Spring 5-26-2015

Support Vector Machines and Metamorphic Malware Detection

Tanuvir Singh
San Jose State University

Support Vector Machines and Metamorphic Malware Detection

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Tanuvir Singh

May 2015

The Designated Project Committee Approves the Project Titled

Support Vector Machines and Metamorphic Malware Detection

by

Tanuvir Singh

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

ABSTRACT

Support Vector Machines and Metamorphic Malware Detection

by Tanuvir Singh

Metamorphic malware changes its internal structure with each infection, which makes it challenging to detect. In this research, we test several scoring techniques that have shown promise in metamorphic detection. We then perform a careful robustness analysis by employing morphing strategies that cause each score to fail. Finally, we show that combining scores using a Support Vector Machine (SVM) yields results that are significantly more robust than we obtained using any of the individual scores.

ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Mark Stamp for his continuous guidance and support throughout this project and believing in me. Also, I would like to thank the committee members Dr. Jon Pearce and Mr. Fabio Di Troia for monitoring the progress of the project and their valuable time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Malware also known as Malicious software is a software program which intends to perform malicious activities on a computer. It can be used to steal sensitive information or to gain un-authorized access to private networks [16]. Different malware are created for different purposes. A malware that changes its internal structure each time it infects a new system is referred to as a metamorphic malware. Once a system has been infected, an antivirus software needs to identify the infection and take steps to eradicate it. A lot of malware are created by developers these days to generate money. In this paper we use the terms virus and malware interchangeably.

As the quantity of new and unknown malware is on the rise by the day, analysis of different kind of malware and their detection has become a prime research area. Malware detection strategies need to be updated regularly so as to cope up with increasing diversity in metamorphic malware. Various techniques have been proposed in the past for malware detection. Initial techniques which were based on virus signature cannot be used for detection of metamorphic malware. In recent years many new machine learning techniques have been proposed for detection of metamorphic malware. Statistical malware detection techniques are based on some statistical characteristics of a malware file, an example being Hidden Markov Models (HMM) [26]. Similarity based techniques try to establish a similarity measure between files of the same family, Simple Substitution Distance is an interesting similarity based technique discussed in paper [24]. Examples of graph-based techniques include Opcode Graph Similarity discussed in [23] and Function Call Graph technique discussed in [8]. Entropy analysis, compression rates, and Principal Component Analysis (PCA) are examples of Structural-based techniques. In our research we will be implementing a subset of these scoring techniques and then will try to devise some kind of morphing strategies to break each of these scores. Our main aim here is to use minimum amount of morphing which is sufficient to produce some kind of misclassification. Then we try to devise a combined morphing strategy that can break all of our scores with minimum amount of morphing. Finally, we will implement a Support Vector Machine (SVM) [18] that will serve to generate an optimal combination of scores, and we will measure the success of this SVM-based score in comparison to the individual scores.

This paper is organized as follows. In Chapter 2, we discuss about available background information on malware with an emphasis on metamorphic malware. Chapter 3 outlines previously implemented statistical and similarity based metamorphic malware detection techniques that form the basis for the research in this paper. Then in Chapter 4, we cover the Support Vector Machines and discuss our specific implementation using Rapidminer. Next, we look at three different malware detection scores such as Hidden Markov Model, Simple Substitution distance and Opcode Graph Similarity techniques. Then we consider a combination of the these three scores using Support Vector Machines. These topics are covered in Chapter 5. Then in Chapter 6 we apply all of our metamorphic detection strategies to some new classes of malware and then compare the results produced by SVMs against the results produced by other three techniques. Chapter 7 contains our conclusions and considerations for future work.

CHAPTER 2

Background

A software program developed with an intention to harm another computer or software is known as malware. Malware are developed with different goals in mind. Usage varies from stealing information, damaging files on a computer to earning money as a developer. malware are hidden and distributed through a variety of channels such as seemingly legit online software download s, CDs installs, email attachments etc. Hence, firewalls and antivirus softwares are needed to be run frequently for malware detection and removal [19].

2.1 Types of Malware

Malware is a generic term which covers a wide range of malicious softwares. The term malware can refer to any of the following:

2.1.1 Trojans

Term Trojan horse is derived from the deceptive horse used in the ancient war of troy. It refers to a software program that presents itself as a legit software but means harm to the host machine. It lures user into clicking on unauthentic links, email attachments, downloading seemingly genuine files, etc., to penetrate the host system and get executed without user's knowledge. Trojan horse can be used to monitor user activity, get remote machine or web cam access, execute other harmful softwares or to steal personal information from the host system. However, trojans do not replicate or spread themselves thereby differ from viruses and worms [3].

2.1.2 Worms

Worm is a malicious software program that can spread itself without any user intervention and hence is self sufficient. It generally uses host machine's network to jump and infect other machines on the network as the infected document travels from one machine to another. Worm can be as small as a macro in a word document. Worms can collapse an entire network just by using up the bandwidth and not performing any malicious activity [3]. Developers also use it to create backdoors and bot nets.

2.1.3 Viruses

Viruses get executed at whatever point the infected frameworks are booted. Once executed virus can spread itself by multiplying its copies and infecting other files and programs [3]. However, viruses are a lot different from worms as human action is required in order to spread viruses. One of the most typical scenario is copying data using an infected thumb drive on different machines thereby spreading viruses on each of the host machines.

2.1.3.1 Encrypted viruses

To evade antivirus softwares malware needs to change its body with each infection. The easiest technique that can be used for this is by encrypting the virus body. Encrypted viruses generally use simple encryption techniques such as computing XOR of a key with each byte of the virus body. Such viruses also have a decryption block of code along with the encrypted body. However, decryption code remains same across all variations of the virus thereby making it vulnerable to detection. Hence, even though the scanner component of antivirus software cannot decrypt the virus body, a signature based detection is highly probable [15]. The first such virus was developed in 1987 called Cascade [22].

2.1.3.2 Polymorphic viruses

In addition to the rather simple encrypted viruses, polymorphic viruses have an additional component called mutation engine. Mutation engine plays a crucial role in efficiently changing the body of the virus. Encypted virus body and the mutation engine is first decrypted by decryption module once the virus is downloaded on the host machine. Every infection then generates a new virus body and decryptor using the encrypted mutation engine [22]. Anti-virus softwares can detect polymorphic viruses by using the method of heuristic analysis in sandbox mode. The first such virus is 1260 developed in 1990 [29].

2.1.3.3 Metamorphic viruses

Metamorphic viruses are the most advanced form of viruses. These are different from polymorphic viruses in a way that they do not have a decryptor or encrypted virus body. Instead “Metamorphic viruses spread by keeping the base functionality same but by changing their whole body i.e. by rewriting themselves”. Hence, generations differ only in the code of the virus [15].

Metamorphic viruses use morphing techniques to change their code before each infection. Few elementary morphing techniques are discussed below.

Subroutines Permutation It is important to change the structure of the virus code in order to change its appearance. We can achieve this by inserting random methods between two functional methods in the code. Thus, antivirus softwares based on structural similarity are not be able to detect such viruses easily. One such virus is win32/Ghost [34].

Instruction Reordering Instruction reordering works by identifying independent instructions within each functional module of the virus code. Such instructions execute independently with respect to other instructions. Reordering these leads to different appearance but same functionality. Due to its simplicity this technique can be easily used to generate different generations of metamorphic viruses [22].

Instruction Substitution This technique uses functional equivalents to replace one or more instructions in the virus code [32].

Register Swapping Another simple approach for keeping the generations different is to just change the registers used by different instructions in the code. A simple swap of registers each time meaning using them alternatively is one of the most trivial examples. Win95/Regs wap [34] is one of the many metamorphic malware which makes use of this morphing technique.

Garbage Instruction Insertion Garbage instruction insertion technique uses "do-nothing" instructions for code obfuscation. Such statements primarily do not have any effect on the code functionally and act as dummy code lines once inserted [7]. Win95/ZPerm [29] is the most common example.

Malware writers have even created metamorphic "engines", which are now available publicly and can be used to generate metamorphic copies of a given malware. These engines are referred to as “Virus Construction Kits” and can be used to create functionally equivalent copies of an executable, and hence ease the process of creating different generations of existing malware. Few examples of such kits are Phalcon/Skism, NGVCK [25], Mass-Produced Code generator (PS-MPC), MPCGEN and Second Generation virus generator (G2) [31]. We focus on NGVCK in sections ahead because it was found to be one of the best engines capable of producing entirely different metamorphic copies in previous researches [25].

2.2 Detection Techniques

With the rapid increase in types and intricacy of obfuscation techniques utilized by malware writers, there is great requirement for antivirus softwares to keep up with the pace. Hence, there lies a vast scope for research and possibilities in detection techniques. Currently multiple techniques are used for malware detection which focus on varied parameters. Some of these techniques are discussed below in detail.

2.2.1 Signature Based Detection

Signature based detection is based on scanning programs and files and computing signatures for each of them. Signature based detection involves searching for a known pattern, referred to as signature, in a given executable file. Most of the antivirus manufacturers maintain a large repository of unique signature for each known virus which is updated on a regular basis [3]. When an anti-virus software scans an executable, it generates the signature of the executable file and looks for a match in its database, if a match is found the executable is deemed to be infected. Signature based detection scheme is used on a vast scale because it is simple, accurate and fast [26]. One of the drawbacks of this technique is that it requires a continuous update of signatures for newly found malware. Also it cannot be used to detect previously unknown malware as it can only detect malware with a known signature. Another of its drawbacks is that it is very easy to evade signature based detection, simple obfuscation techniques like polymorphism and metamorphism can be used to evade signature based detection [21].

2.2.2 Behavior Based Detection

Behavior based detection is another legacy technique wherein the focus is on the actions that a malware performs during its execution and trying to understand the intent of the malware using various techniques. In behavior based detection, the behavior of both benign files and malware files are studied during the first phase generally referred to as the training or learning phase and then during the monitoring phase, we use the information gathered in the training phase to classify a given executable as either malware or benign [12].

2.2.3 Anomaly Based Detection

It is a technique which is inherently similar to behavior based detection. It can actually be seen as a slight variation of behavior based detection, wherein analysis is done by studying the behavior of all files in the training phase, and during the monitoring phase we look out for files which show deviation from normal behavior, such files are classified as infected. However, this technique is more susceptible to false positives. In [30] an unsupervised approach for "Anomaly-based Malware Detection using Hardware Features" has been discussed.

2.2.4 Statistical Malware Detection

Conventional approaches like signature detection cannot be used in the case of metamorphic malware as they evade signature detection by morphing their code. Although metamorphic malware copies can differ from each other a great deal, but still some of the statistics of the metamorphic malware files remains the same. A variety of other techniques, including machine learning [33] and statistical analysis [10] have been studied. In addition, some improved techniques for evading these metamorphic detection schemes have been considered in [15].

2.2.4.1 Hidden Markov Model Based Detection

In recent years a lot of machine learning techniques have been used in the detection of malware, specifically metamorphic malware which cannot be detected by using traditional malware detection techniques. Most of the machine learning techniques work on the principle of analyzing a particular family of virus for some kind of similarity score, which can then be used to detect an incoming file as either a malware belonging to the same family or as a benign file. One of such technique is Hidden Markov Model (HMM), which is one of the most popular machine learning techniques used in the field of malware detection [26]. In this technique, a Hidden Markov Model is trained against known malware opcode sequence [17]. Once the training phase is over, the trained model is used to score incoming files. The score is then compared to a predefined threshold, if it is more than the threshold, the file is classified as an infected file [33]. We will discuss more about this technique in Section 3.1.

2.2.5 Similarity Based Detection

This kind of detection is based on some kind of a similarity measure defined between the metamorphic and benign files. It revolves around finding some kind of characteristics which are similar for a given metamorphic malware family. In previous researches a lot of similarity based detection strategies have been discussed which are all based on analyzing characteristics of opcode sequences of malware files. Some of these techniques are pairwise sequence alignment [1, 20], n-gram similarity [33], cosine similarity [14], and chi-squared similarity [10].

2.2.5.1 Opcode Graph Based Detection

Opcode graph based techniques are the techniques which involve analyzing the graphs generated by opcode sequences instead of analyzing the files themselves. For detection a weighted directed graph is constructed by analyzing a metamorphic malware family. Then a graph is constructed for the file under consideration. Finally the two graphs can be compared to generate a score. The technique works by calculating the absolute difference between corresponding elements in the two graphs. If the computed score is low it means that the two files are similar and hence the file is classified as a malware file otherwise it is classified as a benign file [23]. We will discuss more about this technique in Section 3.2.

2.2.5.2 Simple Substitution Based Detection

Simple substitution is another detection technique which is based on a similarity measure. This technique uses approach which is very similar to the use of simple substitution ciphers in cryptanalysis. It uses Jackobsen's fast algorithm similar to its use in cryptanalysis. Then we try to establish a kind of a similarity score by trying to convert the matrix formed by the opcode sequences of a given malware to that of the fuel under consideration [24]. We will discuss more about this technique in Section 3.3.

CHAPTER 3
Statistical and Similarity based Malware Detection

Signature based detection is one of the most common method used by antivirus softwares for malware detection. But it cannot be used to detect metamorphic malware, as metamorphic malware evade signature based detection by morphing its body in such a way that the internal structure of the morphed malware is completely different from its original copy. Even though the internal structure of the malware has been completely changed by morphing, but still, the instructions that this new morphed malware executes have to be the same in order to perform the same actions. This means that in one way or the other the distribution of these instructions will be the same across all morphed copies. Based on these assumptions various malware detection techniques have been devised, we discuss some of these techniques in the coming sections.

3.1 Hidden Markov Model Method

Hidden Markov Model also referred to as HMM in coming sections is a technique which can be used to recognize patterns. We can use a slight modification of this technique to detect metamorphic malware. A Hidden Markov Model can be trained based on a representative set of malware files and then can be used to detect if given file belongs to a particular malware family or not. We use log likelihood per opcode(LLPO) as a measure to score these files. Once we have trained a Hidden Markov model we can then use it to score new files, these new scores can then be compared to a predefined value also known as threshold. If the score generated by a file is greater then the threshold value then it can be inferred that the given file belongs to the same metamorphic malware family.

3.1.1 Hidden Markov Model

Hidden Markov Model(HMM) Markov Process whose states are unknown [26]. The notation used in HMM is described in [26] as follows:

T	$=$	length of the observation sequence
N	$=$	number of states in the model
M	$=$	number of observation symbols
Q	$=$	$\{q_0, q_1, \ldots, q_{N-1}\}$ = distinct states of the Markov process
V	$=$	$\{0, 1, \ldots, M-1\}$ = set of possible observations
A	$=$	state transition probabilities
B	$=$	observation probability matrix
π	$=$	initial state distribution
$\mathcal{O}$	$=$	$(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$ = observation sequence.

A model is defined using these known values as

$$\lambda = (A, B, \pi)$$

Figure 1 gives a graphical view of the Hidden Markov Process. The state and observation of HMM at any point of time t is represented by X_t and $\mathcal{O}_t$ respectively, as shown in Figure 1.

3.1.2 Implementation

In our implementation we start of by training a Hidden Markov Model based on set of given metamorphic malware files. Once the model has been trained this model represents the statistical properties that define the given malware family in some way. We can then use this resulting model to score

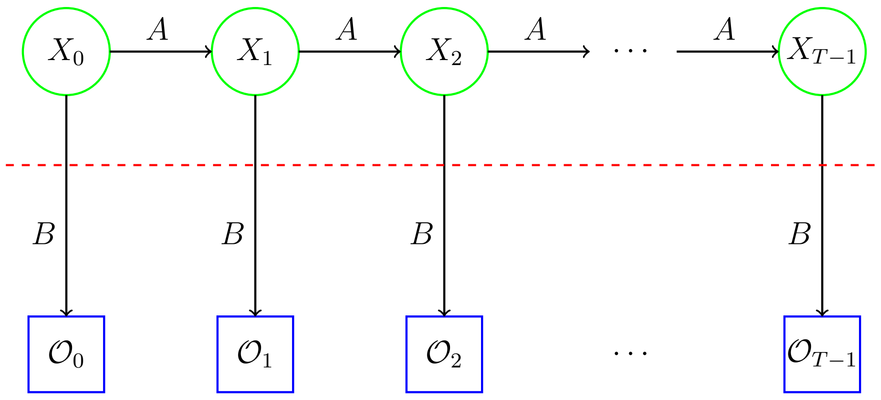


Figure 1: Hidden Markov Model

a file under consideration and then predict whether it belongs to the same family that the model was trained upon or not. In the training phase we extracted sequence of opcodes from given executable files by disassembling these files into machine level code. The we created an observation sequence out of these opcode sequence by combining all of them into a long string of opcode sequences.

Once we have trained our HMM on this given observation sequence we can then use this resulting model to score a file under consideration, and then predict whether it belongs to the same family or not. We would have two sets of datasets, the test set and the compare set. The test set comprises of the malware family under our consideration and the compare set usually comprises of benign files. Then we compare the scores generated by the files in test set to those generated by files in the compare set. Our main aim here is to get a clear separation between the scores generated by each of these data sets. In terms of LLPO our model should assign a higher LLPO value tot he files that belong the the malware family under consideration and assign a lower score to the comparison data set i.e. the benign files.

After training a HMM on NGVCK virus files we use it to score a subset of NGVCK malware files and also an equivalent number of benign files(Cygwin files). Then we compare the scores generated by the test data set comprising of the NGVCK files against those generated by the comparison data set comprising of the benign files.

