Fall 2015

Measuring Malware Evolution

Poonkodi Ponnambalam
San Jose State University

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

Part of the [Computer Sciences Commons](#)

Recommended Citation

Ponnambalam, Poonkodi, "Measuring Malware Evolution" (2015). *Master's Projects*. 449.

DOI: <https://doi.org/10.31979/etd.kv4z-s6hz>

https://scholarworks.sjsu.edu/etd_projects/449

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

Measuring Malware Evolution

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

Poonkodi Ponnambalam

December 2015

© 2015

Poonkodi Ponnambalam

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Measuring Malware Evolution

by

Poonkodi Ponnambalam

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Dr. Mark Stamp Department of Computer Science

Dr. Robert Chun Department of Computer Science

Dr. Thomas Austin Department of Computer Science

ABSTRACT

Measuring Malware Evolution

by Poonkodi Ponnambalam

In this research, we simulate the effect of code evolution by applying a variety of code morphing strategies. Specifically, we consider code substitution, transposition, insertion, and deletion. We then analyze the effect of these code morphing strategies relative to a variety of malware scores that have been considered in previous research. Our goal is to gain a better understanding of the strengths and weaknesses of these various malware scoring techniques. This research should prove useful in designing more robust scores for detecting malware.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my mentor and advisor, Prof. Mark Stamp for his inspiration, guidance and support through out this Masters project. I also thank Dr. Thomas Austin and Dr. Robert Chun for being my committee members and their feedback.

I would like to thank my beloved husband, Vinodh Rajendran for his constant encouragement, support and patience through out my graduate studies.

Lastly, I would like to thank my parents for their moral support.

TABLE OF CONTENTS

CHAPTER

1	The Introduction	1
2	Background	3
2.1	Types of malware	3
2.1.1	Virus	3
2.1.2	Encrypted virus	3
2.1.3	Polymorphic virus	4
2.1.4	Metamorphic virus	4
2.1.5	Worm	4
2.1.6	Trojan	5
2.2	Obfuscation Techniques	5
2.2.1	Dead code insertion	5
2.2.2	Transposition	6
2.2.3	Instruction substitution	6
2.3	Malware Detection Techniques	6
2.3.1	Static analysis	6
2.3.2	Dynamic analysis	8
3	Similarity Techniques	9
3.1	Cosine Similarity	9
3.2	<i>N</i> -gram Similarity	10
3.3	Opcode Graph Similarity	12

3.4	Hidden Markov Model	14
3.5	Other Similarity Methods	18
4	Opcode Morphing Tool	20
4.1	Configuring the Morphing Tool	20
4.2	Deletion	21
4.3	Insertion	23
4.4	Substitution	23
4.5	Transposition	25
5	Experiments	28
5.1	System Configuration	28
5.2	Datasets	29
5.3	Malware Datasets	29
5.3.1	MWOR	30
5.3.2	Zeroaccess	30
5.3.3	Zbot	30
5.3.4	Winwebsec	30
5.3.5	Benign Datasets	31
5.4	Experiments Setup	31
5.4.1	Similarity between malware and morphed variants	33
5.4.2	Detecting malware after morphing	34
5.5	Executing Similarity Methods	35
5.6	Hidden Markov Model Similarity	36
5.7	Opcode Graph Similarity Computation	37

5.8	<i>N</i> -gram Similarity Computation	38
5.9	Cosine Similarity Computation	38
5.10	ROC Curves	39
5.10.1	Area under the curve	39
6	Observations	41
6.0.2	Note about reading figures	41
6.1	Deletion Morphing	42
6.2	Insertion Morphing	43
6.3	Substitution Morphing	44
6.4	Transposition Morphing	45
7	Conclusion and Future work	51
 APPENDIX		
A	Zeroaccess experiments	56
B	Zbot experiments	63
C	Winwebsec experiments	70
D	MWOR experiments with linux utilities	77
E	Zbot combination morphing experiments	83

LIST OF TABLES

1	Opcode sequence input for opcode graph	12
2	Opcode digram frequency graph	13
3	Opcode digram frequency probabilities	13
4	HMM notations	17
5	No of lines changed after transposition morphing	26
6	Change in file size after morphing	27
7	Malware datasets	29
8	Files used to train HMM	37
9	AUC for MWOR	46
A.10	AUC for Zeroaccess	58
B.11	AUC for Zbot	63
C.12	AUC for Winwebsec	76
D.13	AUC for MWOR	78

LIST OF FIGURES

1	Ngram comparison	11
2	Opcode graph transition probabilities	14
3	Opcode graph comparison	15
4	Hidden Markov Model - state diagram	15
5	Hidden Markov Model	16
6	Hidden Markov Model training and testing	18
7	Morphing tool architecture	21
8	Morphing tool deletion	22
9	Morphing tool insertion	24
10	Morphing tool transposition	26
11	Unique opcode count in malware files	31
12	Opcode distribution in malware families	32
13	Distribution of opcodes in cygwin files	33
14	Experiment setup 1 - Morph and compute similarity	34
15	Experiment setup 2 - Detect morphed malware	35
16	Example ROC curve	40
17	MWOR deletion experiments	47
18	MWOR insertion experiments	48
19	MWOR substitution experiments	49
20	MWOR transposition experiments	50
A.21	Cosine mean scores	56

A.22	Opcodegraph mean scores	56
A.23	N-gram mean scores	57
A.24	HMM mean scores	57
A.25	Zeroaccess deletion experiments	59
A.26	Zeroaccess insertion experiments	60
A.27	Zeroaccess substitution experiments	61
A.28	Zeroaccess transposition experiments	62
B.29	Zbot deletion experiments	64
B.30	Zbot insertion experiments	65
B.31	Zbot substitution experiments	66
B.32	Zbot transposition experiments	67
B.33	Zbot n-gram insertion	68
B.34	Zbot n-gram insertion	68
B.35	Zbot n-gram insertion	69
C.36	Cosine mean scores	70
C.37	Opcodegraph mean scores	70
C.38	Ngram mean scores	71
C.39	HMM mean scores	72
C.40	Winwebsec deletion experiments	72
C.41	Winwebsec insertion experiments	73
C.42	Winwebsec substitution experiments	74
C.43	Winwebsec transposition experiments	75
D.44	Distribution of opcodes in linux files	77

D.45	MWOR deletion experiments	79
D.46	MWOR insertion experiments	80
D.47	MWOR substitution experiments	81
D.48	MWOR transposition experiments	82
E.49	Deletion vs Substitution vs AUC for HMM	84
E.50	Deletion vs Transposition vs AUC for HMM	85
E.51	Insertion vs Deletion vs AUC for HMM	86
E.52	Substitution vs Transposition vs AUC for HMM	87
E.53	Insertion vs Transposition vs AUC for HMM	88
E.54	Insertion vs Substitution vs AUC for HMM	89
E.55	Deletion vs Substitution vs AUC for Cosine	90
E.56	Insertion vs Transposition vs AUC for Cosine	91
E.57	Deletion vs Transposition vs AUC for Cosine	92
E.58	Substitution vs Transposition vs AUC for Cosine	93
E.59	Insertion vs Deletion vs AUC for Cosine	94
E.60	Insertion vs Substitution vs AUC for Cosine	95
E.61	Insertion vs Substitution vs AUC for Opcodegraph	96
E.62	Deletion vs Substitution vs AUC for Opcodegraph	97
E.63	Insertion vs Transposition vs AUC for Opcodegraph	98
E.64	Deletion vs Transposition vs AUC for Opcodegraph	99
E.65	Substitution vs Transposition vs AUC for Opcodegraph	100
E.66	Insertion vs Deletion vs AUC for Opcodegraph	101
E.67	Insertion vs Substitution vs AUC for Ngram method	102

E.68	Deletion vs Substitution vs AUC for Ngram method	103
E.69	Insertion vs Transposition vs AUC for Ngram method	104
E.70	Deletion vs Transposition vs AUC for Ngram method	105
E.71	Substitution vs Transposition vs AUC for Ngram method	106
E.72	Insertion vs Deletion vs AUC for Ngram method	107

CHAPTER 1

The Introduction

Computer software is a set of instructions that is given to a computer to perform precise tasks. Each instruction is a specific command to the processor to do a certain operation, such as transform data from one form to another. In this era, software plays a role in virtually all aspects of life, from the financial markets on Wall Street, to controlling space stations, to protecting national security, to playing angry birds on a smart phone. Given that this is the case, it is not surprising that software is a prime target of criminals who want to access user data, or otherwise profit from their illicit activities.

