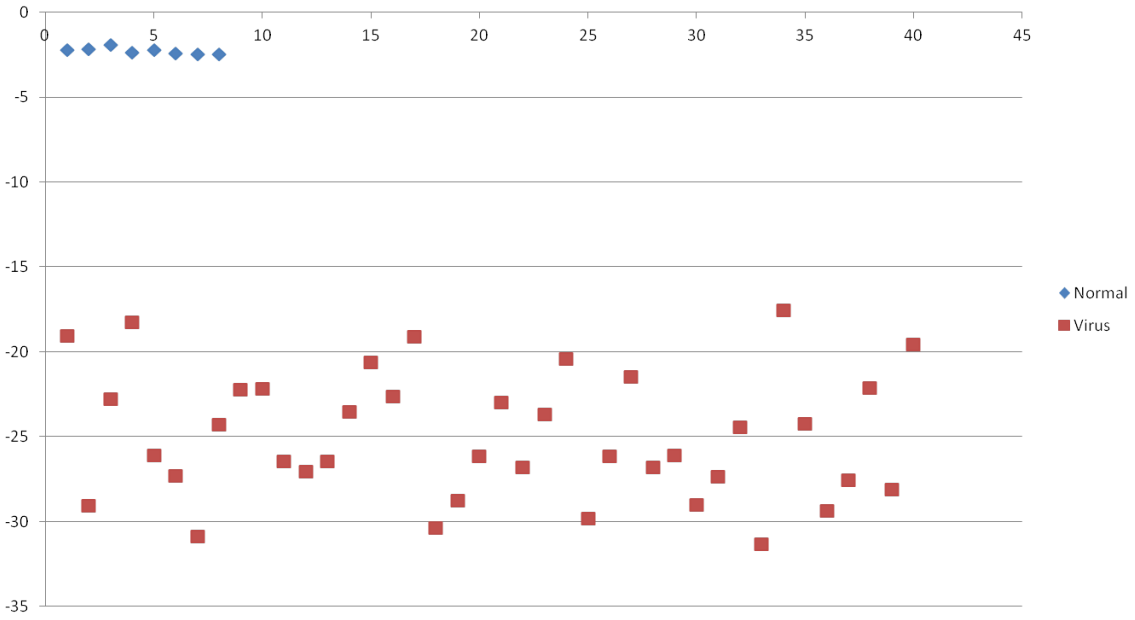

Figure B.12: HMMLL scores for HMM model trained on normal files and tested on base viruses.

The virus files scores are higher than normal files' scores. Figure B.13 shows the scores obtained from the first fold of cross-validation set from the CSD estimator. In this fold, the observed files are the base virus files. All eight of the normal files score below 0.47, while the virus files score above 2.40. Since CSD estimates how close an observed file is to the expected file, the lower the score, the more similar it is to the expected file (in this case, normal).

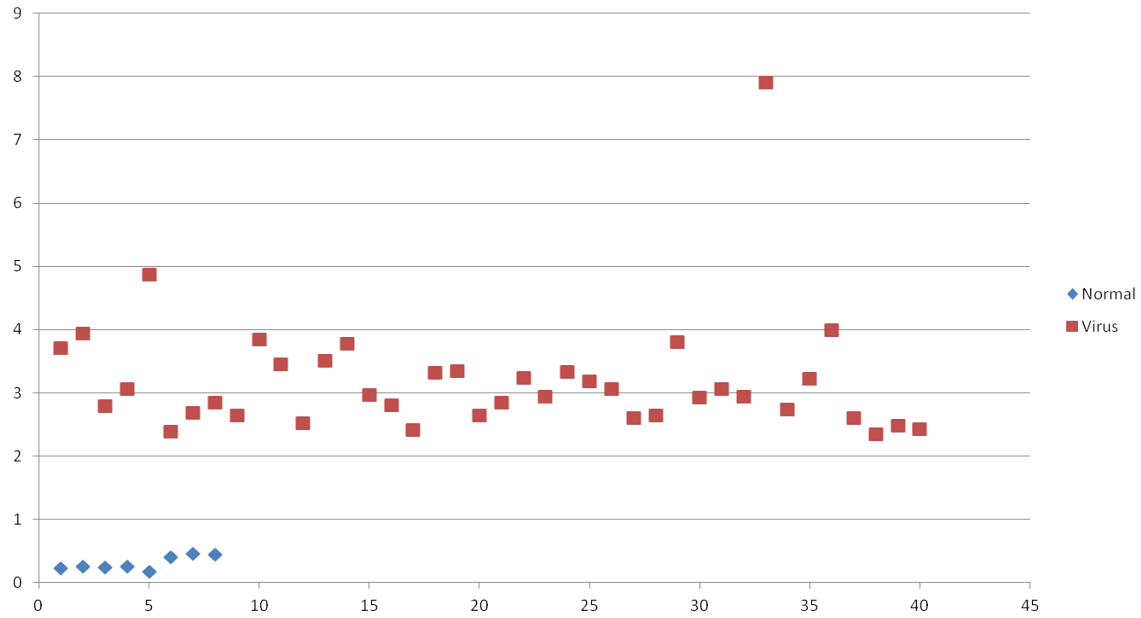

Figure B.13: The CSD scores when trained trained on normal files and tested on unmorphed viruses.

APPENDIX C

ROC Curves

We present the ROC curves in order to illustrate how dead code and subroutine morphing affect the HMM and the CSD detectors.

In the ROC graphs, we plot several parameter combinations for each detector. The legend of each graphs contains series with names such as 0s0, 0s10, or 20s10. The number before the letter "s" represents the amount of dead code with which the virus was morphed with, while the number following the "s" indicates the amount of subroutine code which was used for morphing.

C.1 Training Set: Original Viruses

Training set consists of original viruses without applying additional morphing. The ROC curves for the HMM detector is illustrated in Figure C.14 and Figure C.15.

The following two figures in Figure C.16 and Figure C.17 illustrated the ROC curves for the CSD estimator. ROC curves show that the HMM detector performed better than CSD estimator in this experimental setup.