3.2 Opcode Graph Similarity

The given malware file is first disassembled to generate a sequence of opcodes. This sequence of opcodes is then used to generate a directed graph. To construct this directed graph, first of all we create nodes representing all the unique opcodes that we encounter in the given malware file. Once we have all the edges for this graph we insert edges between each pair of nodes which occur as consecutive opcodes in the given file i.e. a directed edge is constructed for each pair of opcodes opc1 and opc2 such that opc2 follows opc1 in the sequence of opcodes for that particular file. The weight of that edge is defined as the probability of these two opcodes in that particular order.

3.2.1 Implementation

Here our main aim is to construct an opcode graph for each file and then compare opcode graphs from two files to generate a similarity score between those two files. For this we go through the sequence of opcodes in a given file and create an opcode graph. We create a weighted directed graph wherein each node consists of a unique opcode in our file and the edges connecting different opcodes are created for each opcode bigram in that file. We create this graph for both the malware files as well as the benign files. For comparison we can then use these opcode graphs instead of the actual files. Here we discuss the approach that has been previously studied in the paper [23].

Table 1 shows a subset of assembly language trace from a given metamorphic file:

Table 1: Opcode Sequence

Number	Opcode	Number	Opcode
1	`CALL`	11	`JMP`
2	`JMP`	12	`ADD`
3	`ADD`	13	`NOP`
4	`SUB`	14	`JMP`
5	`NOP`	15	`CALL`
6	`CALL`	16	`CALL`
7	`ADD`	17	`CALL`
8	`JMP`	18	`ADD`
9	`JMP`	19	`JMP`
10	`SUB`	20	`SUB`

Table 2 shows a matrix containing count for each instruction digram in the given file. There is a count value against each row-column opcode pair if they exist in the opcode sequence one after the other and the value is determined as the sum of all such occurences. So, the value against row 1 col 3 corresponds to the count of instruction digrams involving `ADD` followed by `JMP`. Then, we convert this table into a row-stochastic table by dividing each count by the number of columns in the table. This leads to a matrix which provides us with transition probabilities between successive opcodes as shown in Table 3. Then, we generate a weighted directed graph for this table as shown in figure 2.

Table 2: Opcode Count

	ADD	CALL	JMP	NOP	SUB
ADD	0	0	2	1	1
CALL	2	2	1	0	0
JMP	2	1	1	0	2
NOP	0	1	1	0	0
SUB	0	0	1	1	0

Table 3: Probability Table

	ADD	CALL	JMP	NOP	SUB
ADD	0	0	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$
CALL	$\frac{2}{5}$	$\frac{2}{5}$	$\frac{1}{5}$	0	0
JMP	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	0	$\frac{1}{3}$
NOP	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0
SUB	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0

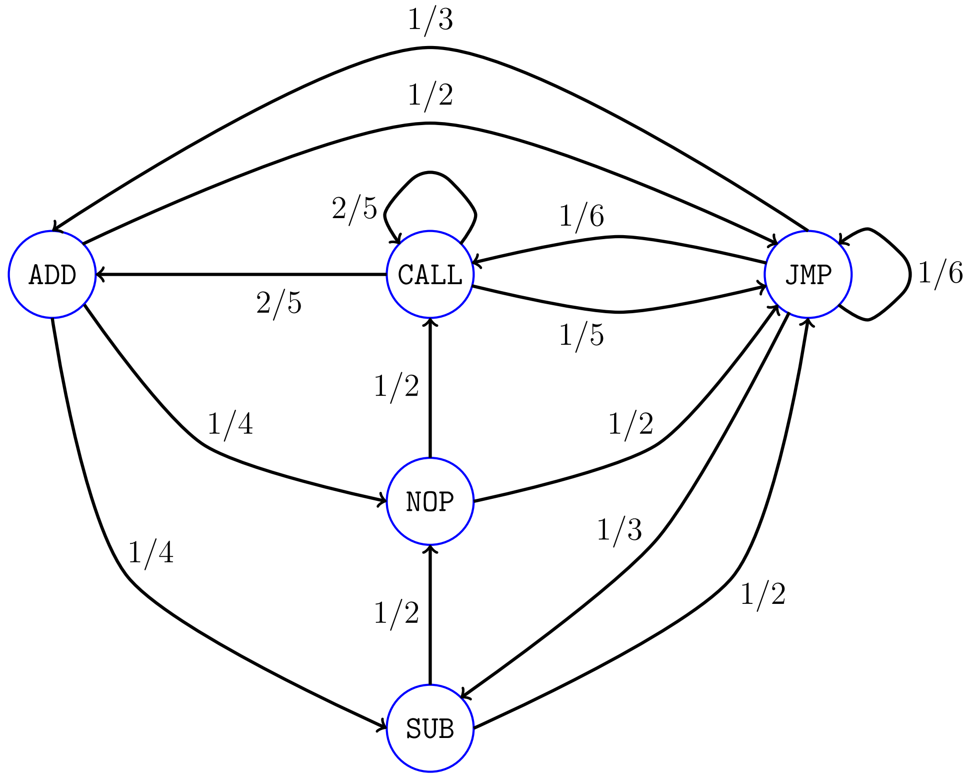


Figure 2: Opcode Graph

Once we have the graph, we need to establish a threshold value which can then be used to compare different scores by following the below steps:

1. Determine the opcode graphs for a set of malware family.
2. Determine the opcode graphs for a set of benign files.
3. Compute the scores for all pairs of metamorphic family viruses from step 1.
4. Determine the scores for differing pairs comprising of one family malware from step 1 and one benign file from step 2.
5. Establish threshold values based on results from step 3 and step 4.

Once a threshold score has been established, we can use this score to compare different sets of files. Now, to check if a given file belongs to a particular malware family or not, we would first generate an opcode graph for the given file. Then, we can compare this graph with that of a file from the malware family in consideration. Once we have this score we can compare it to our threshold value, if it is greater then the threshold value then we classify the given file as a benign file otherwise as a malware file belonging to the same family.

3.3 Simple Substitution Distance

It is an efficient technique which uses the similarity measure between two given files for detection of metamorphic malware. This technique again relies on the fact that we have a representative set of malware files belonging to the same family. Here, we use a distance measure similar to the method used in cryptanalysis, which is based on simple substitution distance. This method uses a hill climbing approach based on Jakobsen's algorithm [13]. The main idea behind the approach is based on the frequency of opcodes, assuming that they stay the same for a given family. So, we can use Jakobsen's algorithm to compare the similarity of a given file against both a malware file and a benign file.

3.3.1 Jakobsen Algorithm

Jakobsen's algorithm is an algorithm which involves refining an initial guess for the encryption key with each iteration. It is based on the assumption that the cipher text is actually in English language and has only 26 different symbols. So, each of these symbols represents a letter in the english language. it is a very fast algorithm because the distribution matrix is only created once and is then changed after every iteration to evaluate the plain text.

The algorithm starts of by calculating the frequency of all the symbols in the cipher text. Once all the frequencies have been calculated we can sort them in reverse order. Then we can compare the maximum frequencies from the cipher text to english language character frequencies and generate a putative key. The algorithm then runs various iterations to improve on this key as follows:

3.3.2 Implementation

In our implementation we follow the same approach of extracting opcode sequences from all the family files as discussed in the previous two sections. Then we create bigram distribution matrix from these opcode sequences which

Algorithm 1 Jakobsen's Fast Attack on substitution ciphers

```
1: Initialize $E$ with expected digram frequencies
2: $C$ = Input cipher text
3: $K$ = Compute Initial Putative Key
4: $P$ = Putative plaintext by decrypting C using K
5: $D$ = Digram distribution matrix for P
6: $score = d(D, E)$
7: for $i = 1$ to $n - 1$ do
8:   for $j = 1$ to $n - i$ do
9:     $D' = D$
10:    swapRows(j, j+i)
11:    swapColumns(j, j+i)
12:    if d(D′,E) < score then
13:      $D = D'$
14:      swapElements(j, j+i) {Swap elements of the putative key}
15:      score = d(D',E)
16:    end if
17:  end for
18: end for
19: return K
```

is the equivalent of E matrix in Jackobsen's algorithm [13]. Then we construct another similar matrix for the file which we want to classify, this matrix is equivalent of D matrix in the algorithm. We constrain both of these matrices to the most common n opcodes. We add one more symbol to represent all the other opcodes which were excluded from the list. So, we are left with $n + 1$ symbols.

Then we choose an initial "key" K such that it is a representative of the frequency of opcodes in the malware family. We assume that the frequencies of opcodes in family viruses should be the same as compared to the code that we are suspecting.

Then we construct the equivalent of matrix D by using the key K to decrypt. This is similar to the procedure followed in Jackobsen's algorithm to construct D. Once we have the E and D matrices we make them row stochastic so that the scores are not dependent on the size of the opcode sequence. For example, if we have the below mentioned opcodes:

$$\texttt{MOV}, \texttt{CALL}, \texttt{ADD}, \texttt{XOR}, \texttt{CMP}$$

where the opcodes are in a reverse sorted order of their frequencies. Assume that the sequence that we want to score is:

$$\texttt{JMP}, \texttt{MOV}, \texttt{MOV}, \texttt{ADD}, \texttt{INC}, \texttt{INC}, \texttt{INC}$$

the frequency counts for these will be:

Table 4: Opcode Frequency Counts

Opcode	INC	MOV	ADD	JMP
Frequency	3	2	1	1

So, our initial guess for the putative key K is:

Table 5: Initial Putative Key Guess

Metamorphic Family	MOV	CALL	ADD	XOR
File to be scored	INC	MOV	ADD	JMP

By using the above putative key K the decrypted sequence results in:

$$\texttt{XOR}, \texttt{CALL}, \texttt{CALL}, \texttt{ADD}, \texttt{MOV}, \texttt{MOV}, \texttt{MOV}$$

This gives us our initial D matrix which is also referred to as digraph distribution matrix as shown in table 6.

Once we have scored the D and E matrices against each other, the following step involves swapping the initial two opcodes in ‘putative key’ k, i.e.

Table 6: Digraph Distribution Matrix

	MOV	CALL	ADD	XOR	CMP	OTHER
MOV	2	0	0	0	0	0
CALL	0	1	1	0	0	0
ADD	1	0	0	0	0	0
XOR	0	1	0	0	0	0
CMP	0	0	0	0	0	0
OTHER	0	0	0	0	0	0

if we are looking at the first row then we will swap `MOV` and `CALL`, to achieve this we can just swap the first row with the second row and repeat the same with columns. This will result in our modified matrix shown in table 7.

Table 7: Modified Digraph Distribution Matrix

	MOV	CALL	ADD	XOR	CMP	OTHER
MOV	1	0	1	0	0	0
CALL	0	2	0	0	0	0
ADD	0	1	0	0	0	0
XOR	1	0	0	0	0	0
CMP	0	0	0	0	0	0
OTHER	0	0	0	0	0	0

From here onwards our algorithm carries on in the same way as in Jackobsen's algorithm [13]. The score for a given file is computed as the score of the final D matrix. In our implementation we limit the size of our matrices to 25, based on previous research done in paper [24].

CHAPTER 4
Support Vector Machines

4.1 Introduction

Support vector machines are an example of supervised learning algorithms which belong to both the regression and classification categories of machine learning algorithms. SVMs is a collection of machine learning algorithms that can be used to recognize pattens in given data [5]. Given a set of training data we would like to classify new examples into one of the possible two categories. For achieving such a task SVM training algorithm can be used to build a model which is capable of performing such classification. We can define the SVM model by having all the example from the training data represented as points on a space. These points will be represented in the two dimensional space such that there is an evident gap between data points belonging to our two different classes. So for new points we can use this model to classify it to one of these two classes based on which side of the gap it belongs to. This kind of classification is known as linear classification [28].

4.1.1 SVM Example

SVM's can be defined by using the concepts of decision planes and support vectors. The decision planes are planes in the two dimensional space that would represent the decision boundaries. Basically a decision plane is defined as a plane which completely separates all data points belonging to two different classes. Figure 3 shows an example of data points in two dimensional space. In this example we have two different types of data points represented by

Red and Green dots. The Green points refer to class A whereas Red points refer to class B. If we look at the figure we can clearly see that we can plot a line segment which can separate all the data points in a way that all points belonging to one class are on one side and all the other points are on the other side of the line segment. This line segment will be our decision plane. Any new examples of data coming in can be classified based on this line segment, if it falls on the left side then it belongs to class B otherwise it belongs to class A [28].

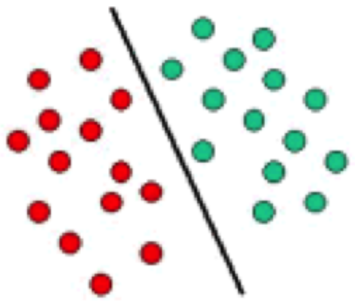


Figure 3: Linear Classifier

The type of classification shown in Figure 3 is referred to as linear classification, i.e. the classifier separates two different classes of data by finding a decision plane in between them.

But in some cases the classification problem is much tougher then the one shown in Figure 3. An example of a different kind of classification problem is shown in Figure 4. If we compare the two figures we can clearly see that the classification task in the second is tougher and we cannot have a simple line segment separating the two classes. In such situations we would have to plot a curve instead. Classification problems which involve plotting of simple line segments to differentiate between data points belonging to two different kinds are referred to as hyper plane classifiers [28]. These kind of problems are best suited for Support Vector Machines.

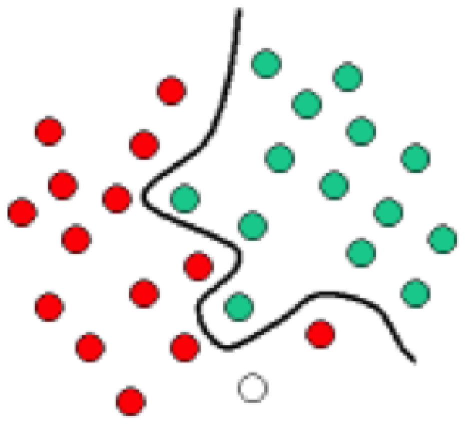


Figure 4: Non Linear Classifier

But sometimes there are cases wherein we cannot separate the datasets linearly. In such scenario's support vector machines allow us to use a special trick known as the kernel trick. A kernel trick is basically using some mathematical functions which are known as "kernels" to map the given data sets into a new feature space(mapping). The only condition here is that when we map our data points into this new feature space, the newly mapped points should be linearly separable in this new feature space. So, we can change the complex problem of plotting a curve into a simple problem of plotting a line segment in the new feature space. Figure 5.

4.1.2 Kernel Mapping

Definition: A kernel is defined as a function that accepts two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ as inputs and produces an output which is defined as the inner product of their images $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1)^T \phi(\mathbf{x}_2)$$

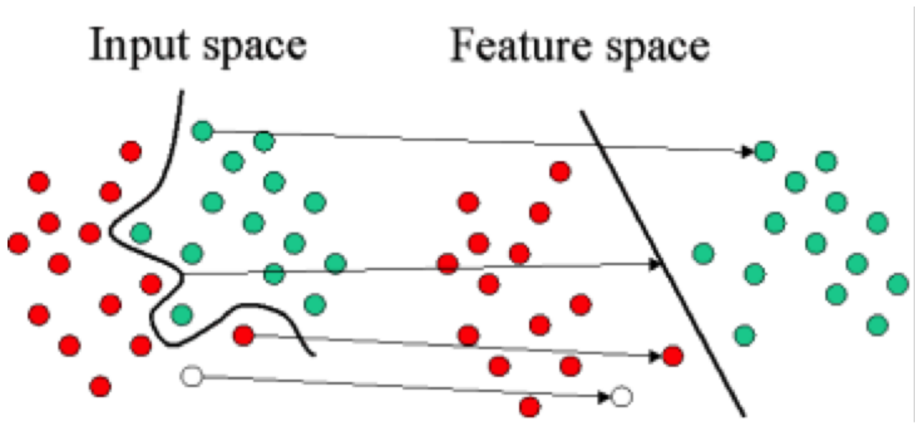


Figure 5: Kernel Mapping

We are not concerned about the dimensionality of the newly formed space because we are only returning the inner products in the new space for the two vectors.

The main idea here is to generate a learning algorithm that operates in kernel space, which is generated by substituting the values of all inner products from the original space into the newly formed kernel space:

$$f(\mathbf{x}) = \phi(\mathbf{x})^T\mathbf{w} + b = \sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}, \mathbf{x}_j) + b$$

The parameter b can be found from any support vectors $\mathbf{x}_i$

$$b = y_i - \phi(\mathbf{x}_i)^T\mathbf{w} = y_i - \sum_{j=1}^{m} \alpha_j y_j (\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)) = y_i - \sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

4.1.3 Linear separation of a feature space

Let's assume that we have a hyper plane in an n-dimensional (n-D) original feature space

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w} + b = \sum_{i=1}^{n} x_i w_i + b = 0$$

where $\mathbf{w} = [w_1, \cdots, w_n]^T$ the weight vector is normal to the hyper plane, and $|b|/||\mathbf{w}||$ is defined as the distance between the plane and origin [11]. Then we can say that our n-Dimensional has been partitioned into two different regions by this new hyper plane [11]. We can go ahead and define a mapping function $y = \text{sign}(f(\mathbf{x})) \in \{1, -1\}$, i.e.,

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w} + b = \begin{cases} > 0, & y = \text{sign}(f(\mathbf{x})) = 1, \ \mathbf{x} \in P \\ < 0, & y = \text{sign}(f(\mathbf{x})) = -1, \ \mathbf{x} \in N \end{cases}$$

Any point $\mathbf{x} \in P$ which exists on the positive side is mapped to 1, while any point $\mathbf{x} \in N$ which exists on the negative side is mapped to -1. A point $\mathbf{x}$ of unknown class will be classified to P if $f(\mathbf{x}) > 0$, or N if $f(\mathbf{x}) < 0$ [11].