Any malicious code that is designed to damage a computer systems is known as malware. Malware writers regularly introduce new features and techniques designed to enable their malware to evade detection. Typically, existing malware is modified to make new malware, rather than starting over from scratch [36]. Thus, malware can be viewed as evolving over time. As in any software development, it is easier to reuse existing code than begin with a blank slate. This is why malware writers tend to reuse previous malware code extensively and release new variants of the same malware [1].

It is important to understand malware evolution and identify the relationship among different malware samples. Understanding malware evolution will certainly help us to better understand the connections between existing malware families and might help us to detect new, zero-day malware.

This paper is organized as follows. Chapter 2, discusses background information

about malware, malware types, malware analysis and malware detection. In Chapter 3, we present background information on various similarity techniques used for malware detection and analysis. Chapter 4 considers how morphing is used to create different variants of a malware and how to simulate malware evolution based on such morphing techniques. Details on our experiments are given in Chapter 5 and Chapter 6 provides some analysis of these experimental results. Chapter 7 contains our conclusions and discussion of future work.

CHAPTER 2

Background

In this chapter, we discuss about the malware classification and details on malware analysis and detection. Malware [9] is a general term used to refer any software or program doing malicious or suspicious activity. Based on the malware behavior, their replication nature and the infection mechanism, they are classified into many types. They can be detected either from the way they are built (through static analysis) or the way they function or behave (through dynamic analysis). The following sections cover the different types of malware and their detection techniques.

2.1 Types of malware

2.1.1 Virus

Computer viruses are similar to biological viruses. When executed, the viruses copy themselves and spread through out the system. They are self-replicating [1] and they do not spread through networks. The primary way the virus spreads to other systems is through human interactions. When humans, intentionally or unintentionally, copy virus to a compact disk or a USB drive from an infected system to a new machine.

2.1.2 Encrypted virus

Encrypted viruses [1, 7] are little more sophisticated form of viruses that use encryption to avoid detection. In encrypted form the virus is not executable. It has to be decrypted before it can be executed. The actual encryption techniques that are used are very weak. They are simple obfuscation techniques to avoid easy detection

and the encrypted virus body remains the same after each infection. Advanced virus scanners can easily detect this type of virus.

2.1.3 Polymorphic virus

Polymorphic virus [1, 7] are very advanced and complex type of encrypted viruses. They have several different decryptor loops that change after each infection. Some known polymorphic viruses like Tremor [1] has over 6 billion decryptor loops that change after every infection. This encrypted virus will be decrypted at runtime and these viruses may or may not carry the morphing engine. Detecting polymorphic virus is very challenging as they change their decryptor after each infection.

2.1.4 Metamorphic virus

Metamorphic virus [1] is an advanced form of viruses that change their body after each infection. These viruses are not encrypted. So, instead of a decryptor loop, they change, i.e. mutate after every new infection. These viruses have a mutation engine that changes the body upon infection. There are known powerful metamorphic viruses like Simile's mutation engine [1]. The whole virus is less than 12000 lines and it can exhibit metamorphism.

2.1.5 Worm

Worms are very similar to viruses except that they propagate through networks [1, 7]. They are capable of self-replication and it replicates to the infected remote computers and executes self. Stuxnet [34] is a popular worm targeted on industrial control systems such as gas pipeline or power plant. The attacker modifies the code of programmable logic controllers (PLC's) to their intent and hide it from

the equipment operator. It is designed to damage the facilities by taking control of the systems.

2.1.6 Trojan

A Trojan horse [1, 7] pretends to be a benign program but it does malicious activity in the backend. They do not self replicate like worms and viruses. Most popular ones are the password-grabbing programs. Citadel [32], Zbot [31], Zeroaccess [33] are popular trojan malware and they are used to get confidential information from the unsuspecting victims.

2.2 Obfuscation Techniques

This section discusses the different techniques used by hackers to evade standard malware detection methods.

2.2.1 Dead code insertion

Malware writers frequently add junk code to the malware that never gets executed. This is a very simple obfuscation technique to evade detection. These inserted lines of code do not change the behavior of the program. Their primary purpose is to change the way the malware looks in signature and opcode distribution. These kind of added instructions are otherwise called as “Garbage Instructions”. Sometimes they add code that may occasionally get executed but it does not really affect the functionality of the program.

2.2.2 Transposition

Blocks of code can be replaced to a different place in the same program to alter the structure of the program. This can be either done by inserting `jmp` instructions in between to make sure the execution path is not altered. Too many `jmp` instructions in the program is a good heuristic that the code is been transposed a lot. The other way to achieve transposition is subroutine transposition. Subroutines are moved to different places in the program. For N subroutines in a program, $N!$ subroutine permutations are possible. This cannot be easily detected compared to transposition using `jmp` instructions.

2.2.3 Instruction substitution

Instructions in the program are replaced with the equivalent instructions and as a result the structure of the program can be easily altered. Since the instructions are replaced with the equivalent instructions, the functionality of the program is not changed. In order to substitute opcodes with the equivalent opcodes, the malware writer should have a good dictionary of opcodes and their equivalent opcodes.

2.3 Malware Detection Techniques

In this age of the Internet, malware detection is a multi-billion dollar industry. There are several public companies like Symantec, McAfee, FireEye etc. that offer commercial malware protection solutions for users.

2.3.1 Static analysis

In static analysis [1, 7], we use the architecture and the syntactic structure of the program. The emphasis is on the actual opcodes that make the program. The

program is disassembled using tools like IDA Pro [5] to get the asm/opcodes of the program that are used in analysis.

Signature based detection

Signature based detection [21] is the most popular method used by the anti-virus companies to detect work based on the signature scanning. A signature is a known pattern detected in the malware. When signatures of all known malware are indexed in a malware signature database, they can be used to check new malware. This scanning is done on-demand and it is very efficient. This is relatively easy and fastest method to do the malware detection and this can scale well with the increasing number of malware. When new malware is detected, we just add the signature to the database. The disadvantage of this method is that signature scanning [1] can be easily defeated by obfuscating the program and altering the signature. Metamorphic and polymorphic malware cannot be detected using signature based scanning. Another drawback is that this works only for known malware signatures. This cannot detect any new malware. For a malware writer, this is easy to beat. Every time a new malware variant is created, they can modify the signature and release it.

Static heuristics based detection

Instead of limiting the scan to only signatures, this method looks for code that are “virus-like”. Heuristics like, finding junk code, use of too many `jmp` instructions, decryptor loops, self-morphing code, decryptor loops and undocumented APIs are red-flags that can be used to detect malware [1]. These heuristic based method can be used to detect unknown malware as well.

2.3.2 Dynamic analysis

Behavior monitor

Here we let the program to run in a controlled environment and observe the actions. If actions are very suspicious the user will be asked to terminate the program. Like static heuristics, here we look for dynamic heuristic to see if the program is a malicious program.