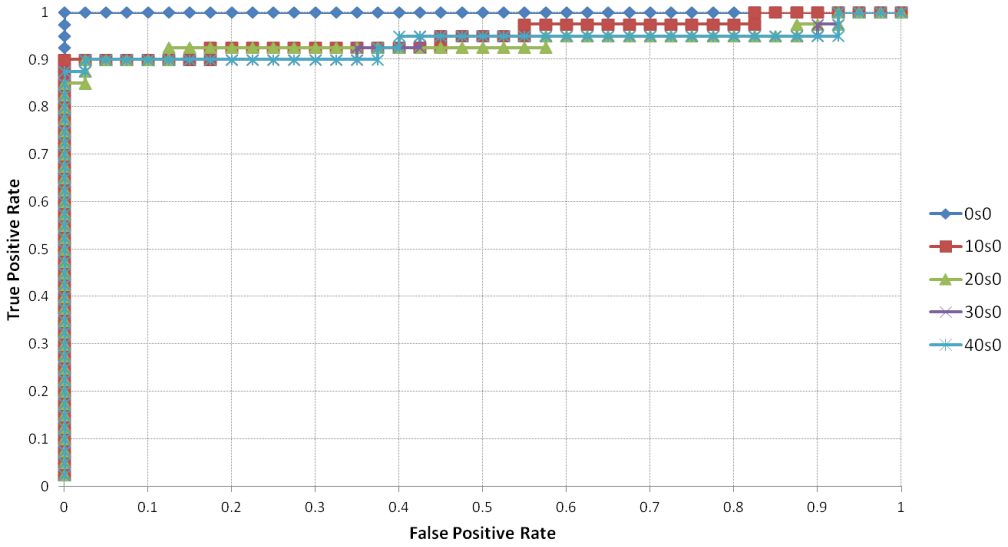

Figure C.14: ROC for the HMM detector, trained on unmorphed viruses, and tested on virus variants morphed with dead code.

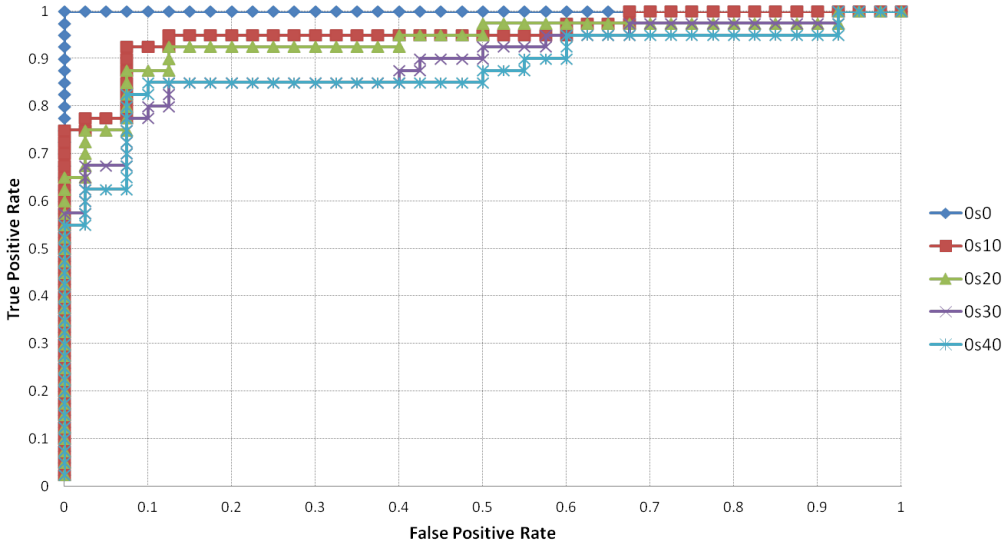

Figure C.15: ROC for the HMM detector, trained on unmorphed viruses, and tested on virus variants morphed with subroutine code.

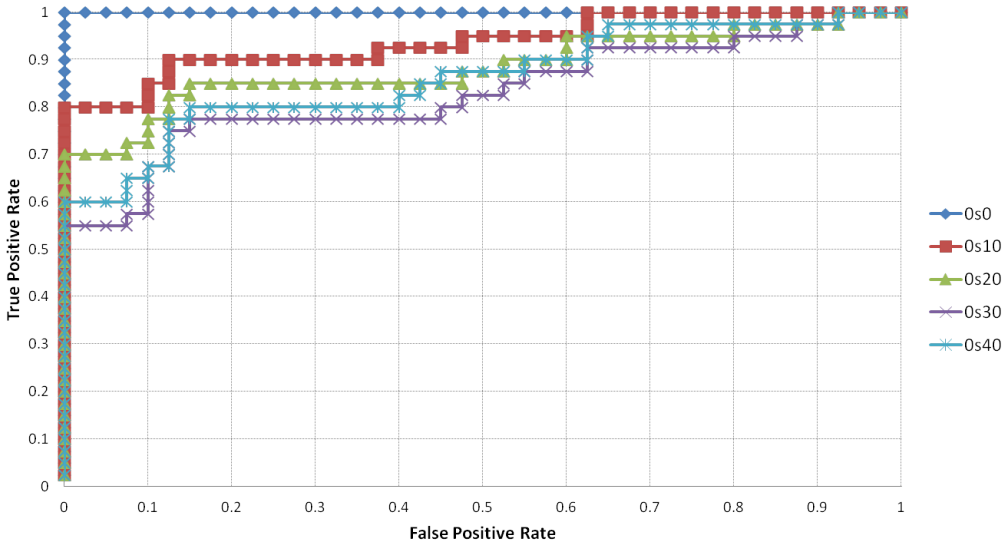

Figure C.16: ROC for the CSD detector, trained on unmorphed viruses, and tested on virus variants morphed with subroutine code.

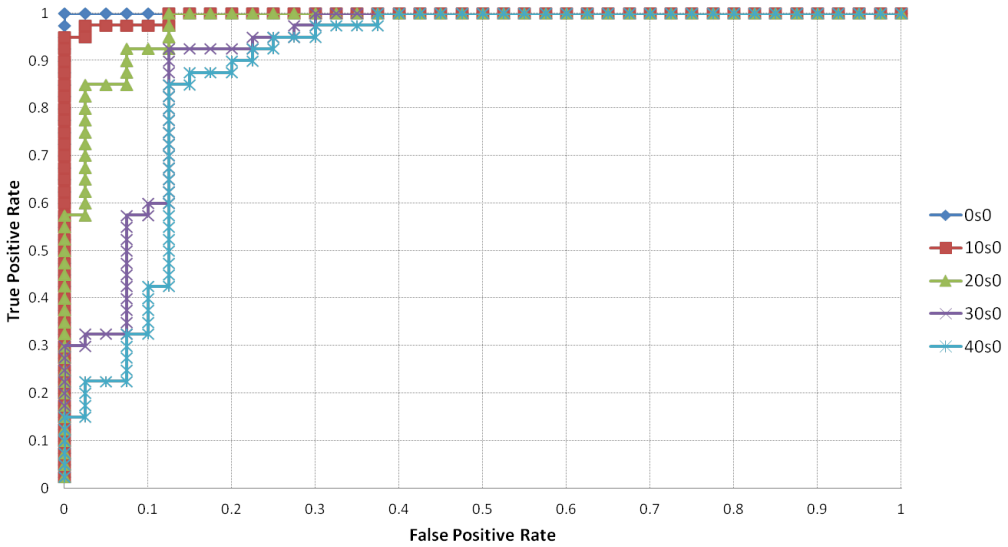

Figure C.17: ROC for the CSD detector, trained on unmorphed viruses, and tested on virus variants morphed with dead code.