4.1.4 The learning problem

Given a set K training samples from two linearly separable classes P and N

$$\{(\mathbf{x}_k, y_k), k = 1, \cdots, K\}$$

where $y_k \in \{1, -1\}$ labels $\mathbf{x}_k$ to belong to either of the two classes. Our main aim is to find a hyper-plane in terms of $\mathbf{w}$ and b, that linearly separates the two classes.

Before $\mathbf{w}$ is properly trained, the actual output $y' = \text{sign}(f(\mathbf{x}))$ may not be the same as the desired output y. There are four possible cases:

	Input $(\mathbf{x}, y)$	Output $y' = \text{sign}(f(\mathbf{x}))$	result
1	$(\mathbf{x}, y = 1)$	$y' = 1 = y$	corrrect
2	$(\mathbf{x}, y = -1)$	$y' = 1 \neq y$	incorrect
3	$(\mathbf{x}, y = 1)$	$y' = -1 \neq y$	incorrect
4	$(\mathbf{x}, y = -1)$	$y' = -1 = y$	corrrect

The weight vector $\mathbf{w}$ is updated whenever the result is incorrect (mistake driven):

- If $(\mathbf{x}, y = -1)$ but $y' = 1 \neq y$ (case 2 above), then

  $$\mathbf{x}^{\text{new}} = \mathbf{w}^{\text{old}} + \eta y \mathbf{x} = \mathbf{w}^{\text{old}} - \eta \mathbf{x}$$

  When the same $\mathbf{x}$ is presented again, we have

  $$f(\mathbf{x}) = \mathbf{x}^T \mathbf{w}^{\text{new}} + b = \mathbf{x}^T \mathbf{w}^{\text{old}} - \eta \mathbf{x}^T \mathbf{x} + b < \mathbf{x}^T \mathbf{w}^{\text{old}} + b$$

  The output $y' = \text{sign}(f(\mathbf{x}))$ is more likely to be $y = -1$ as desired. Here $0 < \eta < 1$ is the learning rate.

- If $(\mathbf{x}, y = 1)$ but $y' = -1 \neq y$ (case 3 above), then

  $$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \eta y \mathbf{x} = \mathbf{w}^{\text{old}} + \eta \mathbf{x}$$

  When the same $\mathbf{x}$ is presented again, we have

  $$f(\mathbf{x}) = \mathbf{x}^T \mathbf{w}^{\text{new}} + b = \mathbf{x}^T \mathbf{w}^{\text{old}} + \eta \mathbf{x}^T \mathbf{x} + b > \mathbf{x}^T \mathbf{w}^{\text{old}} + b$$

  The output $y' = \text{sign}(f(\mathbf{x}))$ is more likely to be $y = 1$ as desired.

Summarizing the two cases, we get the learning law:

$$\text{if} \;\; y f(\mathbf{x}) = y(\mathbf{x}^T \mathbf{w}^{\text{old}} + b) < 0, \;\; \text{then} \;\; \mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \eta y \mathbf{x}$$

The two correct cases (cases 1 and 4) can also be summarized as

$$y f(\mathbf{x}) = y(\mathbf{x}^T \mathbf{w} + b) \geq 0$$

which is the condition a successful classifier should satisfy.

4.1.5 Definition

For a decision hyper-plane $\mathbf{x}^T\mathbf{w} + b = 0$ to separate the two classes P $(\mathbf{x}_i, 1)$ and N $(\mathbf{x}_i, -1)$, it has to satisfy

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 0$$

for both $\mathbf{x}_i \in P$ and $\mathbf{x}_i \in N$. Among all the planes that can actually satisfy our condition, we are concerned in finding a plane that can separate the given two classes in such a way that the margin between them is the maximum possible margin [11].

The optimal plane which maximizes the margin has to lie somewhere in between the two classes, to make the distance from the closest points on each of its sides equal. We then draw two more planes H_+ and H_- that are parallel to each other and also to H_0 and also pass through the closest point to the plain on both of its sides:

$$\mathbf{x}^T\mathbf{w} + b = 1, \quad \text{and} \quad \mathbf{x}^T\mathbf{w} + b = -1$$

All points $\mathbf{x}_i \in P$ on the positive side should satisfy

$$\mathbf{x}_i^T\mathbf{w} + b \geq 1, \quad y_i = 1$$

and all points $\mathbf{x}_i \in N$ on the negative side should satisfy

$$\mathbf{x}_i^T\mathbf{w} + b \leq -1, \quad y_i = -1$$

These can be combined into one inequality

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1, \quad (i = 1, \ldots, m)$$

The equality holds for those points on the planes H_+ or H_-. Such points are called *support vectors*, for which

$$\mathbf{x}_i^T \mathbf{w} + b = y_i$$

i.e., the following holds for all support vectors

$$b = y_i - \mathbf{x}_i^T \mathbf{w} = y_i - \sum_{j=1}^{m} \alpha_j y_j (\mathbf{x}_i^T \mathbf{x}_j)$$

4.2 Implementation

In our implementation of the SVM's we used Rapid-miner studio a tool available with multiple machine learning packages. This learner uses the Java implementation of the support vector machine mySVM by Stefan Rueping. The implementation of mySVM can both be utilized in regression task as well as classification tasks and it provides a very fast implementation which gives good results. mySVM works with all kind of functions, be it linear or quadratic and it is also useful in case of asymmetric loss functions [27]. Figure 6 below shows the design of the SVM process.

4.2.1 Design

Training Data: This corresponds to the training input file which is expected to be an excel file with labelled data in form of tuples defined above.

Testing Data: This corresponds to the testing input file which is expected to be an excel file with un-labelled data in form of tuples defined above.

SVM: This is the learner which generates a model by learning from the Training data.

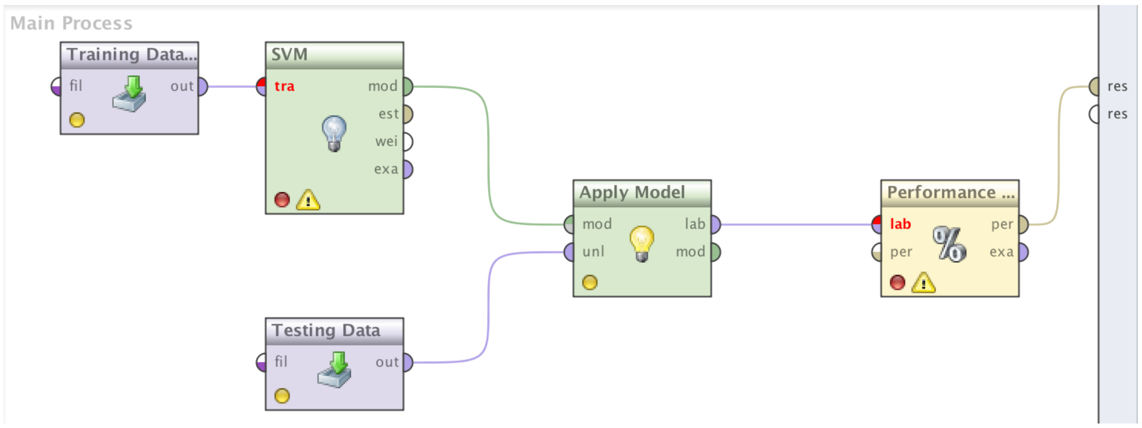


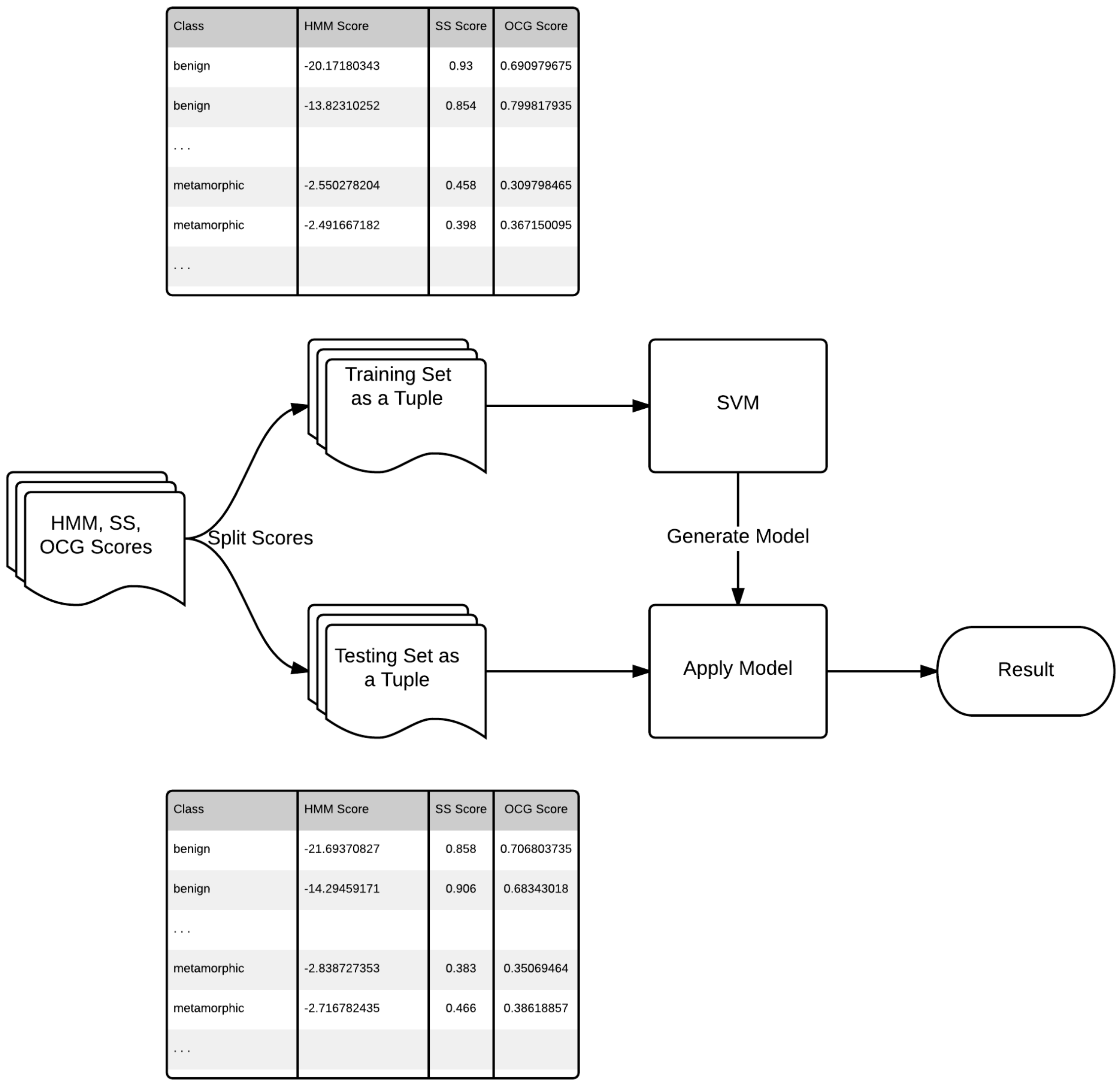
Figure 6: SVM Design

Apply Model: This operator allows us to apply a model onto incoming testing data.

Performance: This operator provides us with a list of performance criteria's which can be used to measure its performance and also for visualization purposes.

4.2.2 Algorithm

Figure 7 explains the scoring process in detail. In our scenario we are using the SVM as a classifier which can classify Benign and Metamorphic files. The features that our SVM is built on correspond to the scores that we received from HMM, Simple Substitution and Opcode graph scoring techniques as discussed in previous sections. The algorithm works in two phases -Training Phase and Testing Phase. In Training Phase, we generate a model by training on our training dataset which is formed by combining the scores from our three different techniques i.e. HMM, Simple Substitution and Opcode graph as a tuple along with a class description referring to as either Benign or Metamorphic file as shown in Figure 7. In Testing Phase, we apply the model that we created in the training phase to our testing dataset which is again the same tuple along with a class description as shown in Figure 7.



Class	HMM Score	SS Score	OCG Score
benign	-20.17180343	0.93	0.690979675
benign	-13.82310252	0.854	0.799817935
. . .			
metamorphic	-2.550278204	0.458	0.309798465
metamorphic	-2.491667182	0.398	0.367150095
. . .			

Class	HMM Score	SS Score	OCG Score
benign	-21.69370827	0.858	0.706803735
benign	-14.29459171	0.906	0.68343018
. . .			
metamorphic	-2.838727353	0.383	0.35069464
metamorphic	-2.716782435	0.466	0.38618857
. . .			

Figure 7: SVM Scoring Process

CHAPTER 5

Experiments

In this chapter we focus our attention on using the below mentioned techniques to score a representative set of metamorphic malware:

- Hidden Markov Model Method
- Opcode Graph Similarity Method
- Simple Substitution Distance Method
- Combining all the above three using SVM's

The first three of these methods work directly on the statistical properties of opcodes and the last method works by combining the scores coming out of these three techniques. Our test set of metamorphic viruses consists of 200 NGVKC files [31]. So all the results to follow in this chapter will revolve around the NGVCK family to be representative of metamorphic files [25]. The NGVCK family of viruses in previous researches has proved that it is one of the most highly metamorphic [33].

Our set of benign files consists of 40 cygwin utility files [6]. We have selected these files for our experiments because we wanted to compare our results with previous implementations which also used these files in their experiments, including [33]. Finally we will be diversifying our malware dataset by moving to a whole new families of metamorphic malware in the next chapter. We use

ROC (Receiver Operating Characteristics) curves for comparing these techniques to each other and for measuring their efficacy [4]. The Area under the curve also refer to as AUC in coming sections, which is the area under this ROC curve will provide us with a measure of the degree of correctness of each of these techniques.

We used a machine with the below mentioned configuration:

Table 8: Machine Specification

Model	MacBook Pro Retina
Processor	2.4 GHz Intel Core i5
RAM	8 GB 1600 MHz DDR3
System type	64-bit OS
Operating System	OS X 10.9.5

5.1 Receiver Operating Characteristics

Receiver Operating Characteristics or in short ROC curve is a kind of graph which is drawn to measure the correctness of a binary classifier [4]. We first calculate the TPR also known as true positive rate which is defined as the number of positives that were predicted positives in actual out of the total predicted positives. Then we calculate FPR also known as false positive rate which is defined as the negative classifications which were accidentally predicted as positives out of the total actual negatives. Once we have these two values we plot the TPR vs the FPR at different levels of threshold to generate an ROC curve. TPR is also known as sensitivity and FPR is also known as the fall out of a classification system [9].

In the following sections we will be discussing about the results of the above mentioned techniques in detail. We will be plotting ROC curves to compare their efficiency. We will also be using area under the curve or in short AUC for these ROC curves as a measure of the correctness of these scoring systems.

5.2 Hidden Markov Model Method

We did a lot of experiments so as to compare the results that we got for our implementation against the scores produced in previous researches [23]. While performing our experiments we saw that the final score produced is not effected much by the number of states that we have considered for the HMM. So, we decided to go with 2 as the default number of states for HMM in our experiments. In this particular experiment, we trained our HMM on a training set which consisted of 160 NGVCK files to generate a model which we could use for scoring similar files. Then we used this model to score another testing set which consisted of 40 NGVCK files alongwith a compare set consisting of 40 Benign files. If we look at the scatter plot in Figure 8 we can clearly see that there is a clear separation between the scores for both kind of files.

Figure 9 shows the corresponding ROC curve that has been plotted for the results received in this experiment. From the curve it is very easy to deduce that our HMM model is capable of producing great results and can easily distinguish the family viruses from benign files. The AUC for this experiment was 1.0, which further reinstates the fact that it produces perfect classification.