Emulation

The malware is executed in a safe, emulated environment [1, 17] and the execution of the malware is carefully monitored. Since the malware is not run on an actual machine, this is safe. The junk codes are not executed and we can trace the execution path of the malware. Thus the dynamic analysis is based on the execution of the malware and so the layer of obfuscation is removed. Some malware are very intelligent and they stop executing in an emulated environment, if it detects it is run in an emulated environment. Some malware might alter their execution path to evade malware detection and pretend to be doing benign activity. These kind of malware can make the dynamic malware detection really difficult too. Though this is a robust technique to detect malware, it is time consuming and it does not scale well with the number of malware we deal every day in the common world. Modern anti-virus companies are focussing more on dynamic analysis to make it more scalable.

CHAPTER 3

Similarity Techniques

Similarity techniques are widely used to compare malware. Malware can be compared among themselves to identify how close they are. These comparisons greatly help to identify commonality between malware. This information is useful in identifying similar malware or the variants of the malware.

3.1 Cosine Similarity

The cosine similarity [13] is one of the most popular methods for computing the distance or similarity between two vectors. It is a measure of a cosine of the angle between them. The value of cosine distance can range from 1.0 for vectors with same orientation and 0.0 for vectors at 90°. Vectors that point in opposite directions have a similarity of -1.0. For practical purposes, if vectors are in positive space, the cosine distance is bounded in [0.0, 1.0]. It is important to note that the cosine similarity measures only the direction, not the magnitude. Given two vectors, the cosine similarity can be computed as given in the formula

$$\cos(a, b) = \frac{a \cdot b}{\sqrt{|a|^2 |b|^2}}$$

Cosine similarity has been used successfully in information retrieval and malware detection. In [35], they use cosine to build similarity between query and documents for a search engine. In [36], cosine has been shown to capture the similarity in malware successfully.

Given the malware A and malware B , the malware is disassembled and their opcodes are extracted. Frequency of the opcodes in each file is calculated and this

becomes the term frequency vector v for the file. Vectors of malware A and B , v_1 and v_2 respectively are obtained. Then the cosine distance between the malware vectors v_1 and v_2 is calculated using the cosine similarity formula.

3.2 N -gram Similarity

N -gram similarity method [38] is a novel technique to compute the similarity between different programs. Given two programs A and B and their opcodes, we want to compute the similarity between them. To extract the opcodes, we ignore and skip the labels, directives and new lines. A and B may have different length of opcode sequences. Let n be the opcode length of A and m be the opcode length of B .

Find the set of unique opcodes and assign a unique number to each opcode. For the opcodes in A and B , build subsequences of 3 consecutive opcodes. This results in $O(m)$ triples for A and $O(n)$ triples for B . While comparing the triples, the order in which they occur does not matter. So, it is advisable to sort the 3 opcodes in a triple in some deterministic order.

Build a boolean 2D matrix of dimensions mn . Compare triples in A with triples in B . If i th triple in A matches j th triple in B , mark the coordinate (i, j) as a match in the matrix. This results in the matrix where the matching indices are true. This complexity of this operation is $O(mn)$.

The N -gram comparison of different types of morphing is shown in the Figure ?? . For identical files, we have a diagonal line. For insertion and deletion morphing, the section of code that is inserted or deleted does not match the base version. For transposition morphing, the moved code will still match in a different region.

Figure 1: Ngram comparison

To eliminate noise resulting from random matches, we remove all matches whose length is less than a chosen threshold of 5. Note that, based on the implementation this can be a complex operation. For efficiency reasons, this is optional.

In the above steps, we find sequential matches between all opcodes of A and B . If the opcode files are identical, $A == B$, then $n = m$. The matrix will then be a square matrix, where only the main diagonal is true. If the line appears away from the diagonal, then the matching opcodes appear in different locations in A and B .

To compute the score, for each of the files, A and B , count the number of opcodes

covered by the matching line segments (one or more). Normalize this number by the total number of opcodes m and n respectively. This results in the ratio of opcodes in A that match opcodes in B . The n -gram similarity is the average of these two ratios, that is,

$$\text{File score} = \frac{\text{Count of matchset in file}}{\text{Total number of opcodes}}$$

and

$$\text{Total } n\text{-gram score} = \frac{\text{Score for file } A + \text{Score for file } B}{2}$$

3.3 Opcode Graph Similarity

This is a graph based similarity technique used to detect similarity between directed graphs [19]. A graph is a structure which consists of set of vertices V and edges E . A graph can be weighted graph or unweighted graph.

Opcodes are extracted from the assembly code of the malware and a list of opcodes are obtained. A graph is constructed with opcodes as vertices and possible transitions as edges. Edge weights are the probabilities of transitions. The resulting graph is a digram statistics graph. The constructed graph becomes the signature for the malware.

To illustrate opcode graph similarity, let us consider an example list of opcodes as shown in the Table 1.

mov	mov	sub	mul	jmp	cmp
cmp	and	cmp	jmp	mov	lea

Table 1: Opcode sequence input for opcode graph

In the example opcode `mov` occurs after `lea` and so there is a transition from `mov` to `lea`. Opcode `add` occurs after `lea`, etc. Using these transitions a matrix of the

transition frequency can be formed, as in Table 2. In this opcode graph, the opcodes are the nodes, weighted edges are the transition counts.

	mov	lea	and	jmp	sub	cmp	mul
mov	1	1	0	0	1	0	0
lea	0	0	0	0	0	0	0
and	0	0	0	0	0	1	0
jmp	1	0	0	0	0	1	0
sub	0	0	0	0	0	0	1
cmp	0	0	1	1	0	1	0
mul	0	0	0	1	0	0	0

Table 2: Opcode digram frequency graph

The digram probabilities is computed from digram frequency count. Each value in the row is divided by the sum of the values in the row. Table 3 shows the transition probabilities among the opcodes.

	mov	lea	and	jmp	sub	cmp	mul
mov	$1/3$	$1/3$	0	0	$1/3$	0	0
lea	0	0	0	0	0	0	0
and	0	0	0	0	0	1	0
jmp	$1/2$	0	0	0	0	$1/2$	0
sub	0	0	0	0	0	0	1
cmp	0	0	$1/3$	$1/3$	0	$1/3$	0
mul	0	0	0	1	0	0	0

Table 3: Opcode digram frequency probabilities

If N is the number of distinct opcodes in the all the files, a graph of matrix $N \times N$ is constructed.

Each malware file is converted to a opcode graph of $N \times N$. Let A be opcode graph of the malware base file a_{ij} and B be the opcode graph of the malware file b_{ij} .

The opcode graphs A and B are compared [19] and the score is computed as

Figure 2: Opcode graph transition probabilities

given in the Equation 1.

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

If both the files are the identical we obtain 0 score. More the files are similar they score less.

3.4 Hidden Markov Model

The Hidden Markov Model [11, 18] is one of the most popular machine learning technique that works well on modelling sequences. It has been successfully used

Figure 3: Opcode graph comparison

in speech recognition [11], speech synthesis, parts-of-speech tagging [15], alignment of bio-sequences , time series analysis, etc. In recent years, HMM has been used extensively in malware analysis [2, 3] and metamorphic virus detection.

The Markov property is that, all of the information required to compute the future states depends only on the present state, not on any of the previous events preceding the current state. In a conventional Markov model, the states are observable, where as in a hidden markov model, the states are latent (or hidden) and they emit observations that are visible.