C.2 Training Set: Viruses Morphed with 10% Dead Code

The training set consists of virus files morphed with 10% dead code. The ROC curves for the HMM detector is illustrated in Figure C.18 and Figure C.19. The following two figures in Figure C.20 and Figure C.21 illustrated the ROC curves for the CSD estimator. The ROC curves show that the HMM detector outperformed the CSD estimator in this experimental setup.

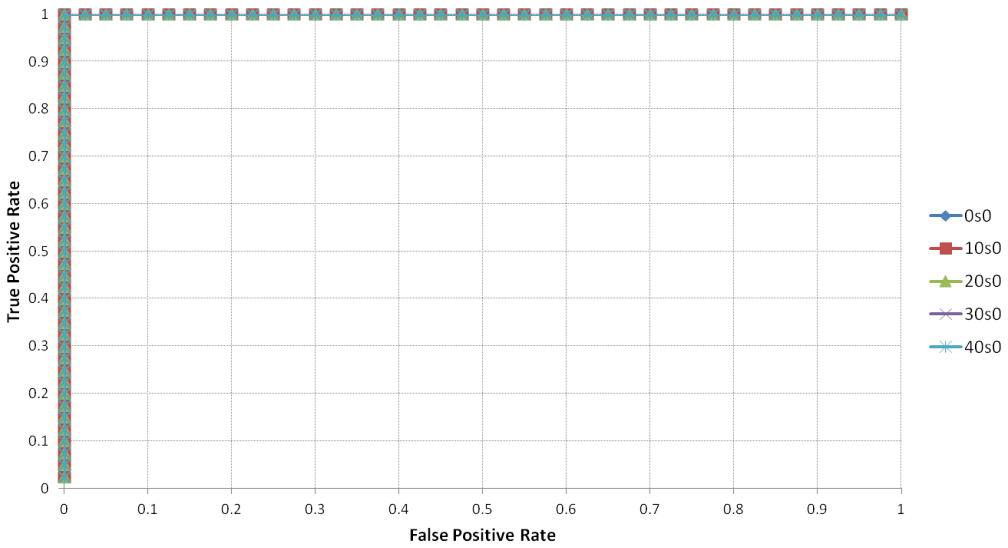

Figure C.18: ROC for the HMM detector, trained on viruses morphed with 10% dead code, and tested on virus variants morphed with dead code.

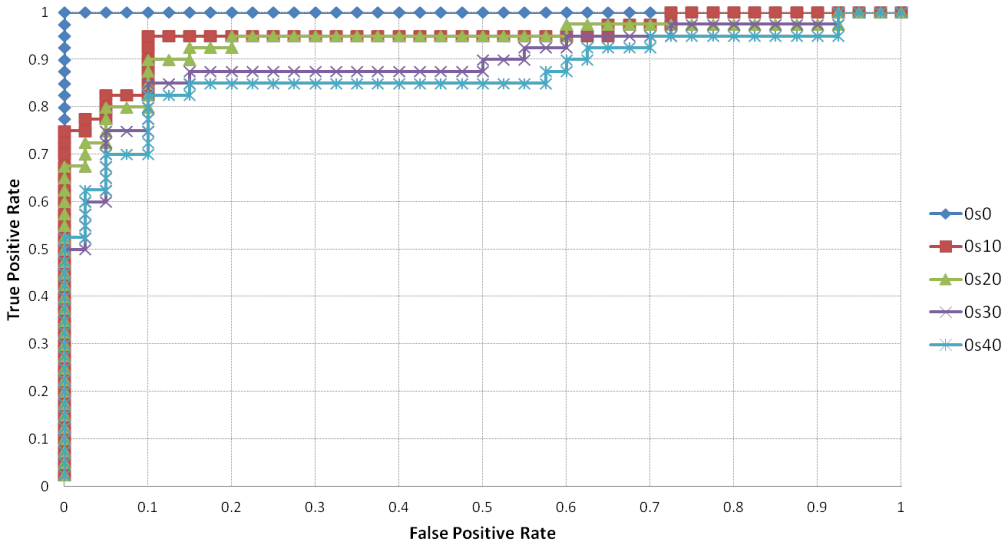

Figure C.19: ROC for the HMM detector, trained on viruses morphed with 10% dead code, and tested on virus variants morphed with subroutine code.

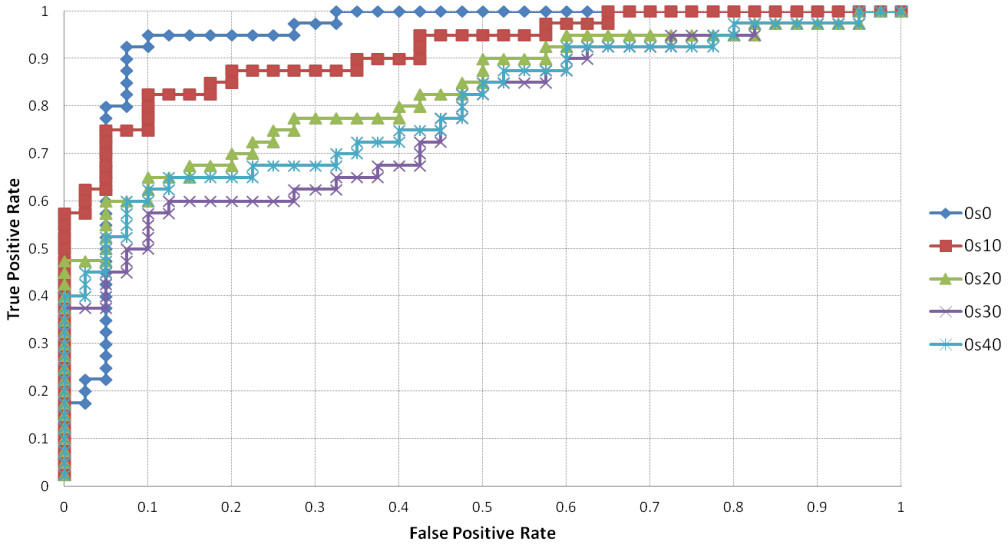

Figure C.20: ROC for the CSD detector, trained on viruses morphed with 10% dead code, and tested on virus variants morphed with subroutine code.