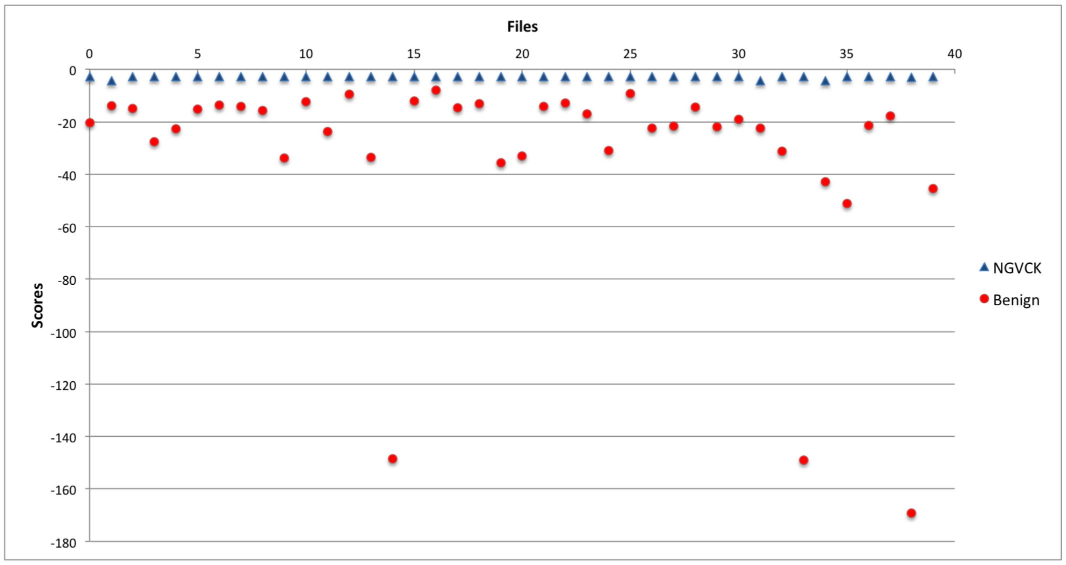


Figure 8: HMM Score Analysis

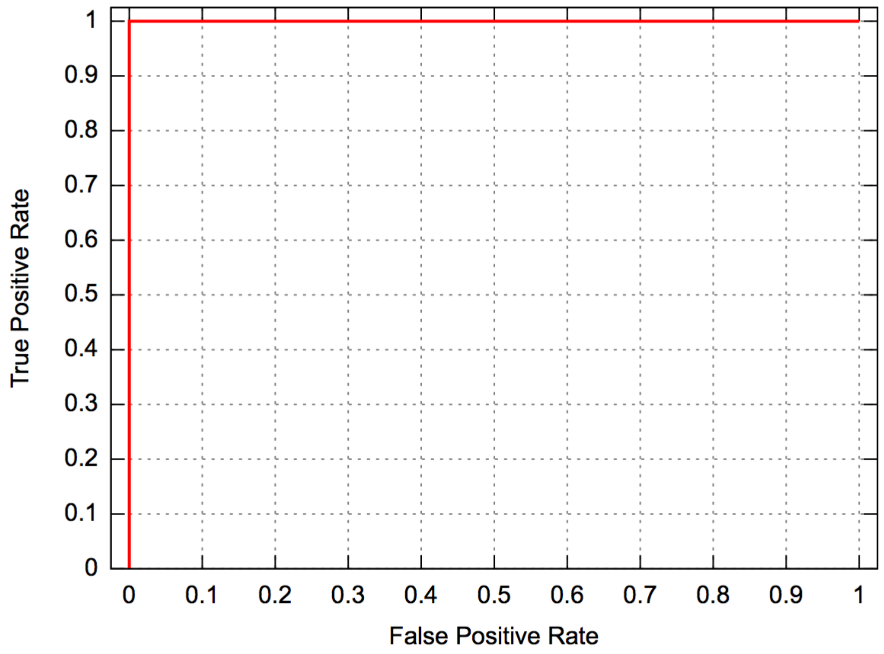


Figure 9: ROC curve for HMM

5.3 Opcode Graph Similarity

This experiment is based around the same setup where we use a subset of 160 NGVCK files for the training phase and use the remaining 40 files along with 40 cygwin utility files for the testing dataset. We use the process defined in Section 3.2.1 to establish a threshold value, which will further be used for the actual distinction between the two type of files.

Figure 10 shows a scatterplot that we got after plotting the results from our experiment. The red dots in the scatter plot denote the similarity score between two different types of files in our experiment i.e. the malware and benign files, whereas blue triangles denote the similarity score between two files which belong to same malware family. Because there is a clear separation between the two scores in Figure 10, we can clearly conclude that this method is able to distinguish between the files correctly.

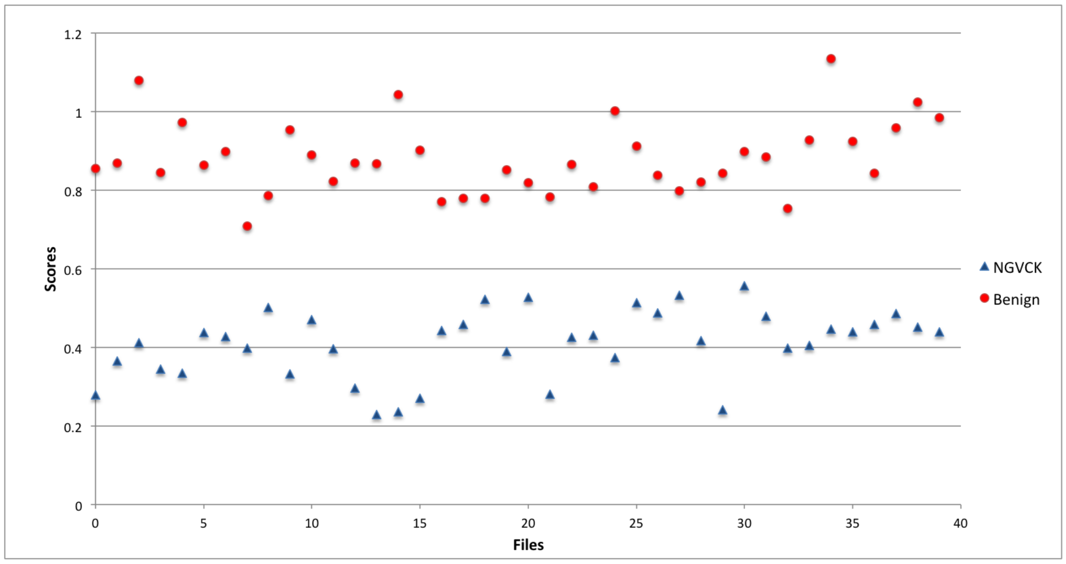


Figure 10: Opcode Graph Similarity Score Analysis

Figure 11 shows the corresponding ROC curve that has been plotted for the results received in this experiment. From the curve it is very easy to deduce that our Opcode graph similarity method is capable of producing great results and can easily distinguish the family viruses from benign files. The AUC for this experiment was 1.0, which further reinstates the fact that it produces perfect classification.

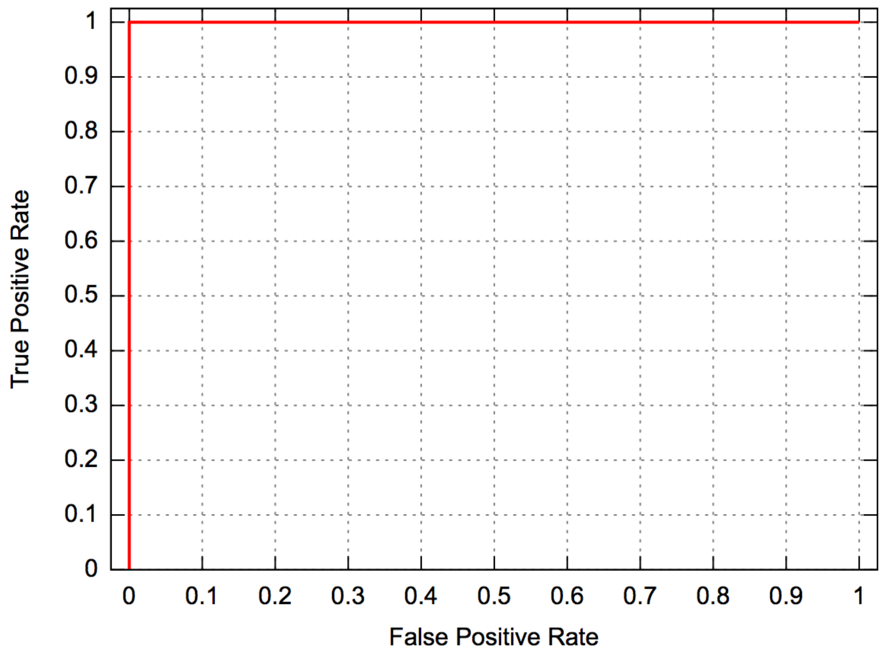


Figure 11: ROC curve for Opcode Graph Similarity Method

5.4 Simple Substitution Distance

In this experiment as well we used the same malware and benign files as discussed above. The scores obtained in this experiment are plotted on scatter plot as shown in Figure 12. From the scatter plot it is quite clear that the malware files produce a lower score for this technique as compared to the benign files which are quite different.

Figure 13 shows the corresponding ROC curve that has been plotted for

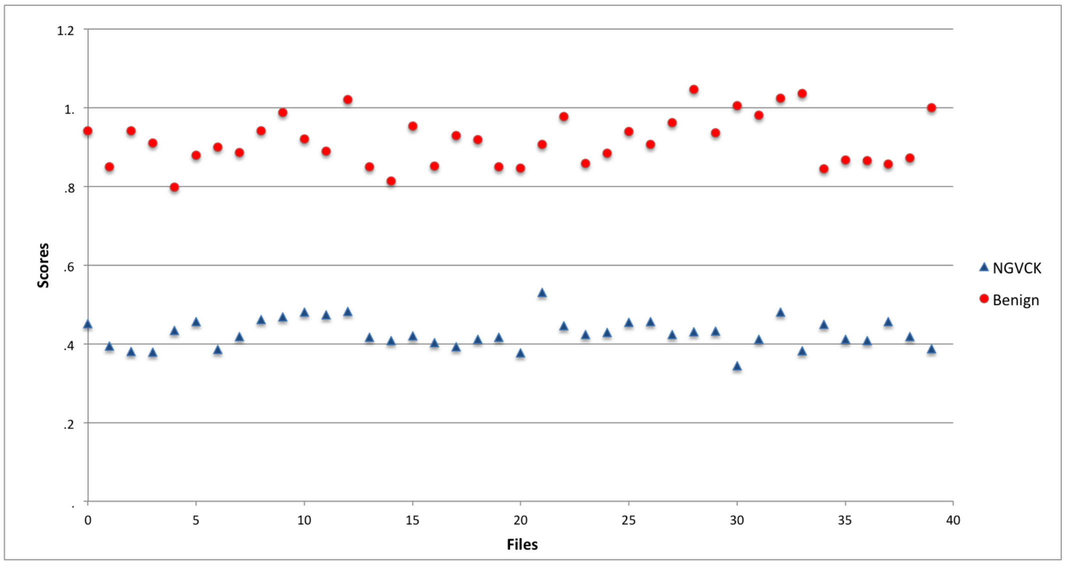


Figure 12: Simple Substitution Method Score Analysis

the results received in this experiment. From the curve it is very easy to deduce that our Simple Substitution method is capable of producing great results and can easily distinguish the family viruses from benign files. The AUC for this experiment was 1.0, which further reinstates the fact that it produces perfect classification.

5.5 Support Vector Machine

In this experiment we are not devising a new technique for detection of metamorphic malware by working on any of its characteristics, rather our aim here is to see if we could combine the scores coming from the three techniques defined above and use an SVM classifier to classify them. So, here rather than directly working with malware and benign files we will be working with scores generated by the three previous techniques as described in Section 5.2, 5.3, 5.4. Once we have scores from the above three techniques, we split them into two files with each containing a tuple:

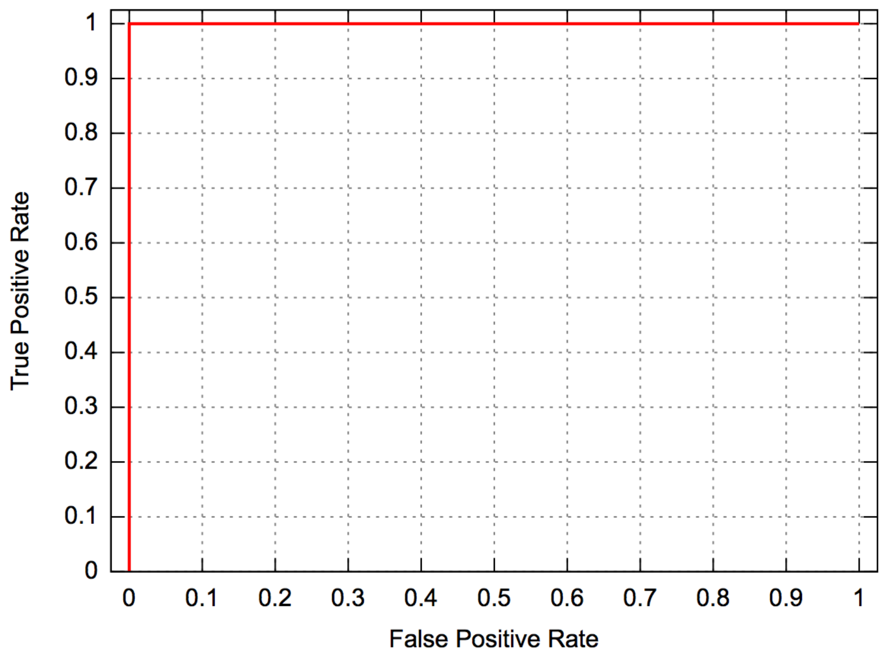


Figure 13: ROC curve for Simple Substitution Method

$(\textit{actual class label}, \textit{HMM score}, \textit{OGS score}, \textit{SSD score})$

The output from the support vector machine is a file which contains a tuple:

$(\textit{actual class label}, \textit{HMM score}, \textit{OGS score}, \textit{SSD score}, \textit{predicted class label})$

as shown in Figure 7. We can then compare the predicted class label against the actual class for calculating the performance of the model. The scores obtained in this experiment are plotted on scatter plot as shown in Figure 14. It can be observed that our SVM technique is able to distinguish between metamorphic malware and benign file with an accuracy of 100%.

Figure 15 shows the corresponding ROC curve that has been plotted for the results received in this experiment. From the curve it is very easy to deduce that our Support vector machine method is also capable of producing results that are similar to the one's produced by the underlying techniques. The AUC for this experiment was 1.0, which shows that it can easily distinguish

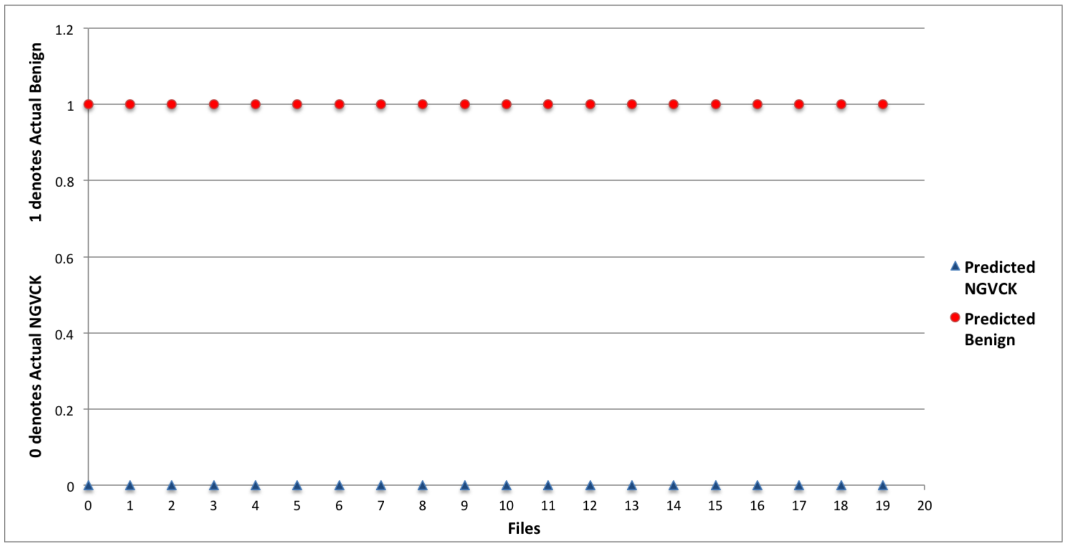


Figure 14: Support Vector Machine Score Analysis

the metamorphic family files from benign files.

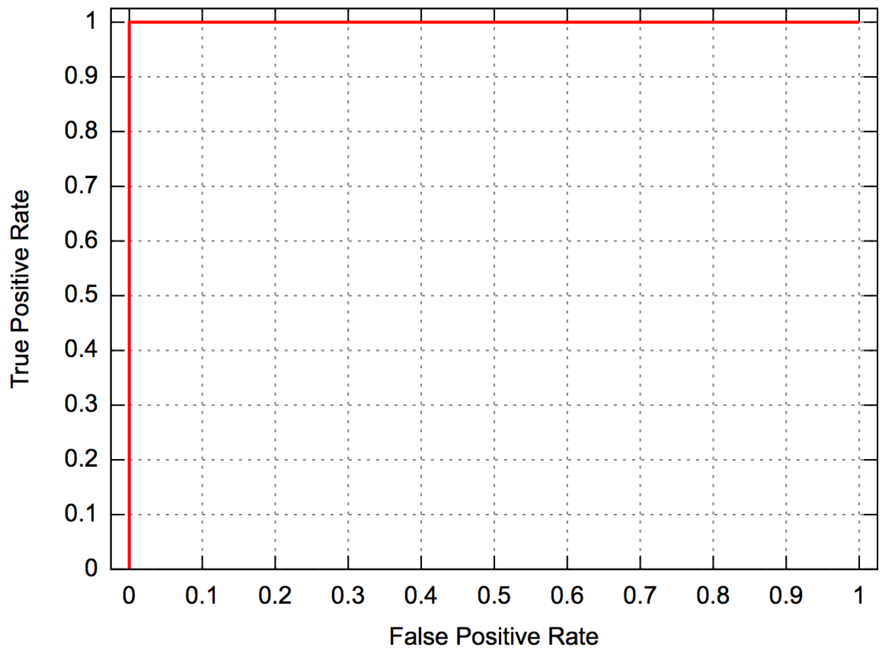


Figure 15: ROC curve for Support Vector Machine Method

In this experiment we have proved that Support Vector Machines can be used to detect metamorphic malware by using the scores of Hidden Markov

Model, Simple Substitution Distance and Opcode Graph Similarity techniques. In the coming sections we will do an in-depth analysis of the robustness of our underlying techniques that we have discussed above.