Figure 4: Hidden Markov Model - state diagram

The transition probability matrix is 2×2 matrix. A_{ij} is the probability to transition from state s_i to state s_j .

$$A = \begin{pmatrix} 2/5 & 3/5 \\ 2/3 & 1/3 \end{pmatrix}$$

The emission probability is a 2×3 matrix for 3 opcodes. $B_j(y_k)$ is the probability to emit symbol y_k in state s_j .

$$B = \begin{pmatrix} 1/2 & 1/3 \\ 1/4 & 1/2 \\ 1/4 & 1/6 \end{pmatrix}$$

In malware analysis, we consider the opcode sequences as the observations and assume that they are emitted by hidden states in the HMM. For these experiments we use 2 state HMM. Each state has a emission probability of emitting one of the N opcodes and a transition probability for all the possible states.

Figure 5: Hidden Markov Model

T	=	length of the observation sequence
N	=	number of states in the model
M	=	number of observation symbols
A	=	state transition probabilities
B	=	observation probability matrix
π	=	initial state distribution
O	=	$(O_0, O_1, \dots, O_{T-1})$ = observation sequence.
X	=	$(X_0, X_1, \dots, X_{T-1})$ = hidden state sequence for the observations

Table 4: HMM notations

There are 3 problems [2, 18] which can be solved using HMM. For these problems, we use the HMM notation as described in Table 4.

Problem 1

Given the model $\lambda = (A, B, \pi)$ and the observation sequence O , find the probability of the observation sequence being generated by the model.

Problem 2

Given the model $\lambda = (A, B, \pi)$ and the observation sequence O , find the optimal state sequence taken by the HMM to generate the observed Sequence.

Problem 3

Given an observation sequence O , number of states N and the number of observations M , generate the model $\lambda = (A, B, \pi)$ that has the maximum likelihood of generating the observation sequence. This is the training part of HMM. We use the Baum-Welch algorithm [18, 26] to train a HMM using expectation-maximization.

Here in the experiments, solving problem 3 is the first step, which is the training

Figure 6: Hidden Markov Model training and testing

part. First the model is generated with the list of opcodes from the base or original malware as shown in the Figure 6. Solving problem 1 is the second step, testing part. The morphed variants are scored against the generated model. The scores tell us how similar the original and morphed malware families are.

The opcode sequences are extracted from the malware and HMM is trained with the resultant observation sequences. HMM captures the properties of the malware from the given observation sequence and this serves as the signature for the malware. Given any new malware observation sequence, HMM can tell us how close the malware are.

3.5 Other Similarity Methods

There are several other distance based similarity methods that have been used in the literature. Toderici et al [30] have used Chi-squared distance to identify metamorphic virus detection. Baysa et al [4] have used structural entropy for malware identification. Simple substitution distance [24] is another popular method used to

compute similarity between two programs. Profile HMM [2] is a variant of Hidden Markov Model has shown good results for malware detection. Deshpande et al [8] have used Eigen value analysis for malware detection. Singular value decomposition (SVD) [12] is another method successfully applied for malware analysis and detection.

In addition to the classical methods, there are other machine learning methods that have also been used for malware analysis. Support vector machines (SVM) [28], k -means [29], logistic regression, naive bayes, neural networks [23] have been successfully used to detect malware.

CHAPTER 4

Opcode Morphing Tool

Malware writers use several different techniques to create a new variant of a malware. They add more features to make the malware more powerful and not easy to detect. The existing malware can be made powerful by adding more features to the malware and this in turn will increase or decrease the number of opcodes in the instruction set. They can delete the easily detectable features to avoid detection and to make it robust. They can also replace the existing opcodes with equivalent opcodes. Subroutines are easy to move around to avoid detection. All of these cases occur as the malware evolves over time. Here we simulate malware evolution by creating its derived variants by insertion, deletion, substitution and transposition of opcodes.

Malware executables are first disassembled using disassembler like IDA pro [5] and their respective assembly files are extracted. It is then easy to extract opcodes from the assembly files. This opcode list is fed as the input to the morphing engine to produce the morphed variants. The morphing engine takes two inputs - the malware opcode file and the morphing ratio configurations to generate the morphed derivatives of the malware. The input morphing ratios guide the morphing engine to trigger morphing routines at appropriate locations in the input malware. The Figure 7 shows the details of the morphing tool.

4.1 Configuring the Morphing Tool

The user inputs the morphing ratio for insertion, substitution, transposition and deletion of opcodes. For the given malware, the tool takes the configuration input containing the morphing ratios for deletion, insertion, substitution and transposition

Figure 7: Morphing tool architecture

to generate the morphed variants.

The opcode list of the malware can be morphed by any of the four approaches or the combination of approaches. The details of the actual morphing functions are discussed in the following sections.

4.2 Deletion

In the deletion phase, the morphing tool deletes opcodes from the input malware. The number of locations at which this deletion happens is a function of the `deletion_morphing_ratio`. If the ratio is 0.1, then we intend to delete opcodes at 10% of the locations in the input. We generate a uniform random number between 0.0, 1.0. If the random number is less than the threshold, 0.1, then we trigger the

deletion phase. At each deletion location, we have a choice of deleting any number of lines. To keep the result of the morphing realistic, we delete a random number (0 to 10) of lines. So, in some locations we might be deleting 1 or 3 opcodes and there might be places where 10 opcodes are deleted. Deleting 10 lines of opcodes from a opcode list is similar to deleting a block of code or subroutine from the program. The Figure 8 shows the details of the deletion phase.

Figure 8: Morphing tool deletion

The morphing engine makes sure that each line of the malware is equally likely to be morphed. In the malware deletion routine, we delete a random number of opcodes. So, on average the deletion routine will trigger for 10% of the opcodes and we delete one or more (up to 10) opcodes for each morph. As a result, the number of opcodes in the output morphed version of a 10% deletion can be lesser than 90% of the input opcodes. On average, a 10% deletion results in 38% reduction of opcodes in total opcodes of the resulting malware.

4.3 Insertion

In the insertion phase, we inject new code and simulate addition of new features in the malware. Similar to the deletion phase, the number of locations where insertion occurs depends on the `insertion_morphing_ratio`. If the ratio is 0.2, then we add code at 20% of the places in the input malware. To keep the injection random, we generate a random number between 0.0 to 1.0 and trigger the insertion phase when it is lesser than the threshold.

The opcodeset from which the opcodes are inserted, can be curated by multiple means. One can generate the distribution of the input opcodes and sample from that distribution. This will keep the ratio of opcodes the same as the input malware. Another way is to sample opcodes randomly from a benign file [28]. In this implementation, we choose the inserted opcodes from a set of all unique opcodes in the malware family. We assign equal probability for all opcodes to be inserted. The Figure 9 shows an example of morphing through insertion.

The number of opcodes inserted at each location is also random. We can insert any number of opcodes from 1 to 10 at each location. If the malware writers change an *if* statement, then it will result in 2-3 opcode change. If they insert a full subroutine, then it may result in a 10 opcode change. This experiment assumes that the malware writers use a combination of approaches to generate the new variant.

4.4 Substitution

In the substitution phase, we substitute some opcodes with other opcodes. Substitution is a popular obfuscation technique because, it is fairly easy to identify equivalent set of opcodes and substitute in the malware. For example, `inc` can be done with `add`, `sub` can be replaced with `dec` and vice versa.

Figure 9: Morphing tool insertion

To morph the malware with substitution, we replace a randomly chosen opcode, with randomly chosen opcodes from unique opcode list. Each opcode in the list has an equal probability to be chosen. The morphing tool generates a random number to decides whether the opcode needs to be morphed. If the opcode needs to be morphed, then it replaces the opcode with the set of opcodes of unique list.

At each substitution point, we remove the existing opcode and replace with up to 3 opcodes in its place. Instruction substitution might increase the number of opcodes if we replace one opcode with many opcodes to do equivalent operation. Substitution can also be viewed as deletion of an opcode and insertion of new opcodes in the place of deleted opcode.