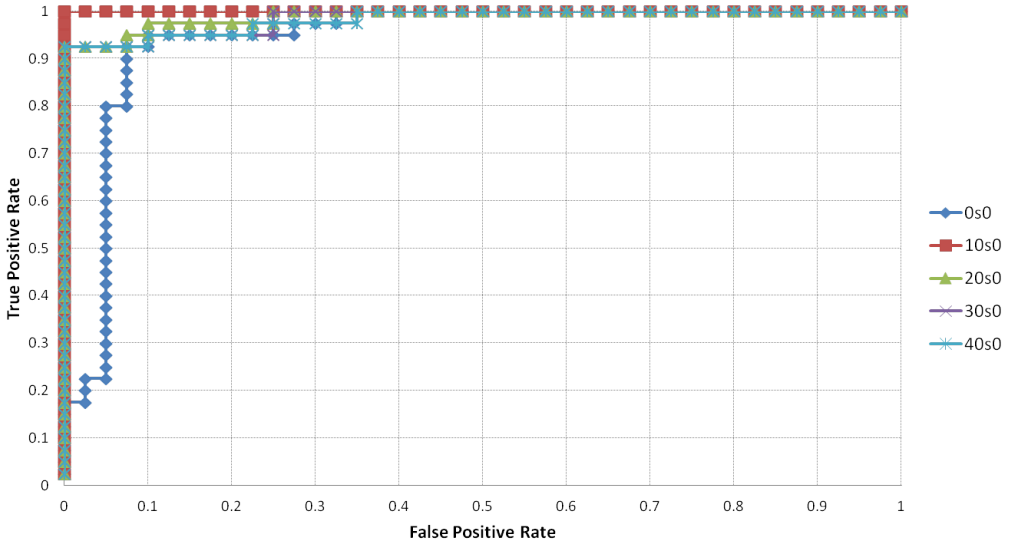

Figure C.21: ROC for the CSD detector, trained on viruses morphed with 10% dead code, and tested on virus variants morphed with dead code.

C.3 Training Set: Viruses Morphed with 10% Subroutine Code

The training set consists of virus files morphed with 10% subroutine code from normal files. The ROC curves for the HMM detector is illustrated in Figure C.22 and Figure C.23. The following two figures in Figure C.25 and Figure C.24 illustrated the ROC curves for the CSD estimator.

The ROC curves show that the HMM detector performed significantly poorer than CSD estimator when the viruses were morphed with 10% of subroutine code.

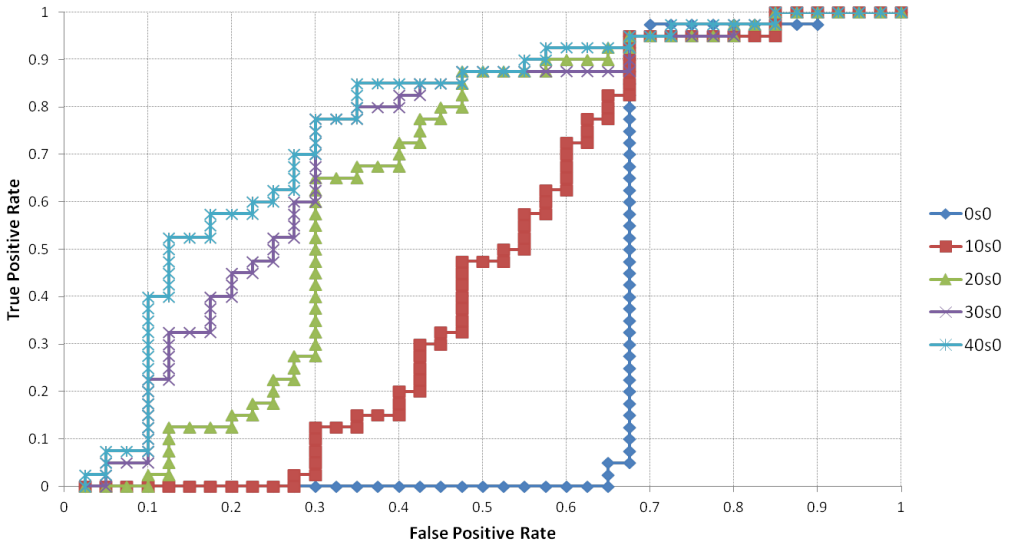

Figure C.22: ROC for the HMM detector, trained on viruses morphed with 10% subroutine code, and tested on virus variants morphed with dead code.

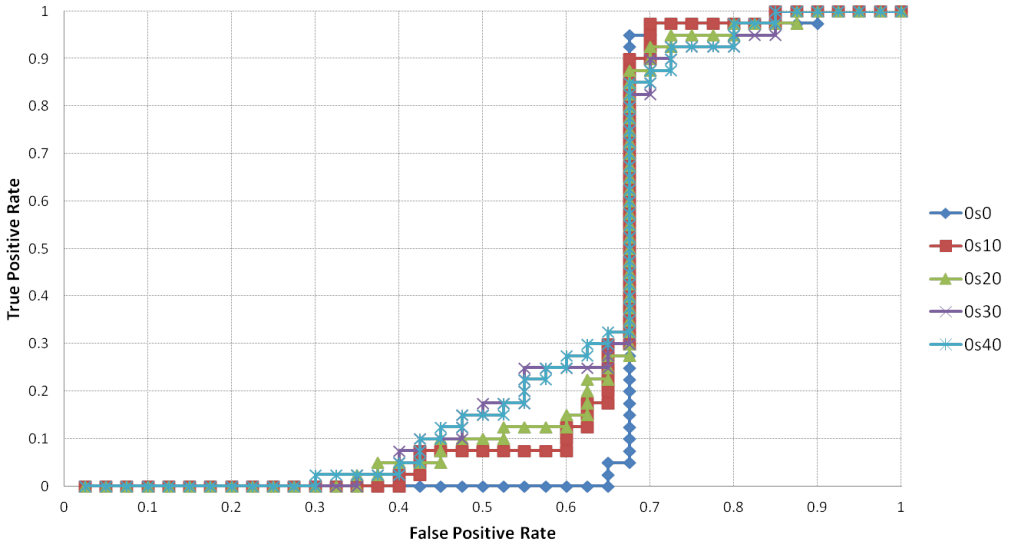

Figure C.23: ROC for the HMM detector, trained on viruses morphed with 10% subroutine code, and tested on virus variants morphed with subroutine code.

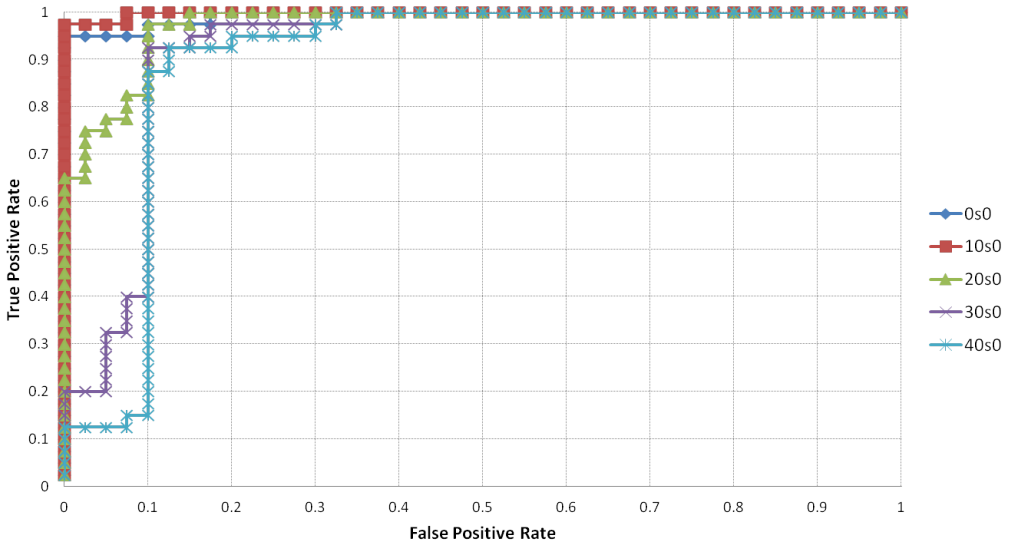

Figure C.24: ROC for the CSD detector, trained on viruses morphed with 10% subroutine code, and tested on virus variants morphed with dead code.