CHAPTER 6

Results

In the previous section we have seen that Support Vector Machines can be used to detect malware files from benign files by using scores from different techniques. In the coming sections we will test the robustness of each of these techniques. Our main aim is to prove that Support Vector Machines can serve as a more robust detection strategy even when the underlying techniques have failed.

6.1 Attacks on Detection Techniques

In this section, we consider the robustness of the above mentioned metamorphic detection techniques. In previous researches [15] for attacks on these statistical metamorphic detection techniques, it has been observed that morphing the metamorphic malware files by inserting code from benign files gives the best results.

6.1.1 Morphing Techniques

In [15], a morphing engine was developed in order to evade detection by known machine learning techniques such as the Hidden Markov Model technique. The engine works by inserting dead code from a benign file into a metamorphic file so as to beat the statistics of the metamorphic file which make it susceptible to detection by HMM [23].

Two different forms of morphing strategies have been discussed in paper [15]:

- Block Morphing
- Random Morphing

In the first case, which we refer to as “block morphing”, the dead code is inserted as a single block of code which is placed into the malware file at one location. Whereas in “random morphing”, the dead code is inserted uniformly into the malware file at constant intervals so as to spread the dead code throughout its body. The paper proves that this method of insertion of “dead code” into the malware file from a benign file is capable of defeating the HMM detection, i.e. it is able to induce false negatives or false positives. In the paper it has also been shown that the first method of insertion that is block morphing is way more effective then the random morphing method.

We performed experiments at different levels of block morphing. The amount of dead code inserted into the file depends on the size of the malware file and is usually inserted as a percentage of its size. For example in order to do a 20% morphing into a metamorphic file of 400 lines we would have to insert 80 lines into it from a randomly selected benign file. The main point to be noted here is that all the steps we are performing here are happening at the assembly code level, but we are just making these changes to the static opcode files that we already have. The assumption here is that in real world a malware writer would have to put in a lot of effort to do the same modifications into an executable, because he has to make sure that none of the inserted code gets executed. Another thing that he has to worry about is that the inserted code should not look obvious and should not be easily detected by someone debugging the executable. So, by ignoring these practical problems which are involved in the insertion of dead code into a malware file, we are considering the worst possible case that a malware writer would have to go through. In the coming sections we look at results of our detection techniques against block morphed NGVCK malware files.

6.2 Results for Morphed Malware

6.2.1 Hidden Markov Model Method

We started off by inserting different levels of opcodes into the NGVCK files from randomly selected benign files. In Figure 16, we show a scatter plot for HMM results for 10% morphed NGCVK malware files vs Benign files, from the scatter plot it is clearly visible that there is no clear distinction between the malware scores and the benign scores at 10% block morphing.

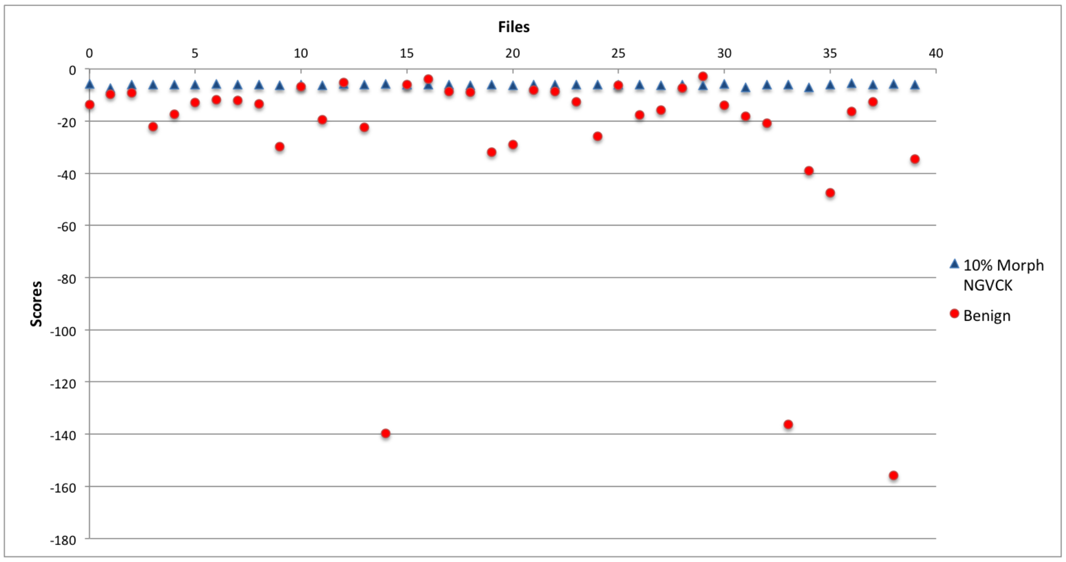


Figure 16: HMM - 10% morphed Scores Analysis

The ROC curve shown in Figure 17 below validates the fact, it can be clearly inferred from the ROC curve that some of the files have been miss-classified. The AUC for Figure 17 has been reduced to 0.90125 from its initial value of 1.000 for un-morphed Malware files vs Benign Files.

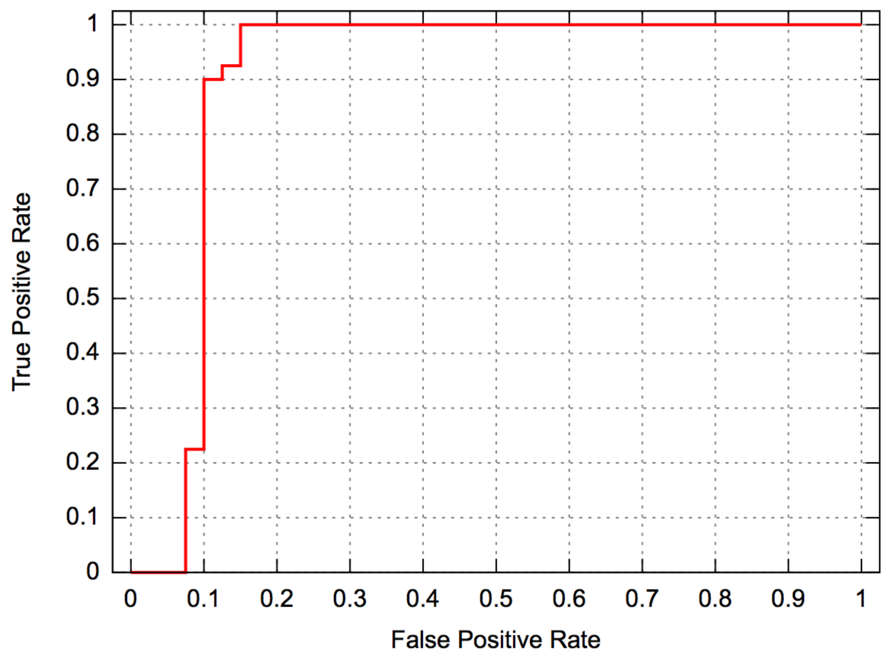


Figure 17: HMM - ROC curve for 10% morphing

Table 9 shows AUC value for ROC Curves plotted for different levels of block morphing. From the table we can see that there is a considerable drop in the AUC values at 10% and 20% block morphing levels, but the AUC values for morphing levels greater than 20% tend to stabilize. Figure 18 shows the graph for these results.

Table 9: HMM AUC values for different levels of Block Morphing

Morphing percentage	**AUC**
0%	1.00000
10%	0.90125
20%	0.81625
30%	0.81875
40%	0.82250

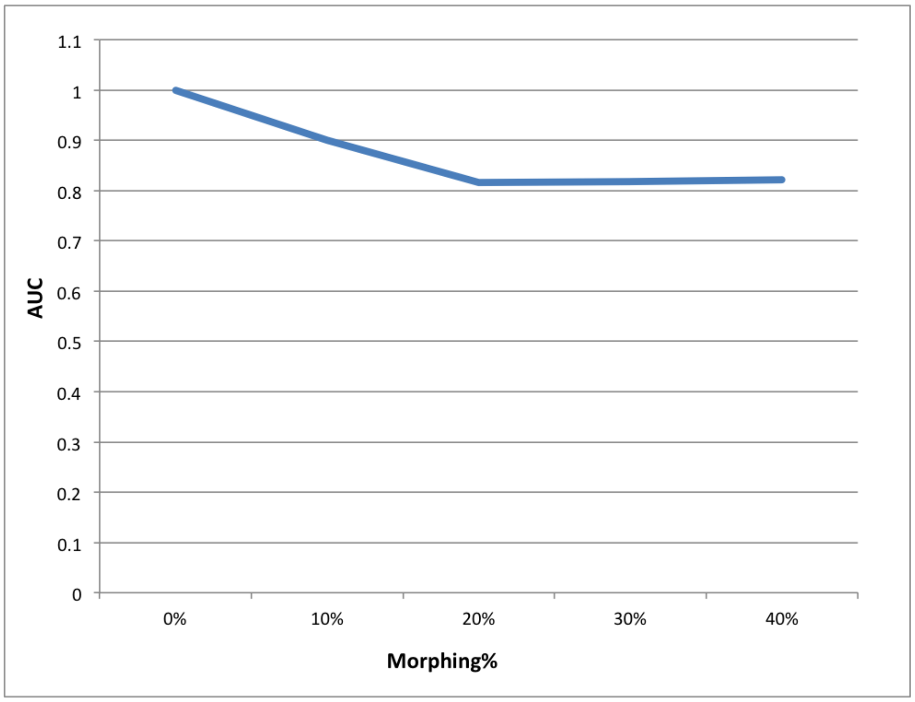


Figure 18: HMM AUC at different Morphing levels(Table 9)

6.2.2 Opcode Graph Similarity Method

In this experiment we inserted different percentages of opcodes into the NGVCK files from randomly selected benign files. In Figure 19 we show a scatter plot for Opcode Graph Similarity method results for 60% morphed NGCVK malware files vs Benign files, from the scatter plot is clearly visible that there is no clear distinction between the malware scores and the benign scores at 10% block morphing. This technique turned out to be comparatively more robust to HMM technique as it could withstand morphing levels of upto 50% before showing any kind of miss-classification.

The ROC curve shown in Figure 20 below validates the fact, it can be clearly seen from the ROC curve that some of the files have been miss-classified at 60% block morphing level. The AUC for Figure 20 has been reduced to

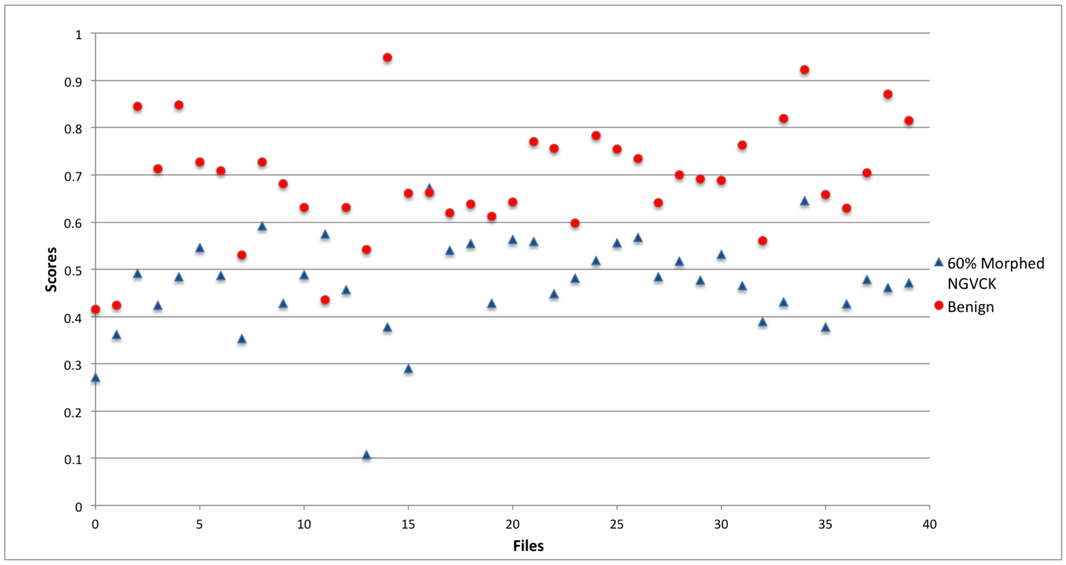


Figure 19: OGS - 60% morphed Scores Analysis

0.91250 from its initial value of 1.000 for un-morphed Malware files vs Benign Files.

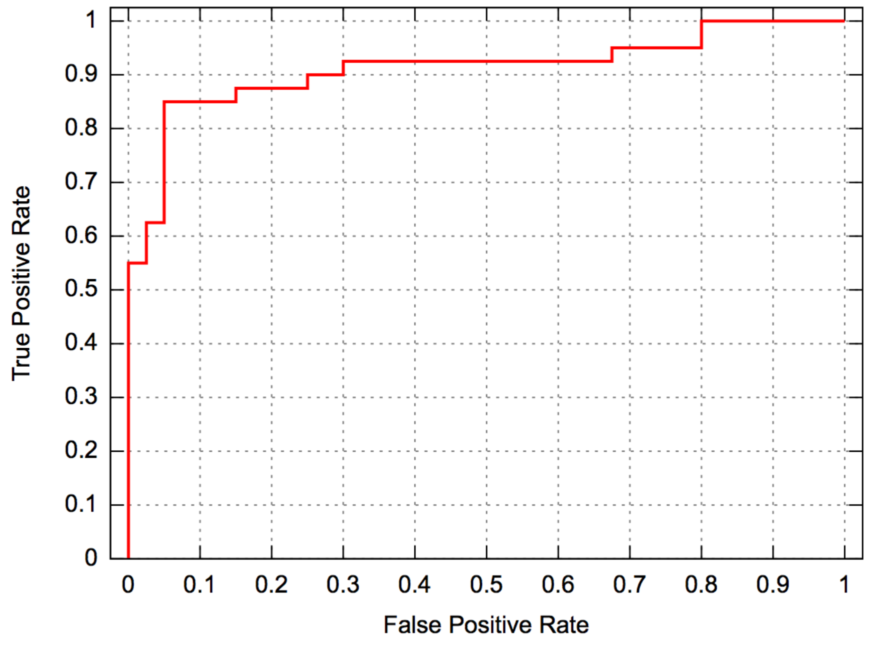


Figure 20: OGS - ROC curve for 60% morphing

Table 10 shows AUC values for ROC Curves plotted for different levels of block morphing. From the table we can see that Opcode graph similarity technique is more robust than HMM technique as at relatively moderate levels of morphing it produced perfect results in the form of AUC value of 1.0. It was only at about 60% block morphing the the results started to deteriorate and the AUC value dropped to 0.91250. Figure 21 shows the graph for these results.

Table 10: OGS AUC values for different levels of Block Morphing

Morphing percentage	**AUC**
0%	1.00000
10%	1.00000
20%	1.00000
30%	1.00000
40%	1.00000
50%	1.00000
60%	0.91250

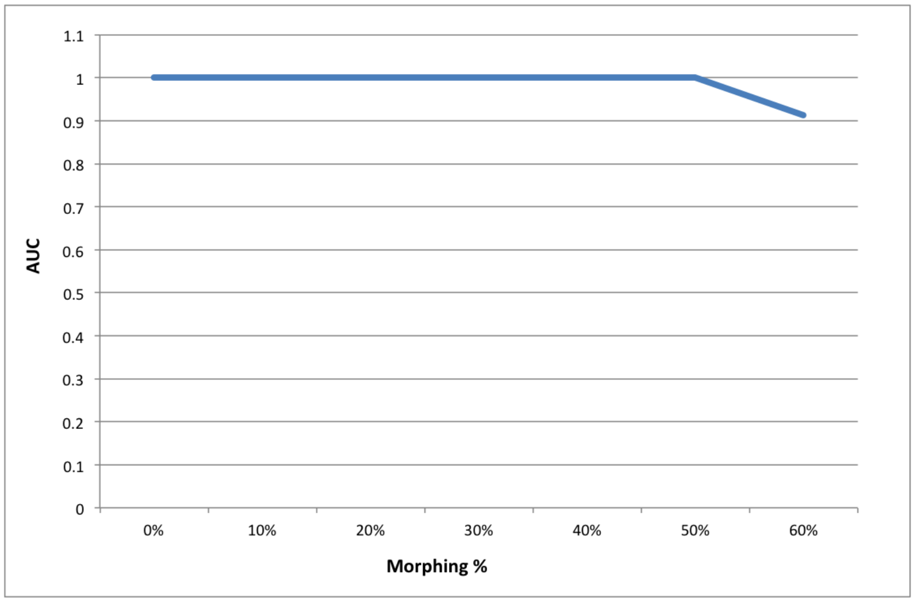


Figure 21: OGS AUC at different Morphing levels(Table 10)

6.2.3 Simple Substitution Distance Method

In this experiment as well we inserted different percentages of opcodes into the NGVCK files from randomly selected benign files. In Figure 22 we show a scatter plot for Simple Substitution Distance method scores for 80% morphed NGCVK malware files vs Benign files, from the scatter plot it is clearly visible that there is no clear distinction between the malware scores and the benign scores at 80% block morphing. This technique turned out to be comparatively more robust to HMM technique, because it could withstand morphing levels of upto 50% before showing any kind of miss-classification. It performed a little better in comparison to the Opcode Graph Similarity technique as well, it took about 80% block morphing to drop the AUC value to 0.89875 whereas similar AUC was achieved in case of Opcode Graph Similarity at a morphing level of 60%.