4.5 Transposition

Transposition relocates a block of code to a different place in the file without changing the execution state of the program. It only alters the structure of the program. It can be achieved in many ways and the following ways are used to transpose the code. Instruction transposition is another easy to implement method of obfuscation. The malware writers frequently move the subroutines in the source file. If there are N subroutines, there are $N!$ permutations of moving them to different parts of the file.

There are two primary ways of implementing transposition, which are

- Adding `jmp` instructions to make sure the control flow is not lost.
- Transposing only subroutines. Here the subroutines are relocated to different places in the program and there is no need to insert `jmp` instructions.

In the Figure 10 the original list of opcodes are relocated to a different places and the flow of execution is not altered by adding appropriate `jmp` instructions between the labels. In this implementation, the morphing engine currently does transposition by moving subroutines. If the `transposition_morphing_ratio` is 0.001, we transpose code 1 in 1000 of the locations. At each morphing location, we choose a block of code upto 10 lines long. This block is moved n lines. In our case n is also random and can vary between 1 and 10.

Transposing opcodes by adding `jmp` instructions increases the number of `jmp` opcodes in the program and too many `jmp` instructions in the code is a good heuristic to detect that it is a malware. One can easily reverse engineer the code and remove the `jmp` instructions and reorder the labels to get the original code back.

Figure 10: Morphing tool transposition

Morphing Ratio	No of lines changed
10%	52%
20%	71%
30%	81%
40%	87%

Table 5: No of lines changed after transposition morphing

To estimate the amount of opcodes actually changed by the morphing techniques discussed above, we compare the line count of the input and the line count of morphed files here. The Table 6 shows the resulting size in the morphed file, when compared to the input malware. When deletion is done at 10%, we actually delete 38% of the opcodes and the resulting file size is only 62.07% of the input file size. We notice that the during transposition, the file size does not change much. The small changes we notice here are because of **jmp** instructions.

Morphing Ratio	Deletion	Insertion	Substitution	Transposition
10%	62.07%	155.19%	110.03%	100%
20%	42.23%	209.53%	120.15%	100%
30%	29.86%	265.00%	129.93%	100%
40%	21.44%	319.88%	140.05%	100%

Table 6: Change in file size after morphing

CHAPTER 5

Experiments

This chapter discusses the various experiments that are done to compare and analyse malware variants.

The input for the experiments is a list of opcodes, extracted from malware samples. These are treated as the base opcodes for that malware family. These base opcodes are morphed using morphing engine, described in Chapter 4 to generate several morphed derivatives of the malware. For the sake of the following experiments, I used morphing ratios of 10%, 20%, 30%, 40%.

These morphed outputs are compared and analyzed with several similarity methods discussed in Chapter 3. The similarity methods such as cosine similarity, n -gram analysis, HMM and opcode graph similarity are used to detect the similarity between base malware files and morphed malware variants. We should note that opcode graph similarity is a graph based score [19], HMM [2] is a statistical-based score.

5.1 System Configuration

The experiments were done in a system with the following configurations.

- Intel Core-i7 processor, with 8 cores
- 8 GB ram
- Ubuntu 14.04 operating system
- The ROC plots were generated in Gnuplot 4.6

- The box plots were generated in R version 3.0.2.

5.2 Datasets

We used different malware families in our experiments. Section 5.3 covers the details of the malware families used in the experiments and the Section 5.3.5 covers the benign files used for different malware families. Opcodes are then extracted from the assembly code of the malware file. We then extract the list of opcodes from assembly code. For all testing, training, computation the extracted list of opcodes is the input. All the opcodes are extracted statically and the list of opcodes we extracted are based on the syntactic structure of the program.

5.3 Malware Datasets

The list of malware families used in the experiments can be found in the Table 7. As discussed earlier the opcodes are extracted from the malware families and used as an input. Different malware families are morphed using morphing tool to simulate code evolution. Each variant is a new malware and we need to see how well the similarity techniques work on the variants after applying morphing. The Figure 12 shows the distribution of opcodes in all malware families.

Malware Families	No of files
WinwebSec	5819
Zbot	2168
Zeroaccess	1306
MWOR	100

Table 7: Malware datasets

5.3.1 MWOR

MWOR stands for Metamorphic Worms. They change their form (i.e) their internal structure [6, 25] but functionally they remain the same. This is used by malware writers to evade malware detection based on signatures. More details on metamorphism can be found in Chapter 2.

5.3.2 Zeroaccess

Zeroaccess [33] is a trojan designed to target the windows operating system. It downloads the malware in the compromised machine and they use rootkit techniques to remain invisible. Their file system is invisible in the compromised system [33] that helps to download the malware.

5.3.3 Zbot

Zbot [31] is a trojan which infects and compromises the system [31]. It steals the confidential information from the system. It is mainly targeted to steal the banking information and other online credentials.

5.3.4 Winwebsec

Winwebsec [37] is a also a Trojan, targeted on windows operating systems. It disguises as an anti-virus software and prompts false messages that the user's system is infected. Some users thinks they are legitimate software and pay for the software to remove the malware threats.

The Figure 11 shows unique opcode count in the malware families. Zeroaccess has more number of opcodes compared to the other malwares.

Figure 11: Unique opcode count in malware files

5.3.5 Benign Datasets

In order to compare and analyse the malware datasets, we need non-malware files to control the experiments. I used the common executables as benign files in the experiments.

MWOR is a unix-based malware whereas winwebsec, zeroaccess and zbot are targeted on windows operating system. For unix malware, unix benign files such as `sort`, `uniq` etc. are used for testing. Other windows malware are tested with window cygwin utility files. The Figure 13 shows the distribution of opcodes in benign cygwin utility files. We notice that the opcode `mov` occurs far more frequently than all other opcodes.

5.4 Experiments Setup

We discuss the setup of all the experiments done in this section. There are two different types of experiments.

Figure 12: Opcode distribution in malware families

Figure 13: Distribution of opcodes in cygwin files

5.4.1 Similarity between malware and morphed variants

In this experiment, we try to show the effect of morphing on the similarity scores. The morphing and similarity computation is shown in the Figure 14. The steps to compute the similarity between the malware and morphed variants are

- Extract opcodes from the malware. Let us call this *base* opcodes.
- Generate morphed variants using the morphing tool described in Chapter 4. In these experiments, we generate 16 different morphed variants for insertion, deletion, substitution and transposition morphing each with different morphing ratios (10%, 20%, 30%, 40%).
- Compute similarity scores, comparing each of the morphed variants with the base malware. We use four different similarity methods - HMM, cosine similar-

Figure 14: Experiment setup 1 - Morph and compute similarity

ity, opcode graph similarity and n -gram similarity as discussed in Chapter 3. The details on training the HMM and other methods are discussed in the next Section 5.5

- Generate box plots from the resulting scores.

5.4.2 Detecting malware after morphing

In a real world scenario, if someone generates a new malware family by morphing an existing family using any available morphing technique, we would like to detect the malware. This experiment explores how well we are able to detect these morphed malware by using the scoring methods discussed in Chapter 3.

The Figure 15 illustrates the experiment setup.

- Extract opcodes from the malware. Let us call this *base* opcodes.

Figure 15: Experiment setup 2 - Detect morphed malware

- Generate morphed variants using the morphing tool.
- Use different comparison methods to detect the malware as opposed to the benign files chosen in Section 5.3.5.
- From the generated malware and benign scores, generate ROC curve. The details of ROC curve will be discussed in Section 5.10.

5.5 Executing Similarity Methods

In our experiments we used similarity methods such as hidden markov model, opcode graph similarity, n -gram similarity and cosine similarity to detect the similarity between the malware and their variants. Note that hidden markov model and opcode graph similarity captures the statistical properties of the malware.