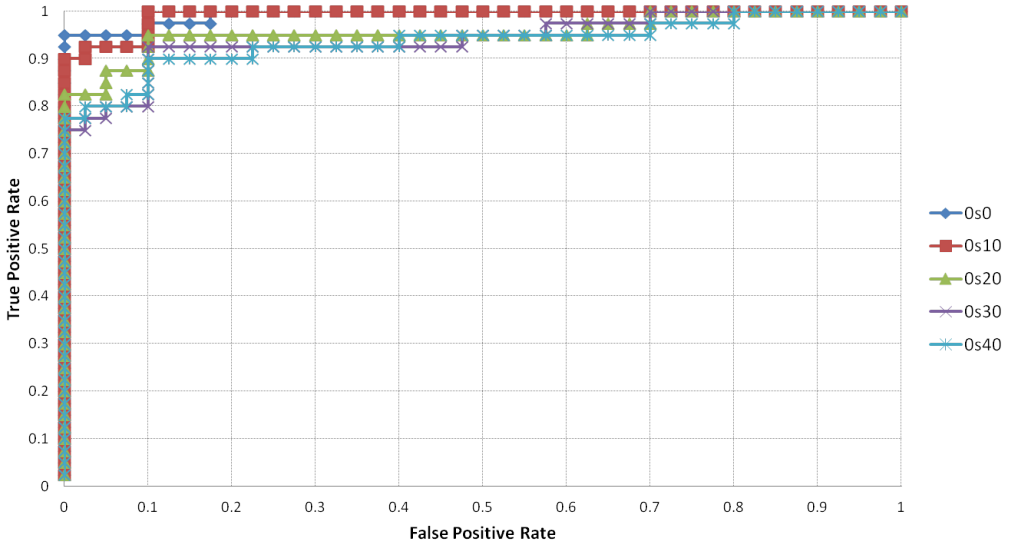

Figure C.25: ROC for the CSD detector, trained on viruses morphed with 10% subroutine code, and tested on virus variants morphed with subroutine code.

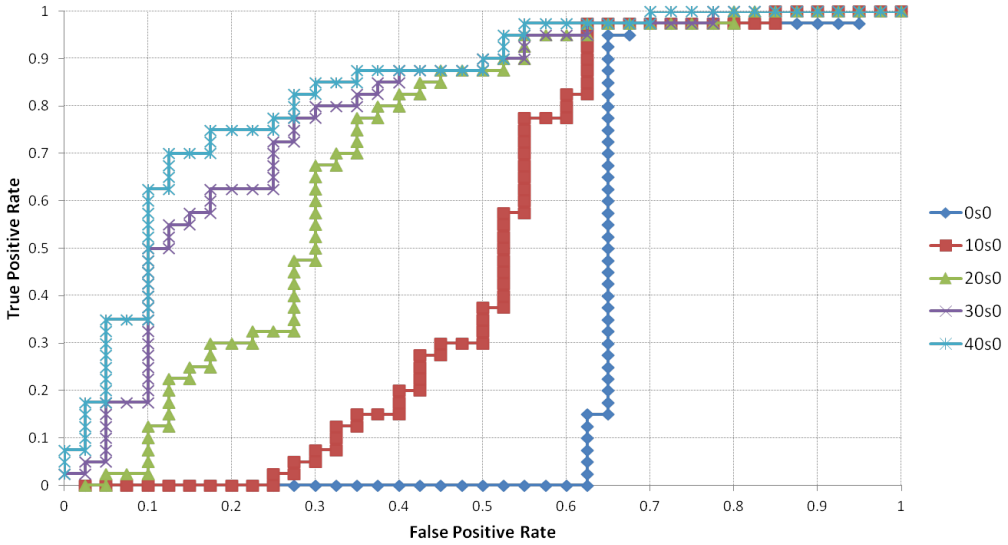

Figure C.26: ROC for the HMM detector, trained on viruses morphed with 10% subroutine and 10% dead code, and tested on virus variants morphed with dead code.

C.4 Training Set: Viruses Morphed with 10% Dead Code and 10% Subroutine Code

The training set consists of virus files morphed with 10% dead code as well as 10% subroutine code from normal files. The ROC curves for the HMM detector is illustrated in Figure C.26 and Figure C.27. The following two figures in Figure C.29 and Figure C.28 illustrated the ROC curves for the CSD estimator.

The ROC curves show that the CSD estimator performed relatively well when compared to the HMM detector. In fact, the HMM detector performed extremely poor in this scenario with high false positive rate.

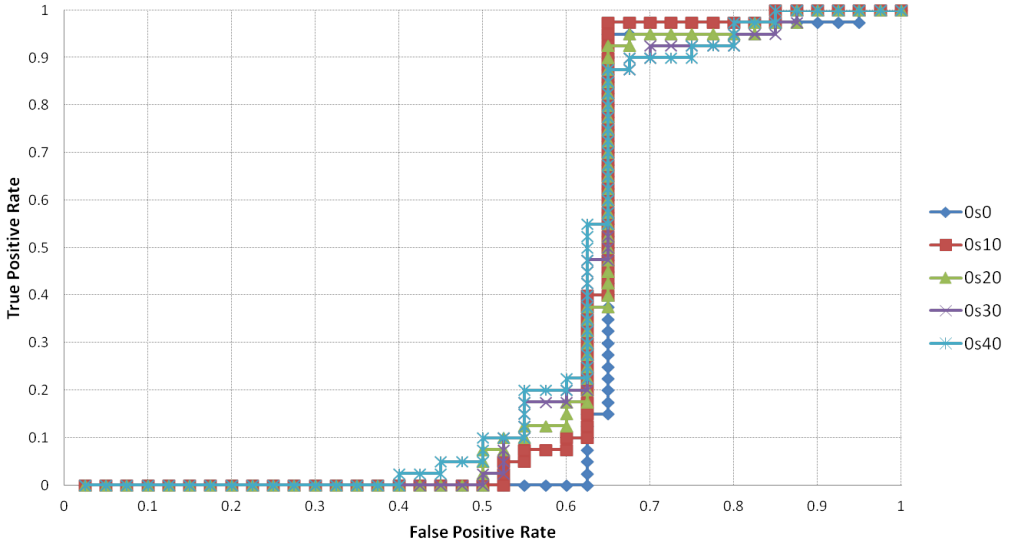

Figure C.27: ROC for the HMM detector, trained on viruses morphed with 10% subroutine and 10% dead code, and tested on virus variants morphed with subroutine code.