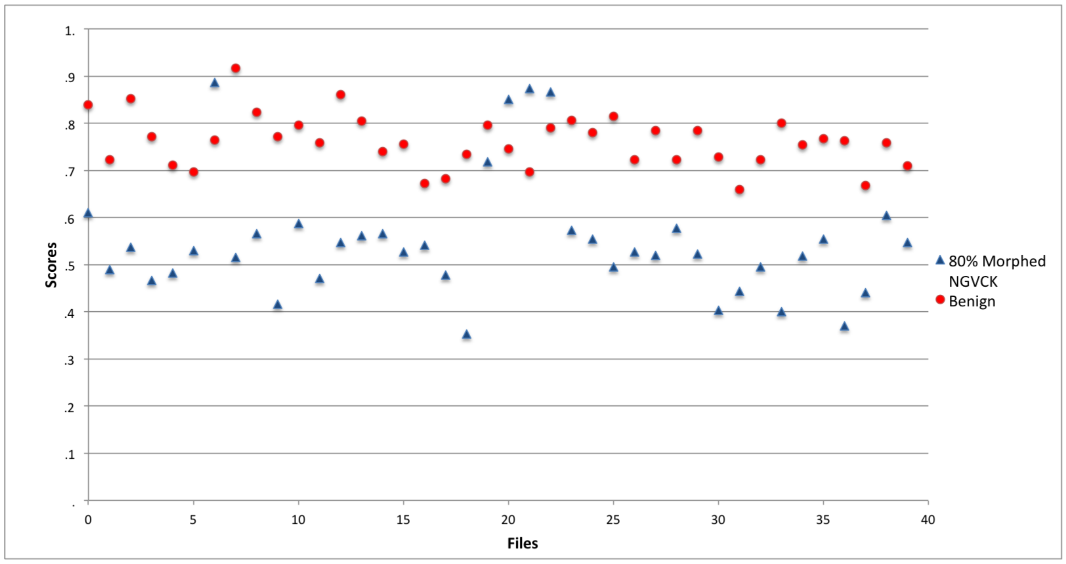


Figure 22: SS - 80% morphed Scores Analysis

The ROC curve shown in Figure 27 below validates the fact, it can be clearly seen from the ROC curve that some of the files have been miss-classified at 80% block morphing level. The AUC for Figure 27 has been reduced to 0.89875 from its initial value of 1.000 for un-morphed Malware files vs Benign Files.

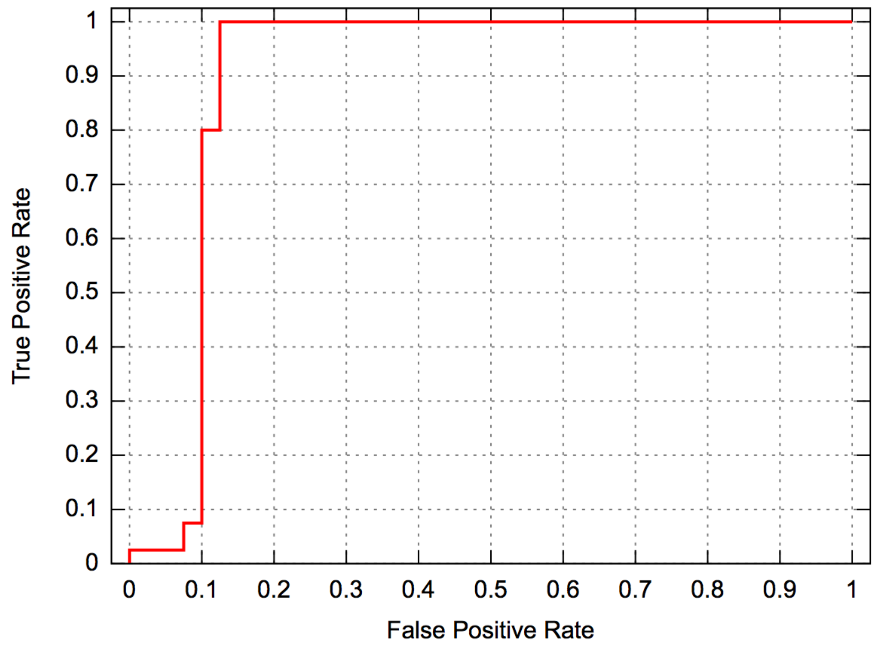


Figure 23: SS - ROC curve for 80% morphing

Table 11 shows AUC values for ROC curves plotted for different levels of block morphing. From the table we can see that Opcode graph similarity technique is more robust than HMM technique as at relatively moderate levels of morphing it produced perfect results in the form of AUC value of 1.0. It was only at about 60% block morphing that the results started to deteriorate and the AUC value dropped to 0.91250. Figure 24 shows the graph for these results.

Table 11: SS AUC values for different levels of Block Morphing

Morphing percentage	**AUC**
0%	1.00000
10%	1.00000
20%	1.00000
30%	1.00000
50%	1.00000
60%	0.99750
70%	0.95063
80%	0.89875

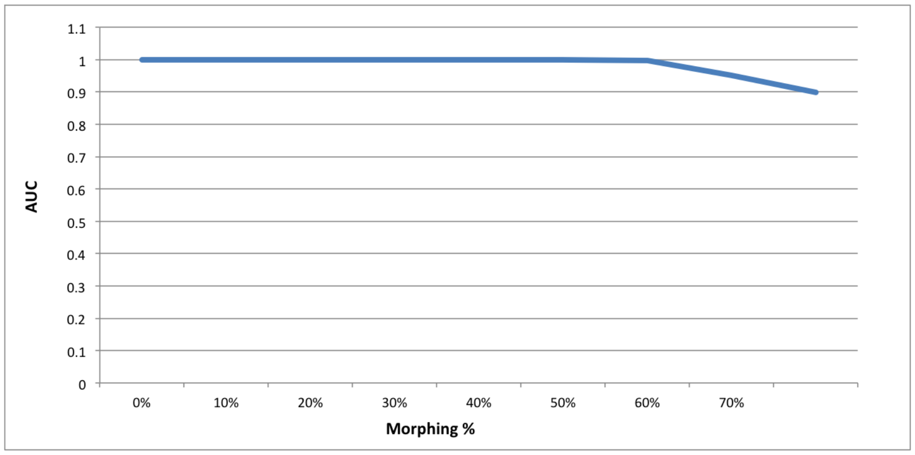


Figure 24: SSD AUC at different Morphing levels(Table 11)

6.2.4 Combining Scores using SVM

In this experiment we want to see if SVM can provide us with better result than individual scores from the three techniques mentioned above. Because the maximum morphing % used to break all the techniques was registered in case of Simple Substitution method, wherein it took about 80% block morphing to bring the AUC down by a considerable level, we choose 80% morphing as a benchmark. Now we generate scores for each of the three techniques at 80% block morphing. Figure 25, 26, 27 below show the ROC curves generated by each of the three techniques at 80% block morphing.

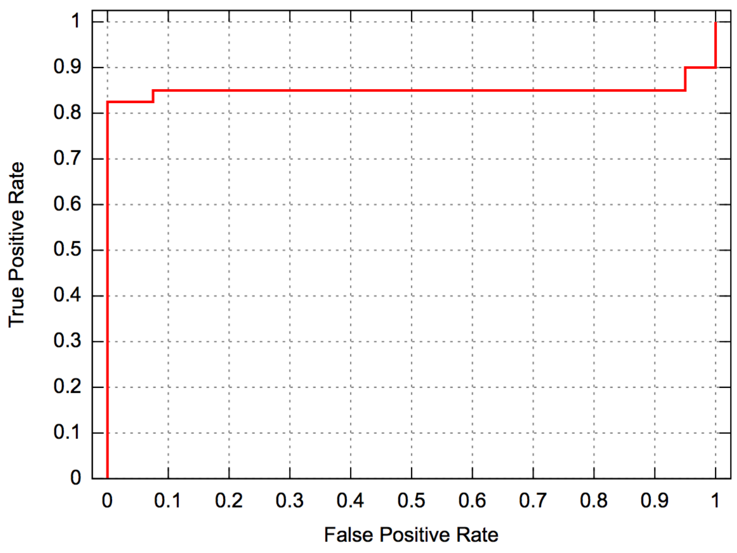


Figure 25: HMM - ROC curve for 80% morphing

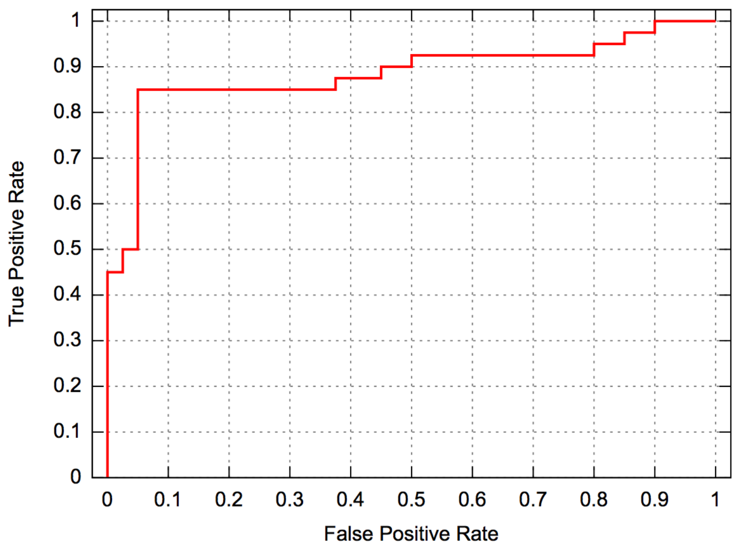


Figure 26: OGS - ROC curve for 80% morphing

In comparison to these above three, we can see that Support vector machine can produce better results by combining the scores from the above three techniques. Figure 28 shows the ROC curve generated by SVM scores. The AUC for this ROC curve is 1.0, which is better than the AUC achieved by any of the three techniques individually at 80% block morphing level.

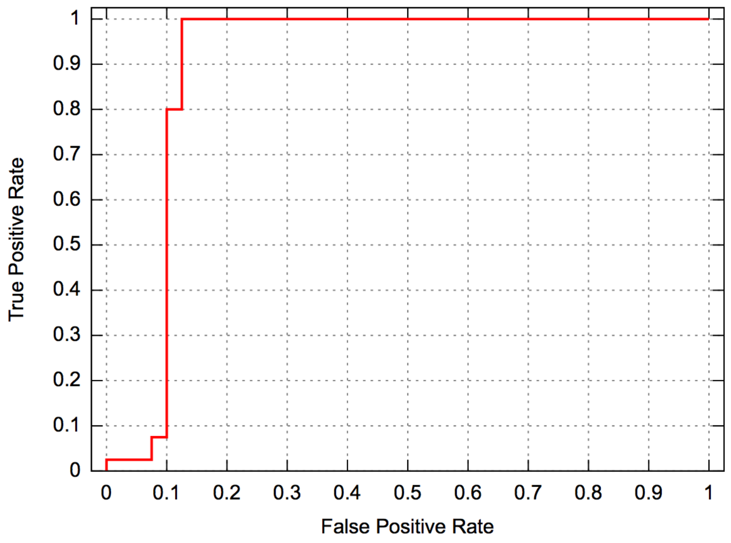


Figure 27: SS - ROC curve for 80% morphing

We summarize these results in Table 12. If we take a look at the table it is evident that SVM produced a perfect AUC score of 1.0 when all the individual techniques had failed with AUC scores around 0.9. So we can conclude that SVM's can produce better results by combining scores generated by different malware detection techniques. Figure 29 shows the graph for these results.

Table 12: Comparison of Scores at 80% Block Morphing

Scoring Technique	**AUC**
Hidden Markov Models	0.85062
Opcode Graph Similarity	0.89875
Simple Substitution Distance	0.88437
Support Vector Machine	1.00000

6.2.4.1 SVM Kernel functions comparison

We tested different kernel methods for our SVM implementation, but got the best results with radial kernel as shown in Table 13. So from here on, all our experiments will be using the radial function as the kernel function.

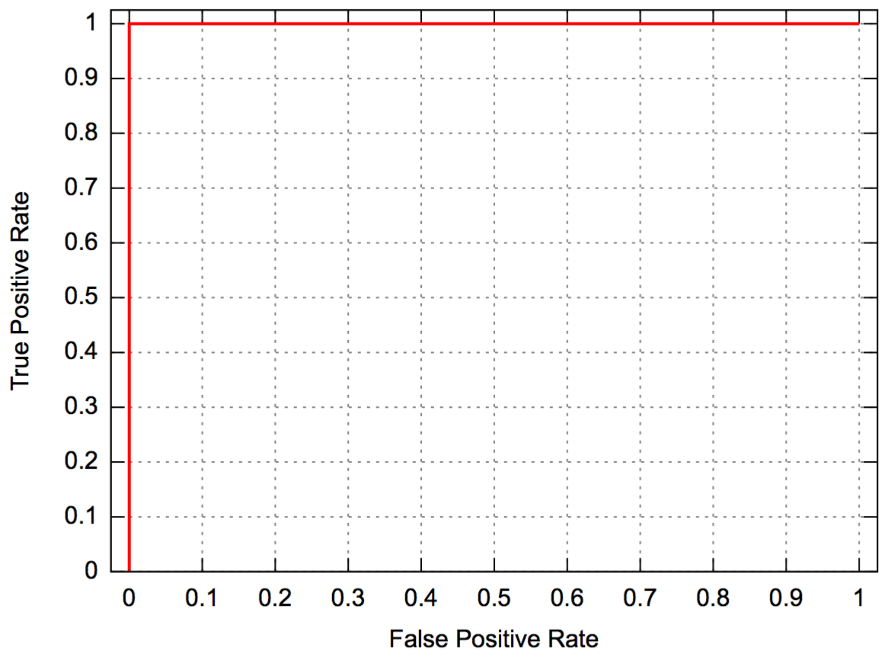


Figure 28: SVM - ROC curve for 80% morphed scores

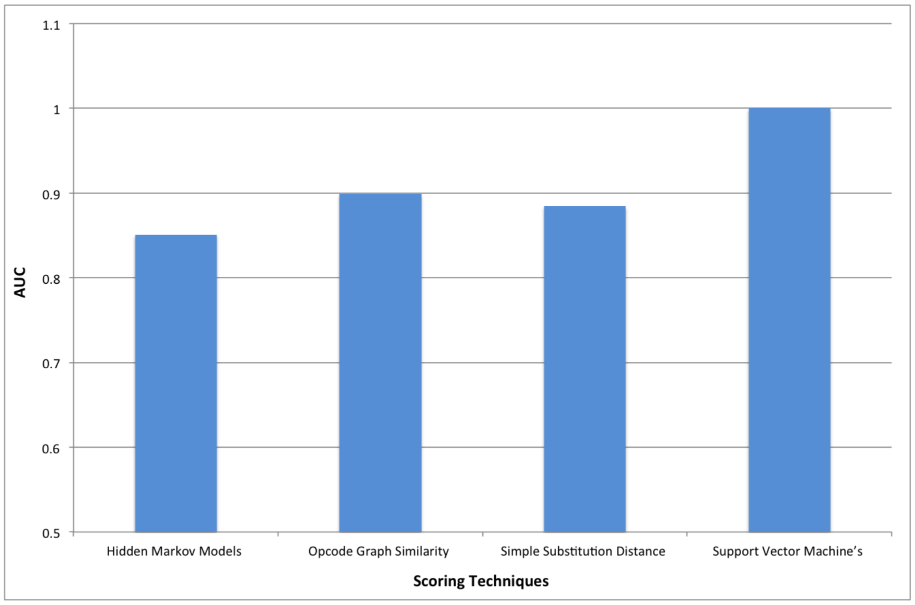


Figure 29: AUC Comparison for 80% Block Morphed scores(Table 12)

Table 13: Comparison of SVM Kernel Functions at 80% Block Morphing

SVM Kernel	AUC
Dot	0.918
Polynomial	0.860
Neural	0.850
Radial	1.000

6.3 SVM vs Individual Techniques

In this section we compare the results produced by SVM against the results produced by each of our individual techniques. In Section 6.2.4 we saw that SVM scores outperformed each of the techniques at a morphing level of 80%. Next we try to establish a morphing ratio wherein SVM's perform better than each of the individual techniques. Table 14 shows AUC values at different morphing percentages for Hidden Markov Model(HMM), Opcode Graph Similarity(OGS), Simple Substitution Distance(SSD) and Support Vector Machine(SVM) techniques respectively. From the table we can see that SVM is a much more robust technique as compared to the other three.

Table 14: Comparison of Scores at Different Morphing levels

Morphing %	HMM AUC	OGS AUC	SSD AUC	SVM AUC
0%	1	1	1	1
10%	0.90125	1	1	1
20%	0.81625	1	1	1
30%	0.81875	1	1	1
40%	0.82250	1	1	1
50%	0.86250	1	1	1
60%	0.87875	0.91250	0.99750	1
70%	0.85437	0.90687	0.95063	1
80%	0.85062	0.88437	0.89875	1
90%	0.87625	0.87250	0.93812	1
100%	0.90000	0.85370	0.90750	1
110%	0.9	0.79563	0.90188	0.97906
120%	0.9	0.78125	0.87438	0.95875

Figure 30 shows a line graph depicting the change in AUC values at different morphing levels for all the four techniques. We can clearly see that SVM performs way better than the other techniques even at 100% morphing level. The morphing ratio where SVM shows up to be better than the three underlying techniques is 60%. Now that we have proved that SVM's can be used to combine results from different malware detection techniques and still produce better results, we would like to put it to test against some other virus families to see how it performs as compared to the three underlying techniques.