5.6 Hidden Markov Model Similarity

To compare the malware derivatives using HMM, the experiment set up is described here. For a specific malware family, say MWOR, we wish to compare the base malware to different morphed versions like del_0.1, del_0.2 etc.

We take opcode sequences of base malware and concatenate them. This is the observation sequence on which the model is trained. Generate HMM training data from all of the input malware samples. The amount of training data required to train a HMM depends on the number of parameters we estimate. In our case, the number of estimated parameters is $N + N^2 + NM$. So a few hundred thousand observations is enough to train a HMM model, for a two state HMM and a few hundred unique opcodes.

The HMM is configured to be two states. This is based on prior research [38] that has shown that two state HMM is capable of detecting malware. Train the HMM using the unsupervised Baum-Welch [2, 18] method discussed in Chapter 3. Then the HMM is trained with the observation sequences for sufficient number of iterations. Here we used 1000 iterations and we noticed that the model converges well before 500 iterations.

The trained HMM model is then used to score the test files of the different morphed versions. The result is normalized with the length of the morphed file to get normalized score. We tested the all the four variants morphed with deletion and generated a box plot to show their scores. The box plot gives the range of the scores, the upper bound and lower bound of the scores. Like wise, the comparisons are done for the other types of variants such as insertion variants, substitution variants and transposition variants.

In order to test how morphed families can be detected using HMM, HMM is trained for each malware variant family. Here, we train HMM on 80% of the files for each family and they are tested against the rest of the 20% of the malware files and benign files. Scatter plots are generated for these scores and ROC curves are used to show how well the morphed malware can be detected using HMM. Based on the ROC, AUC is calculated. The Table 8 lists the number of files and number of iterations used to train HMM model.

Malware Family	No of Training files	No of Iterations
MWOR	100 files	1000
Zbot	200 files	1000
Zeroaccess	100 files	1000
Winwebsec	250 files	1000

Table 8: Files used to train HMM

5.7 Opcode Graph Similarity Computation

To compare malware derivatives using opcode graph similarity, we generate opcode graphs for the base malware file and the morphed malware variant. These opcode graphs are then compared using the method described in Chapter 3 and similarity scores are generated. Box plots are plotted to show the variance in the similarity scores.

In order to see how well the opcode graph similarity method can detect variants of a malware as against the benign files, we compute the opcode graph similarity score for morphed malware:malware and morphed malware:benign pairs. If there are N morphed malware files, we compare $\binom{N}{2}$ files to generate the malware scores. If there are M benign files, then we do $N \cdot M$ comparisons to generate benign scores. The scores are then plotted in scatter plot and ROC is generated. The AUC is calculated

from the ROC.

5.8 *N*-gram Similarity Computation

To compare malware derivatives using *N*-gram similarity, we compare opcodes in the base malware file and the morphed malware files as discussed in Chapter 3. Box plots are used to show the variance in the similarity scores.

To generate the ROC curves, opcode list from the same family are compared. If there are N files, $\binom{N}{2}$ files are compared and malware scores are generated. To generate the scores for benign files, opcode list of the malware files and benign files are compared. If there are N malware files and M benign files, then $N \cdot M$ scores are generated. The scores are then plotted in scatter plot and ROC is generated. The AUC is calculated from the ROC.

5.9 Cosine Similarity Computation

To compare malware derivatives using cosine similarity, we generate opcode frequency vectors for the base malware file and the morphed malware files. We extract the opcode sequences from the base malware files and compute the frequency of the opcodes in the file. This is the opcode frequency vector for the base malware files. We do the similar thing for the variant malware file as well. Then the cosine distance between the opcode frequency vectors is computed using the method described in Chapter 3. We computed the cosine distance between multiple base files and their variants. Box plots are plotted to show the variance in the cosine scores.

To generate the ROC curves, cosine distance of the files from the same family are compared. If there are N files, $\binom{N}{2}$ files are compared and malware scores are generated. To generate the scores for benign files, cosine distance between the mal-

ware files and the benign files are calculated. If there are N malware files and M benign files, then $N \cdot M$ scores are generated. Scatter plots are plotted, ROC curves are generated from scatterplots and then calculated AUC from ROC.

5.10 ROC Curves

Receiver Operating Characteristics (ROC) [10] curve is a convenient way to evaluate the performance of the system. For a binary classifier, say for predicting malware, there are two kinds of errors - malware classified as benign (false negative) and benign classified as malware (false positive). Different applications can have very different requirements for false negatives and false positives. Squashing the classifications to two numbers - *precision* and *recall* loses a lot of information. Instead, generating an ROC curve gives us the ability to visualize the performance of the system in a simple plot.

In an ROC curve, true positive rate (TPR) and false positive rate (FPR) are plotted [27], with different thresholds. We see how well the thresholds separate true positives and false positives. The X-axis and Y-axis always starts at 0.0 and ends at 1.0. An example ROC curve is given in Figure 16.

5.10.1 Area under the curve

The area under the curve (AUC) is a good measure of the performance of a binary classifier. The AUC can be calculated from the ROC curve using the trapezoidal rule. For the points in the ROC curve, we compute the area covered by each pair of neighbouring points with the trapezoidal formula and sum them. Since the ROC curve is plotted on a unit square, the maximum value for AUC is 1.0 for a perfect classifier. It is important to note that the AUC cannot be lower than 0.5. If we get

Figure 16: Example ROC curve

an AUC less than 0.5, we can flip the match and nomatch conditions in the classifier to get an AUC of greater than 0.5.

CHAPTER 6

Observations

The following sections discuss the results on the analysis of different similarity techniques used to measure evolution of malware families MWOR. The opcode list files from the MWOR malware family are morphed with different morphing ratios using morphing tool as explained in Chapter 4. Then the similarity techniques are applied to compare the morphed families and base families as explained in Chapter 5 and box plots are generated. In this chapter we discuss the results and details for the MWOR family. The benign dataset used for this experiment is cygwin utilities. The results, plots and graphs for other families can be found in appendices.

6.0.2 Note about reading figures

The Figures D.45, D.46, D.47, 20 have 8 plots each and here we explain the different plots in figures. The figures are organized into two columns - the left column with subfigures[a,c,e,g] have the results of experiment 1 mentioned in Section 5.4.1 and the right column with subfigures[b,d,f,h] have the results of experiment 2 described in Section 5.4.2. In Figure B.30(a), we compare the base opcodes with morphed variants using HMM in Section 3.4. The box plots combine the scores of 100 different files for MWOR. Figures B.30(c), B.30(e), B.30(g) compare the base opcodes with morphed variants using opcode-graph, n -gram comparison and cosine similarity discussed in Sections 3.3, 3.2 3.1 respectively.

6.1 Deletion Morphing

The plots in Figure D.45 show the performance of different methods in comparison.

- **HMM:** As the morphing ratio increases, the HMM scores do not change much. This indicates that HMM is able to detect similar malware even after heavy deletion morphing. If the code has only undergone deletion morphing, HMM is a suitable technique to identify malware. The plot on the right Figure D.46(a) shows that HMM models built from the morphed samples can be used to effectively discriminate between morphed malware and benign files.
- **OpcodeGraph:** As we increase the morphing ratio, the scores of opcode graph increase significantly. This is because of the fact, we delete more and more opcodes and a lot of opcode to opcode transitions disappear. This is the reason why the scores change significantly. The ROC plot Figure D.46(b) shows that, opcode graph technique is able to split the files even after substantial deletion.
- **N -gram:** As we increase the code deletion, the scores of n -gram similarity method decrease from 0.5 to less than 0.1. This is because of lot of missing tri-grams in the input. Missing tri-grams lead to lesser matching trigrams and hence lesser score. This makes the n -gram similarity well suited to measure how much deletion morphing a program has undergone. The ROC plot Figure D.46(c) shows that, as we increase the morphing ratios, n -gram similarity cannot effectively distinguish between the benign and malware files.
- **Cosine:** The box plot Figure B.30(g) shows that cosine similarity scores do not change much when the morphing ratio changes. The total difference in the scores is less than 0.01. Even after bulk code deletion, the cosine angle does not

change much. The ROC plot shows that, cosine similarity is able to distinguish between the benign and malware files. This can be attributed to very large angles between benign and malware files in general.