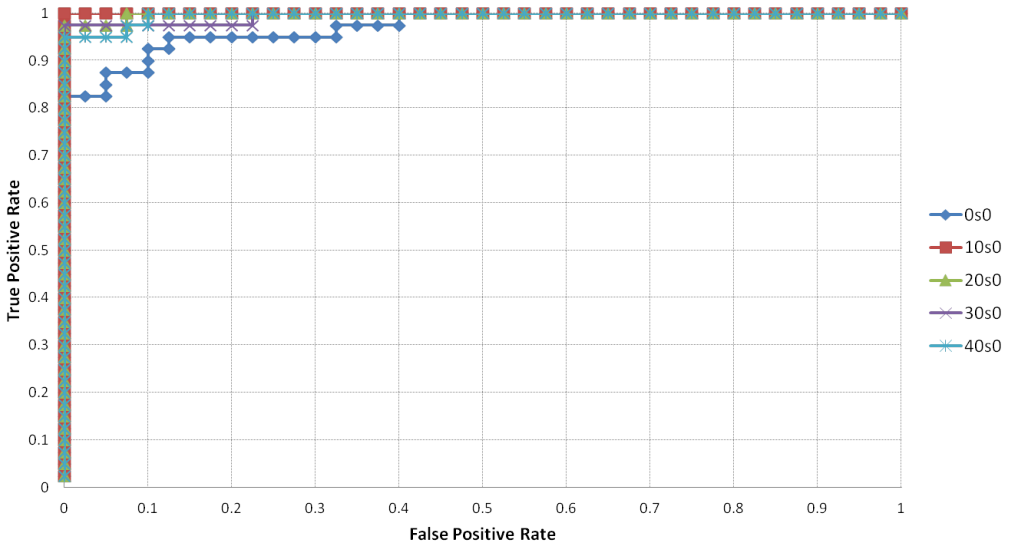

Figure C.28: ROC for the CSD detector, trained on viruses morphed with 10% subroutine and 10% dead code, tested on virus variants morphed with dead code.

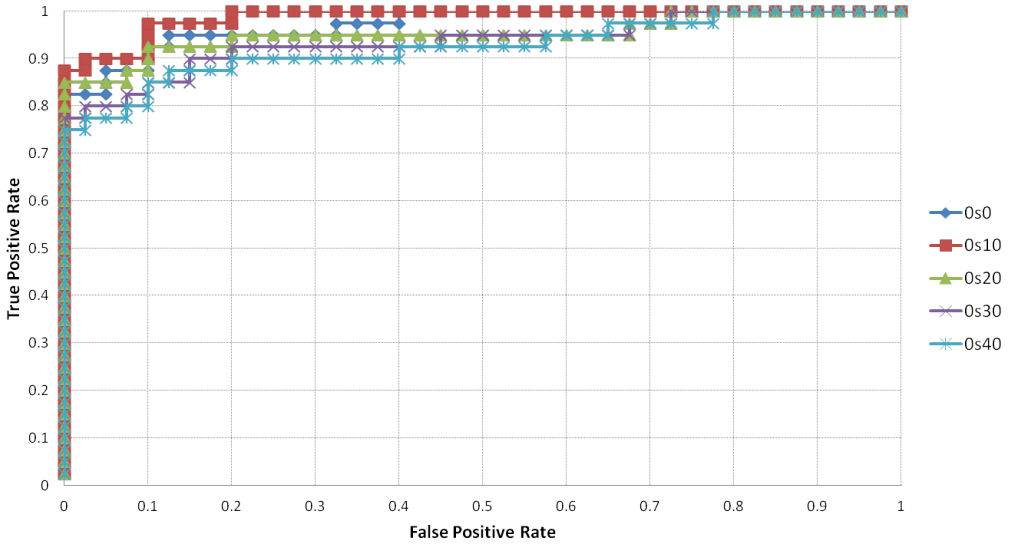

Figure C.29: ROC for the CSD detector, trained on viruses morphed with 10% subroutine and 10% dead code, and tested on virus variants morphed with subroutine code.

C.5 Training Set: Normal

The training set consists of normal files. The ROC curves for the HMM detector is illustrated in Figure C.30 and Figure C.31. The following two figures in Figure C.32 and Figure C.33 illustrated the ROC curves for the CSD estimator.

The ROC curves show that the HMM detector performed extremely well in this case. It achieved score closed to 100% true positive rate with 0% false positive rate.

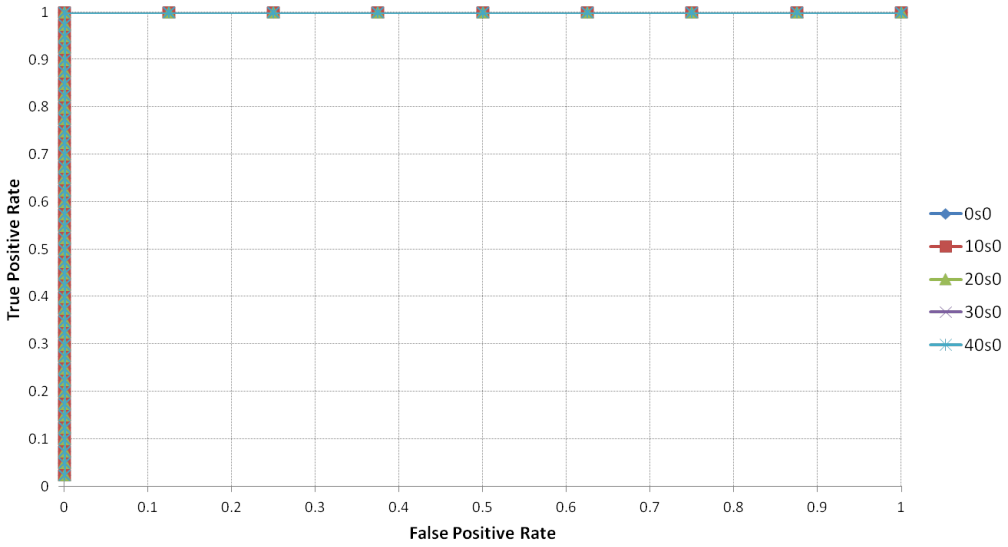

Figure C.30: ROC for the HMM detector, trained on normal files, and tested on virus variants morphed with dead code.