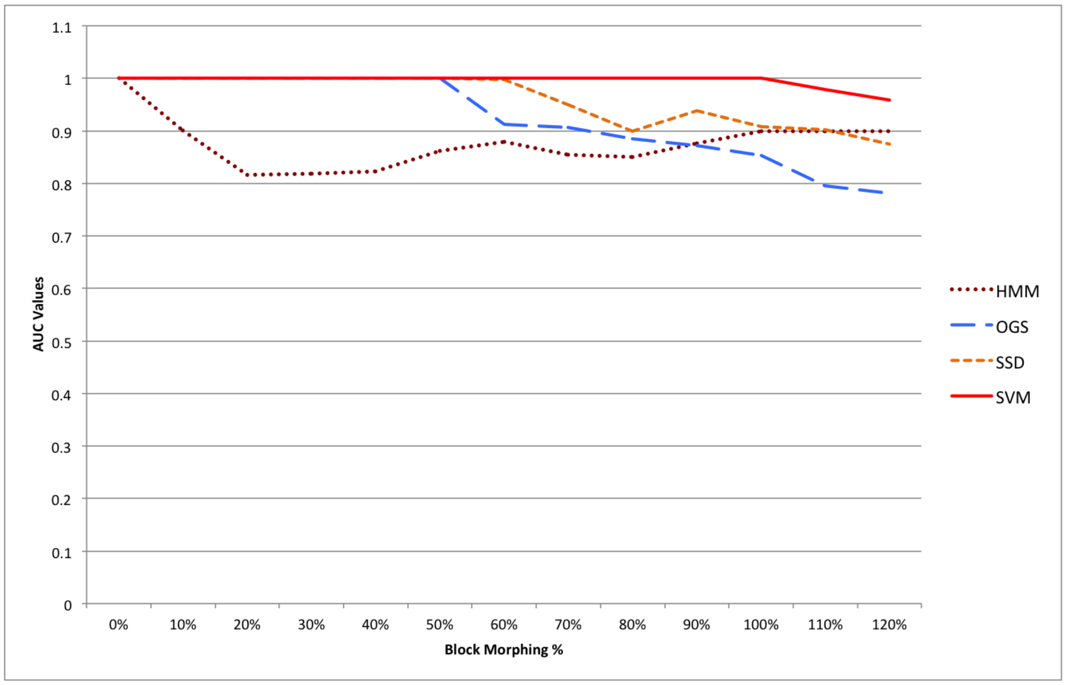


Figure 30: Comparison of AUC Values at different Morphing levels

6.3.1 ZeroAccess Malware

In this experiment we want to validate how SVM performs against the individual techniques for ZeroAccess Malware. We extract the opcode sequences from the ZeroAccess malware files and use them as the dataset for our malware detection techniques. For comparison we use the same set of benign files that we used to compare against the NGVCK virus. Table 15 shows the comparison between AUC values for SVM and the underlying techniques, we can see that for ZeroAccess Malware SVM outperforms the other three techniques and provides us with better result.

Table 15: Comparison of Scores for Zero Access Malware

Scoring Technique	**AUC**
Hidden Markov Models	0.97875
Opcode Graph Similarity	0.47531
Simple Substitution Distance	1
Support Vector Machine's	1

6.3.2 Zbot Malware

In this experiment, we extract the opcode sequences from the Zbot malware files and use them as the dataset for our malware detection techniques. Table 16 shows the comparison between AUC values for SVM and the underlying techniques, we can clearly see that for Zbot Malware as well SVM outperforms the other three techniques and provides us with better result.

Table 16: Comparison of Scores for Zbot Malware

Scoring Technique	**AUC**
Hidden Markov Models	0.9875
Opcode Graph Similarity	0.61938
Simple Substitution Distance	0.675
Support Vector Machine's	1

6.3.3 WinWebSec Malware

In this experiment as well we followed the same approach to generate scores as in the above section. Table 17 shows the comparison between AUC values for SVM and the underlying techniques. For WinWebSec malware family HMM was able to distinguish between the malware and benign files with an AUC of 1. Although the other two techniques were not able to distinguish between malware and benign files, but by combining the scores from the three techniques SVM was able to distinguish between the files with an AUC value of 1. So we can see that in the above three cases SVM has performed at least as well or better than the three individual techniques.

Table 17: Comparison of Scores for WinWebSec Malware

Scoring Technique	**AUC**
Hidden Markov Models	1
Opcode Graph Similarity	0.85437
Simple Substitution Distance	0.93563
Support Vector Machine's	1

6.3.4 SmartHDD Malware

In this experiment as well we followed the same approach to generate scores as in the above section. This was the first experiment where we saw that HMM performed better than SVM technique. Although the AUC value produced by SVM was better than the AUC value for all the techniques but HMM also produced good scores in this case. Table 18 shows the comparison between AUC values for SVM and the underlying techniques for SmartHDD malware.

Table 18: Comparison of Scores for SmartHDD Malware

Scoring Technique	**AUC**
Hidden Markov Models	0.99875
Opcode Graph Similarity	0.94875
Simple Substitution Distance	0.91156
Support Vector Machine's	1

6.3.5 Harebot Malware

In this experiment we saw that that HMM performed as good as the SVM technique. The AUC value produced by SVM was far better than the AUC value for both Opcode graph similarity and Simple substitution distance. The scores for both the Opcode graph technique as well as Simple substitution technique were very low and not suitable for any kind of classification task. Table 19 shows the comparison between AUC values for SVM and the underlying techniques for Harebot malware.

Table 19: Comparison of Scores for Harebot Malware

Scoring Technique	**AUC**
Hidden Markov Models	1
Opcode Graph Similarity	0.4
Simple Substitution Distance	0.6125
Support Vector Machine's	1

6.3.6 Sesh Malware

This was another experiment where we saw that HMM performed better than the other two techniques but was still outperformed by SVM. Although the AUC values from Opcode graph and Simple substitution techniques were not sufficient to be used for practical malware detection purposes. Table 19 shows the comparison between AUC values for SVM and the underlying techniques for Sesh malware.

6.3.7 Combined results

Here we look at the results from the above sections combined together in Table 21. A bar graph representing the same scores is shown in Figure 31.

Table 20: Comparison of Scores for Sesh Malware

Scoring Technique	**AUC**
Hidden Markov Models	0.994
Opcode Graph Similarity	0.608
Simple Substitution Distance	0.583
Support Vector Machine's	1

From the graph it is quite clear that SVM produce better results as compared to the other three techniques when AUC values for each of the techniques are above 0.7 in general. But if the AUC values from our underlying techniques deteriorate further, then SVM scores are no better than individual scores.

Table 21: Combined Results

Malware	**HMM AUC**	**OGS AUC**	**SSD AUC**	**SVM AUC**
WinWebSec	1	0.85437	0.93563	1
Zbot	0.9875	0.61938	0.675	1
ZeroAccess	0.97875	0.47531	1	1
Sesh	0.994	0.608	0.583	1
Harebot	1	0.4	0.6125	1
SmartHdd	0.99875	0.94875	0.91156	1

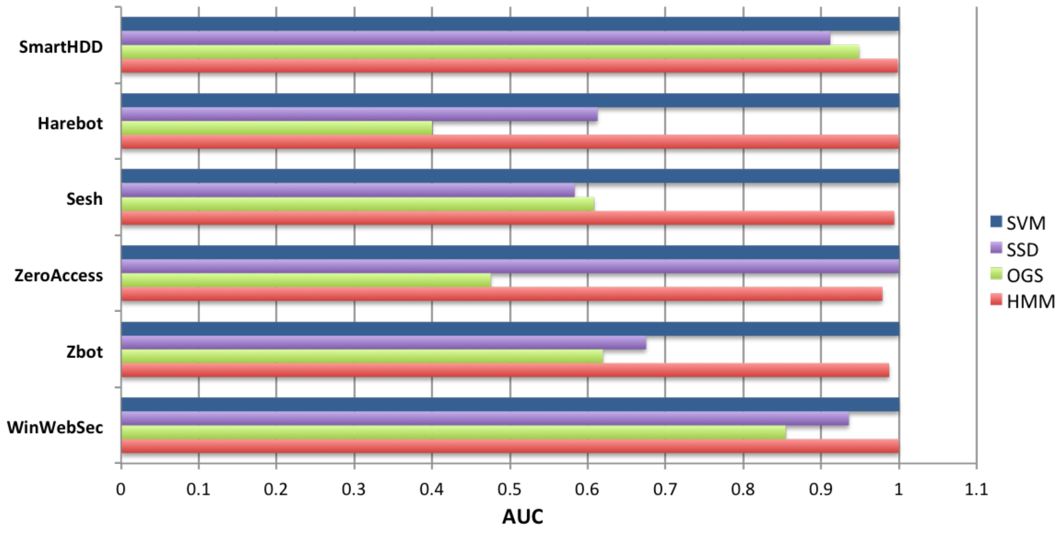


Figure 31: Combined AUC Comparison(Table 21)

6.3.8 Morphing results

After having the scores from all the techniques compared against the malicia malware dataset, we wanted to see how the scores compared against each other for different levels of morphing percentages against each of these malware families. Tables 22,23,24,25,26,27 show the results for these experiments.

Table 22: Winwebsec

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	1	0.85437	0.93563	1
10	1	0.85937	0.91969	1
30	0.885	0.85438	0.91563	0.978
50	0.84875	0.81813	0.89437	0.938
70	0.82063	0.79062	0.86719	0.905
90	0.73063	0.76187	0.85438	0.905
110	0.72875	0.73312	0.85219	0.93
130	0.71312	0.70563	0.815	0.903
150	0.69063	0.6725	0.75969	0.867

Table 23: Zeroaccess

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	1	0.47531	1	1
10	0.42063	0.43125	1	1
30	0.1325	0.4125	1	1
50	0.11875	0.37	1	1
70	0.115	0.3825	1	1
90	0.12188	0.3775	0.981	1
110	0.13875	0.37125	0.973	0.998
130	0.14625	0.3625	0.969	0.975
150	0.145	0.33125	0.943	0.932

Table 24: Zbot

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	0.9875	0.61938	0.675	1
10	0.9725	0.57875	0.77563	0.998
30	0.82875	0.51875	0.87125	0.99
50	0.79437	0.46875	0.85469	0.95
70	0.78375	0.43	0.8525	0.938
90	0.7975	0.4025	0.8275	0.952
110	0.70688	0.365	0.7825	0.905
130	0.69188	0.3375	0.74	0.86
150	0.43938	0.32875	0.68219	0.7

Table 25: Harebot

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	1	0.4	0.6125	1
10	0.78937	0.4075	0.69719	0.972
30	0.59437	0.36875	0.70281	0.872
50	0.515	0.33375	0.68812	0.702
70	0.52687	0.34375	0.72875	0.74
90	0.27125	0.32	0.72469	0.705
110	0.29125	0.30875	0.71188	0.73
130	0.21625	0.305	0.71469	0.822
150	0.22438	0.305	0.66844	0.9

Table 26: Security Shield

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	0.994	0.608	0.58313	1
10	0.99375	0.59625	0.58312	1
30	0.77313	0.64625	0.76	0.87
50	0.75188	0.615	0.79375	0.798
70	0.7825	0.5925	0.76531	0.855
90	0.785	0.58375	0.7175	0.892
110	0.78187	0.5625	0.73625	0.865
130	0.72688	0.555	0.71562	0.788
150	0.7275	0.5425	0.68406	0.725

Table 27: Smart HDD

Morphing	AUC			
Percent	HMM	OGS	SSD	SVM
0	0.99875	0.94875	0.91156	1
10	0.9575	0.93125	0.87875	0.985
30	0.89	0.84	0.90844	1
50	0.8425	0.78313	0.88018	0.932
70	0.62062	0.72875	0.85781	0.885
90	0.46312	0.7	0.845	0.745
110	0.40312	0.68062	0.83375	0.778
130	0.39687	0.66875	0.79906	0.855
150	0.37	0.66063	0.74813	0.727

CHAPTER 7
Conclusion and Future Work

The aim of this experiment was to design and test a system which could combine the scores generated by existing detection techniques and produce better results. The main idea behind it was that if we could combine the scores generated by different malware detection technique, we could create a technique which is more robust because it would be looking at different characteristics of the same files. Such a combined approach could leverage the relative strengths of each of its components to yield a stronger overall detector. In our approach, we combined scores from Hidden Markov Model, Opcode Graph Similarity and Simple Substitution Distance techniques using Support Vector Machines as a classifier. It was found that Support Vector Machines was able to detect the given malware files from benign files by combining three different detection strategies into one.

In previous researches, it has been found that detection by these methods can be evaded by adding dead code. During our experiments, it was observed that Hidden Markov Model, Opcode Graph Similarity, Simple Substitution Distance start misclassifying at 10%, 60% and 80% block morphing respectively. Whereas we were able to use Support vector machine method to combine the results from the above three techniques at each morphing level and were still able to detect malware files from benign files even at 100% block morphing level. We were able to establish that SVMs produced more robust results and would only deteriorate in AUC after 100% morphing level, which was way better in comparison to all the underlying techniques.

The approach discussed in this paper can be used for all metamorphic malware files. During our research we also validated our approach against differing classes of metamorphic malware. We observed that Support Vector machine method was able to produce equivalent or better results for most of the malware families. We also noted that SVM generally tends to perform better when the AUC values for the underlying detection techniques are greater than 0.8, when AUC values start to decrease below 0.8, the performance of SVM method deteriorates.

Future work for this experiment can include enhancing our SVM technique even further. In our experiments we used statistical detection techniques such as Hidden Markov Model, Opcode Graph Similarity and Simple Substitution Distance as the underlying types for our method. In order to improve this method even further, we could incorporate some more detection techniques to our underlying detection strategies for Support Vector Machine classifier.

Finally, it could be really useful if we had a standard metamorphic malware dataset, so that we could compare the results of our proposed detection scheme, based on its performance against this standard data-set.

LIST OF REFERENCES

[1] S. Attaluri, S. McGhee, and M. Stamp. Profile hidden Markov models and metamorphic virus detection. *Journal in Computer Virology*, Volume 5, No 2, pp. 151–169. (2009)

[2] T. Austin, E. Filiol, S. Josse, M. Stamp. Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach. *Proceedings of 46th Hawaii International Conference on System Sciences.* (2013)

[3] J. Aycock, Computer Viruses and Malware. *Advances in Information Security*, Springer-Verlag, New York. (2006)

[4] A. P. Bradley. The use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Journal Pattern Recognition*, Volume 30, Issue 7, pp. 1145–1159. (1997)

[5] N. Cristianini and J. Shawe-Taylor. An Introduction to Support Vector Machines and other kernel-based learning methods, Cambridge University Press. (2000)

[6] Cygwin, Cygwin utility files. `http://www.cygwin.com/`

[7] E. Daoud, I. Jebril. Computer Virus Strategies and Detection Methods. *International Journal of Open Problems in Computer Science and Mathematics*, Volume 1, Issue 2. (2008)

[8] P. Deshpande, Metamorphic Detection Using Function Call Graph Analysis, Master's report, Department of Computer Science, San Jose State University. `http://scholarworks.sjsu.edu/etd_projects/336` (2013)

[9] T. Fawcett. An Introduction to ROC Analysis. `http://people.inf.elte.hu/kiss/13dwhdm/roc.pdf`

[10] A. Hii. Chi-squared distance and metamorphic detection, Master's report, Department of Computer Science, San Jose State University. `http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=7710&context=etd_theses` (2011)

[11] Introduction to Support Vector Machines. `http://fourier.eng.hmc.edu/e161/lectures/svm`

[12] G. Jacob, H. Debar, E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal of Computer Virology*, Volume 4, pp. 251–266. (2008)

[13] T. Jakobsen. A Fast Method for the Cryptanalysis of Substitution Ciphers. *Cryptologia*, Volume 19, pp. 265–274. (1995)

[14] A. Karnik, S. Goswami, and R. Guha, Detecting obfuscated viruses using cosine similarity analysis. *First Asia International Conference on Modelling & Simulation*, pp. 165–170. (2007)

[15] D. Lin, M. Stamp. Hunting for Undetectable Metamorphic Viruses. *Journal in Computer Virology*, Volume 7, Issue 3, pp. 201–214. (2011)

[16] Malware. `http://en.wikipedia.org/wiki/Malware`

[17] M. Musale. Hunting for Metamorphic JavaScript Malware, Master's report, Department of Computer Science, San Jose State University. `http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1357&context=etd_projects` (2014)

[18] A. Ng, Support Vector Machines. `http://cs229.stanford.edu/notes/cs229-notes3.pdf`

[19] OECD, Malicious software (malware): A security threat to the Internet economy. `http://www.oecd.org/dataoecd/53/34/40724457.pdf`

[20] M. Patel, Similarity tests for metamorphic virus detection, Master's report, Department of Computer Science, San Jose State University `http://www.cs.sjsu.edu/faculty/stamp/students/patel_mahim.pdf` (2011)

[21] S. Priyadarshi. Metamorphic Detection via Emulation, Master's report, Department of Computer Science, San Jose State University. `http://scholarworks.sjsu.edu/etd_projects/177` (2011)

[22] B. Rad, M. Masrom, S. Ibrahim. Camouflage in Malware: from Encryption to Metamorphism. *International Journal of Computer Science and Network Security*, Volume 12, Issue 8, pp. 74. (2012)