6.2 Insertion Morphing

For insertion, we insert opcodes randomly from the input files. While comparing the base opcodes and the morphed-with-insertion opcodes, we find that all four comparison methods do reasonably well.

- **HMM:** With insertion morphing, HMM produces much lower scores when the morphing ratio increases. As we insert more and more opcodes while morphing, this introduces unseen opcode to opcode transitions in the resulting morphed variant. Since the HMM has not seen these transitions it may have assigned very low probability for these transitions. This results in increasingly lower scores as we increase the morphing ratio. The ROC plot Figure D.47(a) shows that HMM is still very powerful to detect benign files from heavily morphed files.
- **OpcodeGraph:** As we add more opcodes, the opcode graph distance increases. The ROC plot Figure D.47(b) indicates that opcode graph does not detect benign and morph files well as the morphing ratio increases.
- ***N*-gram:** The *n*-gram scores decrease substantially as we insert more opcodes. The ROC plot shows that *n*-gram similarity is able to detect malware and benign files with a lot of insertion. In insertion we only introduce new tri-grams and all existing tri-grams remain the same except at the boundary of insertion.
- **Cosine:** The scores of cosine is very similar to the deletion case.

6.3 Substitution Morphing

Here we compare the base opcodes with the morphed variants after substitution. All four methods are employed for comparison.

- **HMM:** As the morphing ratio of substitution increases, the HMM scores change significantly. This indicates that HMM is not able to detect that the morphed families are variants of malware after insertion morphing. In substitution we replace the original opcode with the opcodes from the unique opcode list. So in a way, this can be viewed as deleting the opcode and inserting opcodes. Substitution of opcodes increased the code from 10% to 40% for 10% to 40% substitution respectively. In the Figure B.33(a) we can see that the similarity score decreases from -15 to -35 for 54 % to 219% increase in the data and Figure 20(a) shows that the score decrease from -10 to -30 for 10% to 40% increase in the data. As we already know that HMM scores does not change much after bulk deletion of opcodes, the morphing effect we see in HMM scores is only because of insertion of data in substitution. For insertion and substitution morphing, HMM stays same. Though HMM cannot say the variants are similar malwares, the ROC plot in Figure D.48(a) shows that the HMM is able to distinguish between malware variants and benign.
- **OpcodeGraph:** As we increase the substitution morphing ratio, the scores of opcode graph increase and this explains that they are becoming dissimilar from base malwares. For similar malwares opcode graph would score 0 and here we can see that opcode graph scores are going up , shows that the base malwares and morphed malwares are different. Also the ROC plot shows that opcode graph does not do well after code substitution morphing and this shows that

the opcode graph is not classifying benign and malware variants correctly.

- ***N*-gram:** As we increase the substitution ratio, the *n*-gram scores go from 0.4 to less than 0.05, indicating that they are becoming very dissimilar. We observe from the Figure D.48(c) that the *n*-gram similarity is doing good for substitution.
- **Cosine:** The box plot in Figure 20(g) shows that cosine similarity scores change well, as compared to deletion. Even after bulk code deletion, the cosine angle does not change much but for substitution the scores go down to 0.96 as opposed to deletion. Substitution morphing has good effect on the malware and the similarity methods cannot detect they are similar malwares. The ROC plot shows that, cosine similarity is able to distinguish between the benign files and malware variants. This can be attributed to large distance between the benign and malware files in general.

6.4 Transposition Morphing

All the four methods shows no difference in the comparison scores. Cosine similarity calculates distance between the opcode frequency vectors and in transposition we only move the existing code in different places, we do not alter the occurrences of the opcodes in the program. So there is no wonder cosine similarity scores 1, while comparing all the four variants. Also we notice that all four methods do well in detecting the malware after transposition morphing. In transposition here, we do transposition using `jmp` instructions and did relatively less morphing when compared to other morphing methods. For the malware families in the appendices, we do subroutine transposition with a higher morphing ratios.

The Table 9 shows the AUC values for different variants scored using different similarity methods. We can observe that HMM and cosine similarity works well in detecting the malware and benign files. For transposition all the four methods work the same in classifying malware and benign because their similarity scores remain the same and they do not change much.

		AUC values for Morphed variants-MWOR			
Method	Resulting file size %	Cosine	<i>N</i> -gram	OpcodeGraph	HMM
Deletion	63.07%	1	0.877	0.982	1
Deletion	42.23%	1	0.560	0.978	1
Deletion	29.86%	1	0.784	0.981	1
Deletion	21.44%	1	0.845	0.975	1
Insertion	155.19%	1	0.916	0.603	1
Insertion	209.53%	1	1	0.609	1
Insertion	265.00%	1	0.999	1	1
Insertion	319.88%	1	0.955	1	1
Substitution	110.03%	1	0.855	0.694	1
Substitution	120.15%	1	0.999	0.622	1
Substitution	129.93%	1	0.963	0.596	1
Substitution	140.05%	1	0.945	0.526	1
Transposition	100.20%	1	0.995	0.977	1
Transposition	100.41%	1	0.991	0.977	1
Transposition	100.58%	1	0.988	0.977	1
Transposition	100.79%	1	0.985	0.977	1

Table 9: AUC for MWOR

Figure 17: MWOR deletion experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-gram

(f) N-gram-roc

(g) Cosine

(h) Cosine-roc

Figure 18: MWOR insertion experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-gram

(f) N-gram roc

(g) Cosine

(h) Cosine-roc

Figure 19: MWOR substitution experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-gram

(f) N-gram -roc

(g) Cosine

(h) Cosine-roc

Figure 20: MWOR transposition experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-gram

(f) N-gram -roc

(g) Cosine

(h) Cosine-roc

CHAPTER 7

Conclusion and Future work

The morphing techniques described in this paper simulates real world malware evolution by insertion, deletion, substitution and transposition of opcodes. The developing morphing tool is a general purpose morphing engine that can be used to simulate real-world malware evolution and generate training data for malware analysis algorithms.

We did two main experiments - one to morph the malware and apply different methods to compare the base malware with its morphed variants, two to analyse how well the morphed variants can be discriminated from benign files. We applied four different comparison methods in the above experiments.

We found that in most cases HMM performed better than the other methods. Even after morphing ratio was increased to 40%, resulting in 200% change in the input file, the HMM was able to detect malware from benign files. If we have multiple morphed samples of a malware, taken at different stages of evolution, we can use n-gram similarity and opcode graph similarity to correctly identify the position of the malware sample in its evolution stages, assuming that the morphing methods described above are used.

We studied the comparison methods using insertion, deletion, substitution and transposition in isolation as well as combination of different comparison methods as explained in Appendix E.

There is a big scope of improving and scaling this research project. One can apply these techniques developed on several other malware families and compare with the

methods discussed. This project is limited to using static opcode features from the malware. One can extract function API calls or opcodes dynamically and run through the morphing-comparison cycle. There is a lot of recent work on malware detection using machine learning, especially in the field of deep neural-networks [22]. The above experiments can be formulated in the deep-learning frame work.