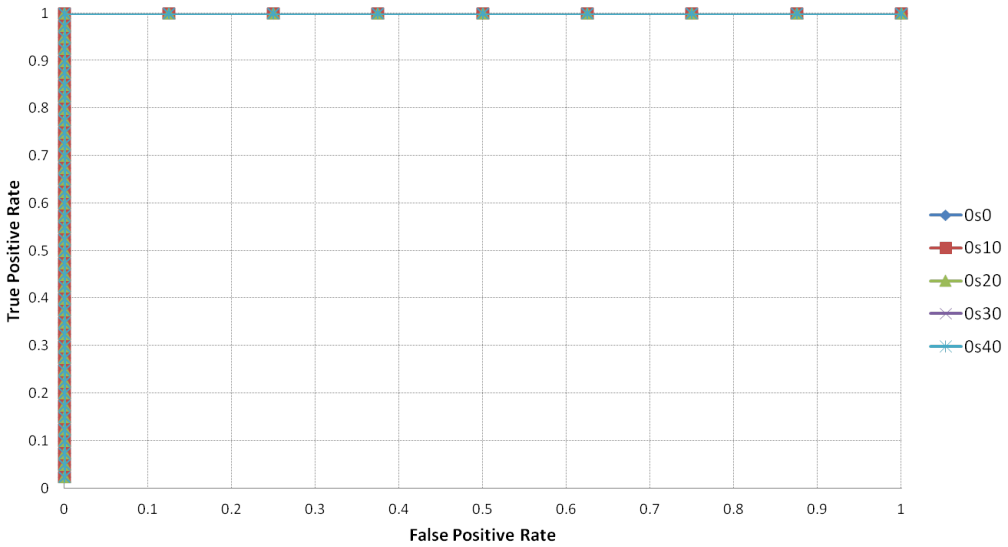

Figure C.31: ROC for the HMM detector, trained on normal files, and tested on virus variants morphed with subroutine code.

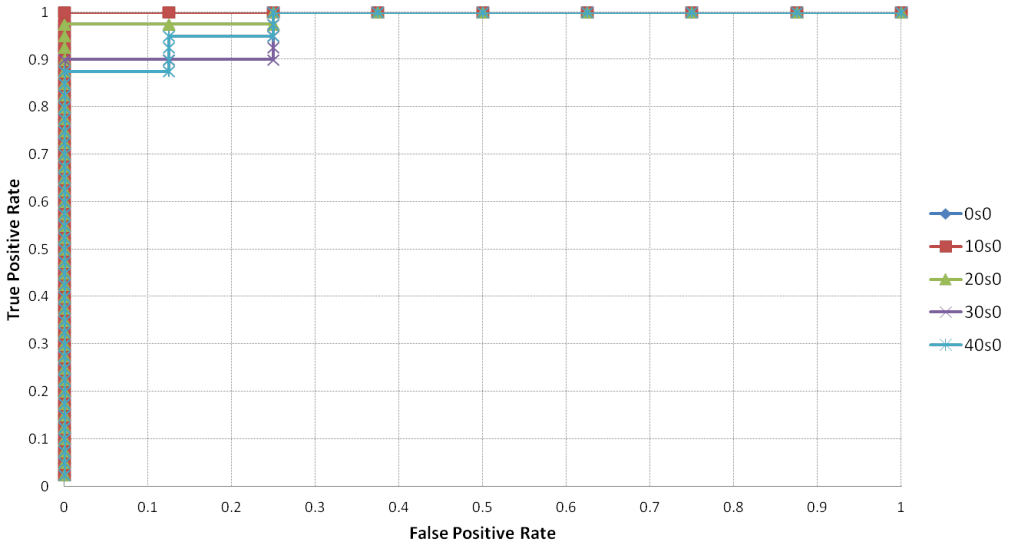

Figure C.32: ROC for the CSD detector, trained on normal files, and tested on virus variants morphed with dead code.

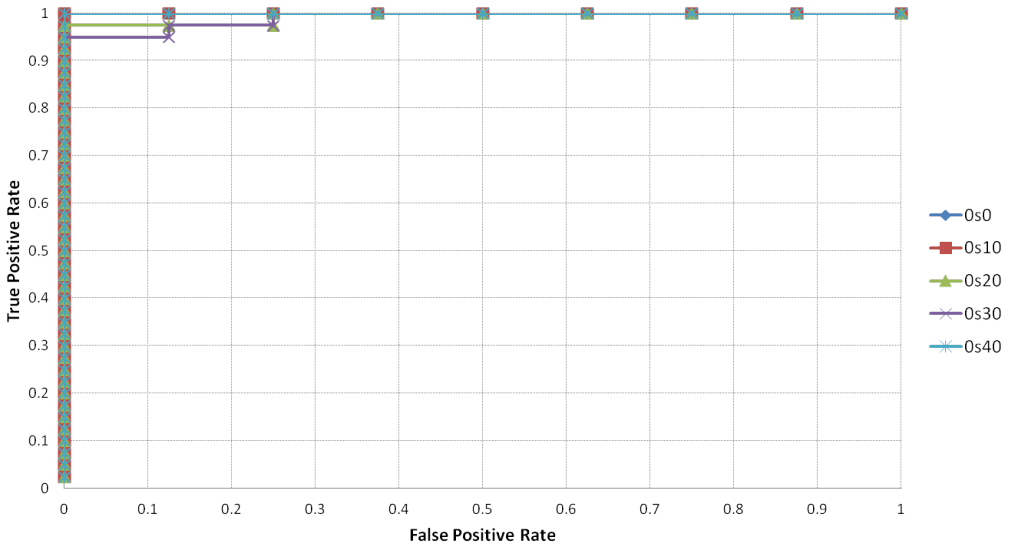

Figure C.33: ROC for the CSD detector, trained on normal files, and tested on virus variants morphed with subroutine code.

APPENDIX D

Analysis of Unigram and Bigram Features

We tested three different types of features used by the CSD estimator. When training the unigram model, we simply used instructions as features. The alternative was to use bigrams. Bigrams are groups of two instructions instead of a single instruction. When using a bigram model, there are two possible choices in terms of how to group consecutive instructions. The first is to simply use the two consecutive instructions in the order they appeared in the file. The other is to always make sure that the first instruction in the bigram comes alphabetically before the second (i.e., sorted). We refer to the first approach as "unordered bigrams" while the second is referred to as "ordered bigrams."

We demonstrate the concepts described above with the following instruction sequence: `MOV`, `AND` (listed in the order in which they appear in the file). The unigrams, in this case are simply `mov` and `and`. The ordered bigram is `and_mov`, while the unordered bigram is `mov_and`.

In our four experiments, we observed that the unigram model outperformed the bigrams in three cases. Therefore, in our hybrid virus detector, we chose to only use the unigram model.

D.1 Training on Unmorphed Virus Files

The unigrams performed better overall when compared to bigrams.