[23] N. Runwal, R. M. Low, and M. Stamp. Opcode Graph Similarity and Metamorphic Detection. *Journal in Computer Virology*, Volume 8, Issue 1–2, pp. 37–52. (2012)

[24] G. Shanmugam, R. Low, M. Stamp. Simple Substitution Distance and Metamorphic Detection, *Journal of Computer Virology and Hacking Techniques*, Volume 9, Issue 3, pp. 159–170. (2013)

[25] Snakebyte. Next Generation Virus Construction Kit (NGVCK). `http://vx.netlux.org/vx.php?id=tn02` (2000)

[26] M. Stamp. A Revealing Introduction to Hidden Markov Models. `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf` (2012)

[27] Support Vector Machine (LibSVM). `http://docs.rapidminer.com/studio/operators/modeling/classification_and_regression/svm/support_vector_machine_libsvm.html`

[28] Support Vector Machines (SVM) Introductory Overview. `http://www.statsoft.com/textbook/support-vector-machines`

[29] P. Szor. The Art of Computer Virus Research and Defense, Addison-Wesley Professional. (2005)

[30] A. Tang, S. Sethumadhavan, and S. Stolfo. Unsupervised Anomaly-based Malware Detection using Hardware Features. `http://www.cs.columbia.edu/~simha/preprint_raid14.pdf`

[31] VX Heavens. `http://vx.netlux.org/`

[32] Walenstein, R. Mathur, M. Chouchane, R. Chouchane, and A. Lakhotia. The Design Space of Metamorphic Malware. *In Proceedings of the 2nd International Conference on Information Warfare*. (2007)

[33] W. Wong and M. Stamp. Hunting for metamorphic engines, *Journal in Computer Virology*, Volume 2, No 3, pp. 211–229. (2006)

[34] P. Zbitskiy. Code Mutation Techniques by means of Formal Grammars and Automatons, *Journal in Computer Virology*, Volume 5, No 3, pp. 199–207. (2009)

APPENDIX

ROC Curves

A.1 Hidden Markov Model

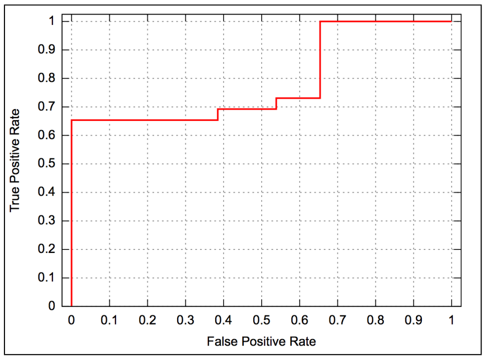


Figure A.32: ROC curve for Harebot - HMM

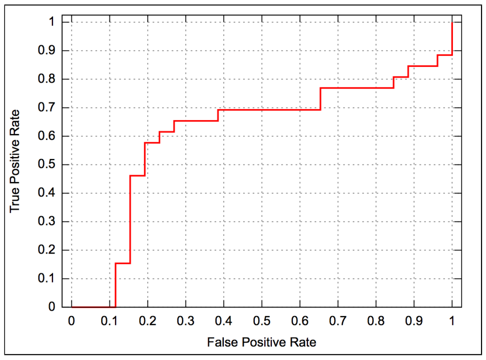


Figure A.33: ROC curve for Sesh - HMM

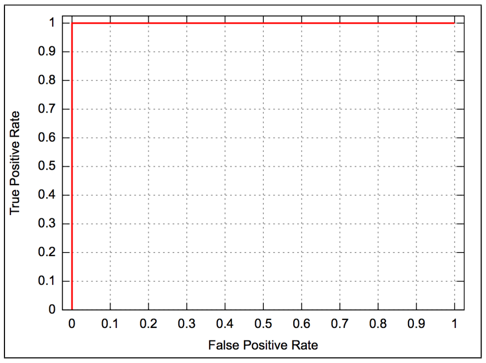


Figure A.34: ROC curve for SmartHDD - HMM

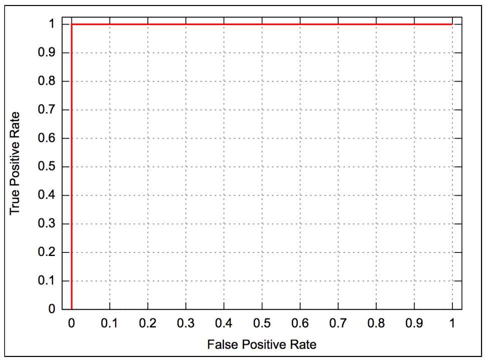


Figure A.35: ROC curve for WinWebSec - HMM

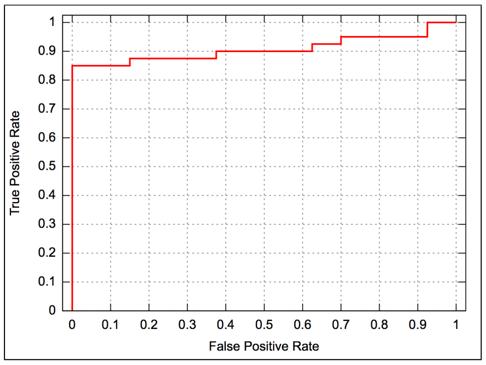


Figure A.36: ROC curve for Zbot - HMM

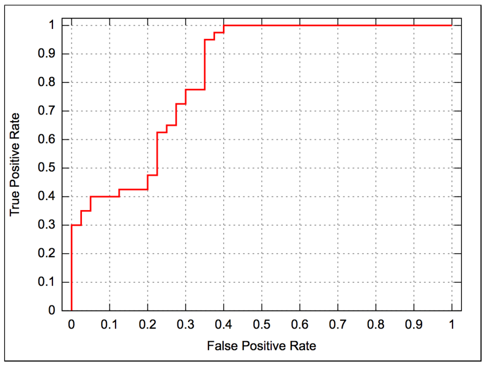


Figure A.37: ROC curve for ZeroAccess - HMM

A.2 Opcode Graph Similarity Method

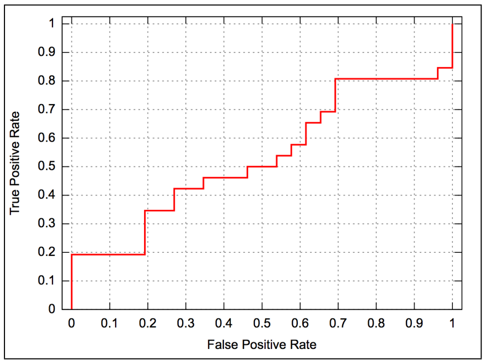


Figure A.38: ROC curve for Harebot - OGS

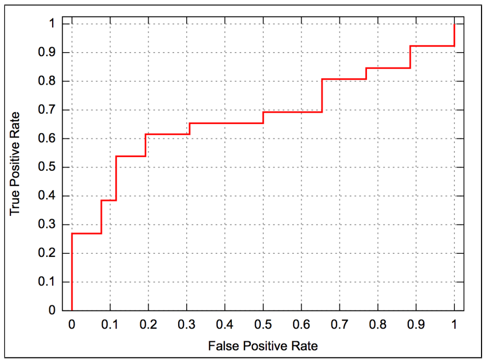


Figure A.39: ROC curve for Sesh - OGS

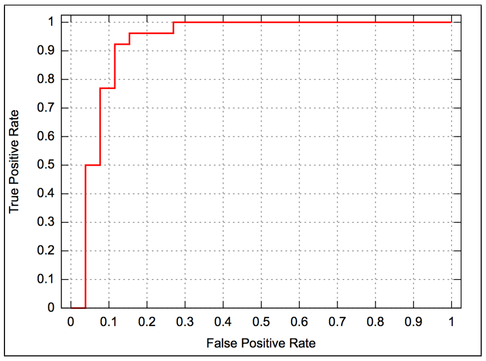


Figure A.40: ROC curve for SmartHDD - OGS

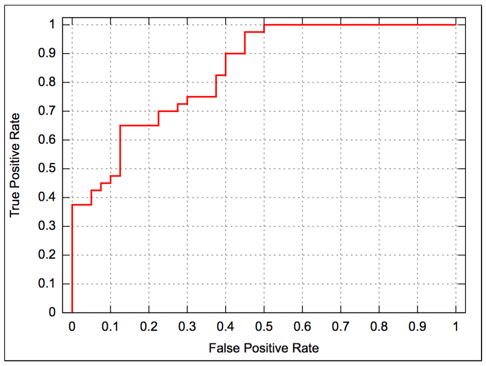


Figure A.41: ROC curve for WinWebSec - OGS

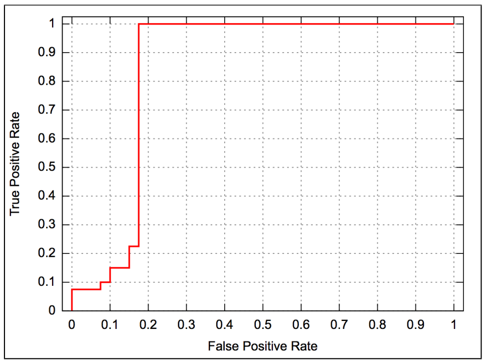


Figure A.42: ROC curve for Zbot - OGS

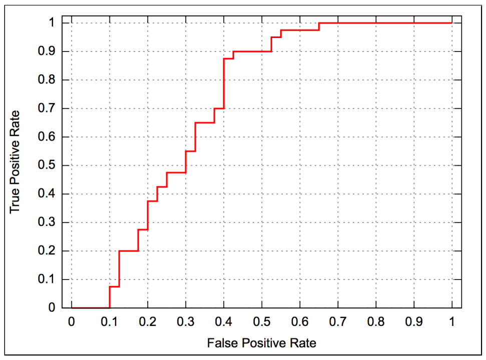


Figure A.43: ROC curve for ZeroAccess - OGS

A.3 Simple Substitution Distance Method

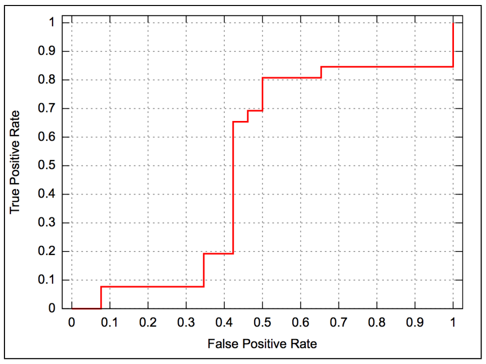


Figure A.44: ROC curve for Harebot - SS

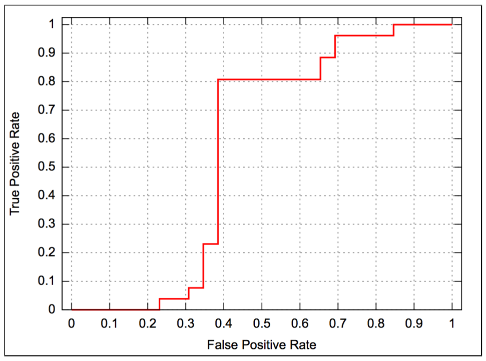


Figure A.45: ROC curve for Sesh - SS

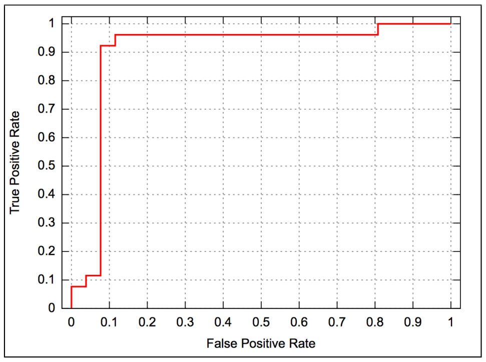


Figure A.46: ROC curve for SmartHDD - SS

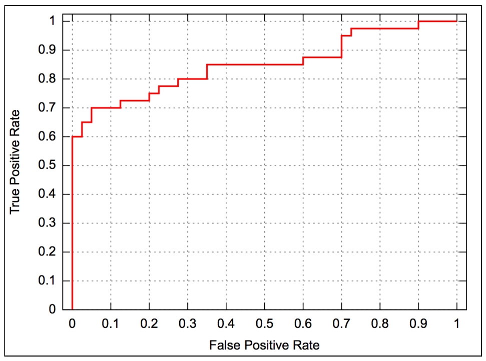


Figure A.47: ROC curve for WinWebSec - SS

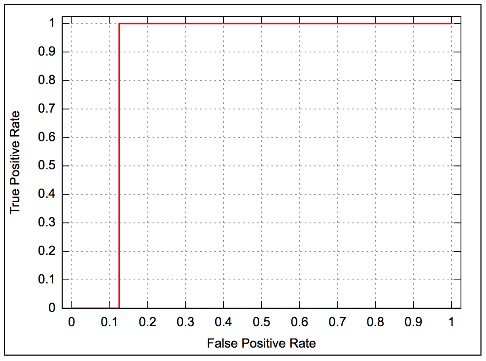


Figure A.48: ROC curve for Zbot - SS

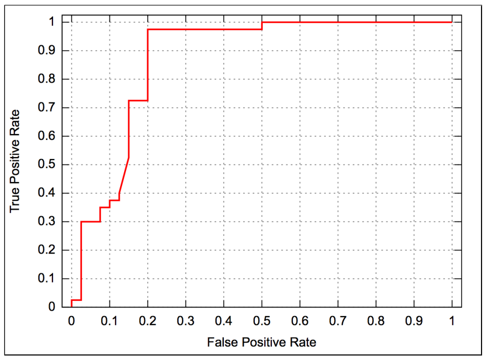


Figure A.49: ROC curve for ZeroAccess - SS

A.4 Support Vector Machine Method

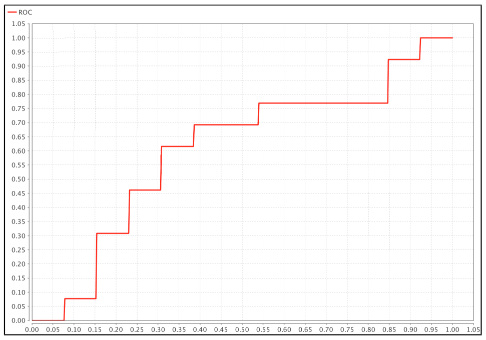


Figure A.50: ROC curve for Harebot - SVM

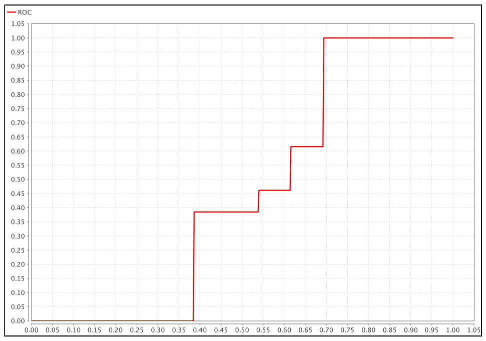


Figure A.51: ROC curve for Sesh - SVM

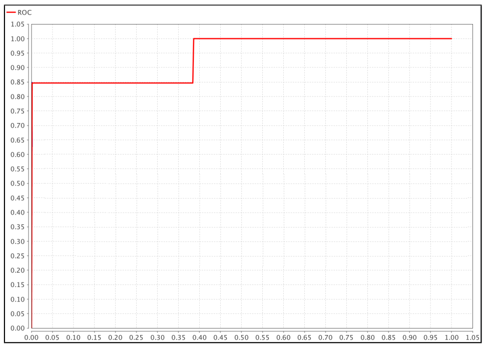


Figure A.52: ROC curve for SmartHDD - SVM

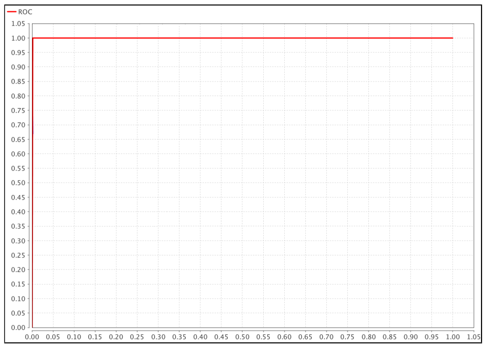


Figure A.53: ROC curve for WinWebSec - SVM

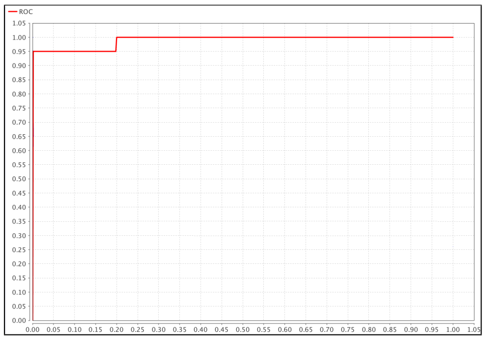


Figure A.54: ROC curve for Zbot - SVM

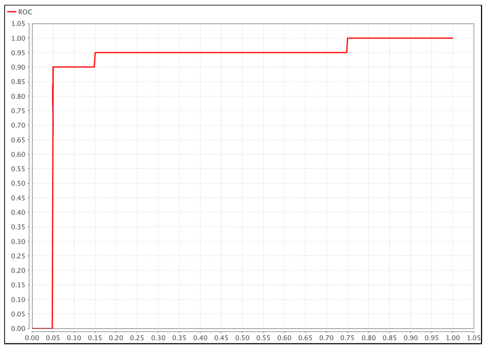


Figure A.55: ROC curve for ZeroAccess- SVM