Table D.15: MMA for unigrams and bigrams. Training done on unmorphed viruses.

Dead Code	Subroutine	Unigrams	Ordered Bigrams	Unordered Bigrams
0%	0%	**99.50%**	97.00%	88.00%
	10%	85.75%	82.75%	**89.25%**
	20%	81.50%	79.50%	**90.50%**
	30%	80.25%	76.00%	**89.75%**
	40%	76.75%	72.25%	**89.50%**
10%	0%	**90.75%**	86.00%	78.00%
	10%	**83.75%**	80.75%	79.00%
	20%	**79.50%**	78.00%	78.50%
	30%	76.75%	75.75%	**78.25%**
	40%	74.75%	73.50%	**79.75%**
20%	0%	**86.25%**	83.00%	68.50%
	10%	**82.50%**	79.50%	69.50%
	20%	**78.75%**	76.75%	72.00%
	30%	**75.75%**	73.50%	72.50%
	40%	**74.25%**	73.50%	73.25%
30%	0%	81.75%	**82.00%**	61.00%
	10%	79.00%	**79.50%**	64.00%
	20%	76.25%	**76.50%**	65.50%
	30%	74.00%	**75.00%**	67.75%
	40%	73.25%	**74.50%**	67.75%
40%	0%	78.00%	**80.75%**	55.75%
	10%	76.00%	**78.00%**	58.75%
	20%	74.00%	**76.00%**	59.50%
	30%	73.25%	**74.50%**	61.50%
	40%	71.50%	**73.50%**	63.25%
Average		**79.35%**	78.32%	72.84%

D.2 Training on Virus Files Morphed with 10% Dead Code

Out of the four experiments we performed, this is the only case when the unigram model did not outperform the bigram models. The unordered bigrams performed better than the unigram model by 0.56%.

Table D.16: MMA for unigrams and bigrams. Trained on morphed viruses with 10% dead code.

Dead Code	Subroutine	Unigrams	Ordered Bigrams	Unordered Bigrams
0%	0%	92.00%	**96.25%**	69.75%
	10%	80.75%	**83.25%**	71.00%
	20%	76.00%	**79.75%**	72.75%
	30%	73.50%	**79.25%**	73.25%
	40%	71.50%	**74.75%**	73.50%
10%	0%	**95.75%**	95.50%	85.25%
	10%	86.75%	**88.25%**	85.00%
	20%	80.00%	83.00%	**84.00%**
	30%	77.25%	81.00%	**83.00%**
	40%	75.25%	78.50%	**81.75%**
20%	0%	**94.00%**	93.75%	89.25%
	10%	87.25%	87.00%	**87.50%**
	20%	81.75%	83.00%	**87.75%**
	30%	78.25%	80.25%	**87.00%**
	40%	76.00%	78.75%	**86.50%**
30%	0%	**93.00%**	91.75%	87.50%
	10%	**87.25%**	86.75%	86.75%
	20%	82.75%	83.00%	**85.75%**
	30%	78.00%	81.00%	**86.25%**
	40%	77.75%	80.75%	**85.00%**
40%	0%	**92.50%**	91.00%	87.50%
	10%	**88.00%**	85.00%	86.50%
	20%	82.75%	82.00%	**86.00%**
	30%	81.50%	81.25%	**86.50%**
	40%	78.50%	79.75%	**87.00%**
Average		82.72%	78.32%	**83.28%**

D.3 Training on Virus Files Morphed with 10% Subroutine Code

The unigram model performed better than the bigram models in every parameter combination.

Table D.17: MMA for unigrams and bigrams. Trained on morphed viruses with 10% subroutine code.

Dead Code	Subroutine	Unigrams	Ordered Bigrams	Unordered Bigrams
0%	0%	**96.25%**	78.50%	78.75%
	10%	**92.25%**	77.25%	75.25%
	20%	**90.00%**	74.50%	76.75%
	30%	**88.50%**	75.50%	75.25%
	40%	**86.75%**	73.00%	75.75%
10%	0%	**95.00%**	76.00%	68.00%
	10%	**92.50%**	75.00%	68.25%
	20%	**90.00%**	76.25%	67.50%
	30%	**88.75%**	73.50%	68.50%
	40%	**88.50%**	72.75%	67.50%
20%	0%	**92.00%**	75.75%	62.50%
	10%	**90.25%**	74.50%	61.50%
	20%	**89.50%**	74.75%	65.25%
	30%	**88.75%**	73.75%	66.25%
	40%	**88.25%**	73.00%	66.00%
30%	0%	**89.50%**	73.25%	56.50%
	10%	**89.75%**	74.00%	57.25%
	20%	**90.25%**	73.25%	58.50%
	30%	**88.50%**	74.25%	58.75%
	40%	**88.25%**	71.50%	59.50%
40%	0%	**87.50%**	72.25%	53.50%
	10%	**88.75%**	71.50%	55.00%
	20%	**88.75%**	73.00%	56.00%
	30%	**87.75%**	72.50%	55.25%
	40%	**87.25%**	72.50%	56.25%
Average		**89.74%**	74.08%	64.38%

D.4 Training on Virus Files Morphed with 10% Dead Code and 10% Subroutine Code

The unigram model performed better than the bigram models in most parameter combinations except in three cases.

Table D.18: MMA for unigrams and bigrams. Trained on morphed viruses with 10% dead code and 10% subroutine code.

Dead Code	Subroutine	Unigrams	Ordered Bigrams	Unordered Bigrams
0%	0%	**90.00%**	88.00%	88.50%
	10%	**90.25%**	81.25%	89.25%
	20%	87.75%	80.50%	**88.00%**
	30%	86.25%	79.50%	**86.75%**
	40%	83.50%	79.25%	**86.25%**
10%	0%	**97.75%**	83.50%	89.75%
	10%	**95.50%**	80.75%	88.00%
	20%	**92.75%**	77.75%	87.50%
	30%	**91.00%**	75.50%	85.75%
	40%	**88.50%**	75.25%	83.00%
20%	0%	**96.25%**	75.25%	83.00%
	10%	**94.00%**	72.50%	81.75%
	20%	**92.25%**	71.25%	81.75%
	30%	**90.75%**	69.25%	82.00%
	40%	**89.50%**	68.00%	80.25%
30%	0%	**95.00%**	65.00%	74.25%
	10%	**93.00%**	65.50%	75.25%
	20%	**92.75%**	66.50%	76.25%
	30%	**89.75%**	63.00%	76.00%
	40%	**89.75%**	63.25%	77.75%
40%	0%	**93.75%**	60.25%	69.75%
	10%	**93.50%**	60.50%	70.50%
	20%	**91.00%**	61.25%	73.25%
	30%	**91.50%**	61.50%	72.50%
	40%	**90.25%**	61.25%	74.50%
Average		**91.45%**	74.08%	80.86%