In this morphing-tool implementation, while injecting new opcodes, we assumed that all opcodes are equally likely. This assumption can be relaxed and we can sample insertion opcodes based on its distribution in the input malware.

LIST OF REFERENCES

- [1] J. Aycock, *Computer Viruses and Malware*, Springer Science and Business Media, 2006
- [2] S. Attaluri, S. McGhee, and M. Stamp, Profile hidden Markov models and metamorphic virus detection, *Journal in computer virology*, 5(2):151–169, 2009
- [3] T. H. Austin, E. Filiol, S. Josse, and M. Stamp, Exploring hidden Markov models for virus analysis: a semantic approach, *System Sciences (HICSS)*, 2013
- [4] D. Baysa, R. M. Low, and M. Stamp, Structural entropy and metamorphic malware, *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, 2013
- [5] E. Chris, *The IDA pro book: the unofficial guide to the world’s most popular disassembler*, No Starch Press, 2011
- [6] P. Deshpande, and M. Stamp, Metamorphic detection using function call graph analysis, Master’s report, Department of Computer Science, San Jose State University, 2013, http://scholarworks.sjsu.edu/etd_projects/336
- [7] F. N. Dezfouli, A. Dehghantanha, R. Mahmood, N. F. B. M. Sani, S. B. Shamsuddin, and F. Daryabar, A Survey on Malware Analysis and Detection Techniques, *International Journal of Advancements in Computing Technology*, 5(14):42, 2013
- [8] S. Deshpande, Y. Park, and M. Stamp, Eigenvalue analysis for metamorphic detection, *Journal of computer virology and hacking techniques*, 10(1):53–65, 2014
- [9] E. Gandotra, D. Bansal, and S. Sofat, Malware analysis and classification: A survey, *Journal of Information Security*, 2014
- [10] J. A. Hanley, and B. J. McNeil, The meaning and use of the area under a receiver operating characteristic (ROC) curve, *Radiology*, 143(1):29–36, 1982
- [11] X. D. Huang, Y. Ariki, and M. A. Jack, *Hidden Markov models for speech recognition*, Edinburgh: Edinburgh university press, 1990
- [12] R. K. Jidigam, T. H. Austin, and M. Stamp, Singular value decomposition and metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 1–14, 2014.

- [13] K. Kaushal, P. Swadas, and N. Prajapati, Metamorphic malware detection using statistical analysis, *International Journal of Soft Computing and Engineering (IJSCE)* , 2(3):49–53, 2012
- [14] M. E. Karim, A. Walenstein, A. Lakhota, and L. Parida, Malware phylogeny generation using permutations of code, *Journal in Computer Virology*, 1(1–2):13–23, 2005
- [15] J. Kupiec, Robust part-of-speech tagging using a hidden Markov model, *Computer Speech and Language*, 6(3):225–242, 1992
- [16] J. Lee, T. H. Austin, and M. Stamp, Compression-based analysis of metamorphic malware, to appear in *International Journal of Security and Networks*
- [17] E. Manuel, T. Scholte, E. Kirda, and C. Kruegal, A survey on automated dynamic malware-analysis techniques and tools, *ACM Computing Surveys (CSUR)*, 44(2):6, 2012
- [18] L. R. Rabiner, and B. H. Juang, An introduction to hidden Markov models, *ASSP Magazine*, IEEE 3(1):4–16, 1986
- [19] N. Runwal, R. M. Low, and M. Stamp, Opcode graph similarity and metamorphic detection, *Journal in Computer Virology*, 8(1–2):37–52, 2012
- [20] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laordan, and P. G. Bringas, Idea: Opcode-sequence-based malware detection, *Engineering Secure Software and Systems*, Springer Berlin Heidelberg, 35–43, 2010
- [21] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, N-grams-based File Signatures for Malware Detection, *ICEIS(2)*, 9:317–320, 2009
- [22] J. Saxe, and K. Berlin, Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features, *arXiv preprint arXiv:1508.03096*, 2015
- [23] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “Andromaly”: a behavioral malware detection framework for android devices, *Journal of Intelligent Information Systems*, 38(1):161–190, 2012
- [24] G. Shanmugam, R. M. Low, and M. Stamp, Simple substitution distance and metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, 2013
- [25] S. M. Sridhara, and M. Stamp, Metamorphic worm that carries its own morphing engine, *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, 2013

- [26] M. Stamp, A revealing introduction to hidden Markov models, *Department of Computer Science San Jose State University*, 2004
- [27] M. Stamp, *Information Security: Principles and Practice*, second edition, Wiley, 2011
- [28] T. Singh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, Support vector machines and malware detection, *Journal of Computer Virology and Hacking Techniques*, 1–10, 2015
- [29] P. Swathi, A Comparison of Clustering Techniques for Malware Analysis, 2015
- [30] A. H. Toderici, and M. Stamp, Chi-squared distance and metamorphic virus detection, *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013
- [31] Symantec(2010), Trojan.Zbot,
http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99
- [32] Symantec, Trojan.Citadel,
https://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=26434, Accessed 2015
- [33] Symantec(2013), Trojan.ZeroAccess,
http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99
- [34] Symantec(2011), W32.Stuxnet Dossier,
https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf
- [35] P. Vinod, R. Jaipur, V. Laxmi, M. Gaur, Survey on malware detection methods, *Proceedings of the 3rd Hackers Workshop on Computer and Internet Security (IITKHACK'09)*, 2009
- [36] A. Walenstein, M. Venable, M. Hayes, M. Thompson, C. Lakhoria, Exploiting similarity between variants to defeat malware, *Proc. BlackHat DC Conf*, 2007
- [37] Win32/Winwebsec
<http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fWinwebsec>, Accessed 2015
- [38] W. Wong, and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, 2(3):211–229, 2006

APPENDIX A

Zeroaccess experiments

Figure A.21: Cosine mean scores

Figure A.22: Opcodegraph mean scores

Figure A.23: N-gram mean scores

Figure A.24: HMM mean scores

Similarity Method	Morph Vs Benign scores	Morph Vs Morph
opcode graph	30 .30 = 900	30 choose 2 = 436
<i>N</i> -gram	30 .30 = 900	30 choose 2 = 436
Cosine	30 .30 = 900	30 choose 2 = 436

HMM	Morphing ratio	Total no of files	No. of iterations
Deletion	all	100	1000
Substitution	all	100	1000
Transposition	all	100	1000
Insertion	all	40	1000

		AUC values for Morphed variants-Zeroaccess			
Method	Morphed data %	Cosine	<i>N</i> -gram	OpcodeGraph	HMM
Deletion	37.91%	0.985	0.575	0.601	0.946
Deletion	57.89%	0.991	0.579	0.635	0.922
Deletion	70.18%	0.986	0.761	0.688	0.951
Deletion	78.59%	0.982	0.764	0.723	0.930
Insertion	54.93%	0.994	0.523	1	0.956
Insertion	110.05%	0.999	0.835	1	0.978
Insertion	164.99%	1	0.786	1	0.99
Insertion	219.76%	1	0.827	1	0.99
Substitution	10%	0.991	0.508	1	0.793
Substitution	20%	0.993	0.693	1	0.951
Substitution	30%	0.997	0.791	1	0.971
Substitution	40%	0.998	0.797	1	0.986
Transposition	52%	0.992	0.571	0.561	0.928
Transposition	71%	0.992	0.521	0.558	0.925
Transposition	81%	0.992	0.518	0.557	0.925
Transposition	87%	0.992	0.523	0.559	0.925

Table A.10: AUC for Zeroaccess

Figure A.25: Zeroaccess deletion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-gram-roc

(d) Cosine-roc

Figure A.26: Zeroaccess insertion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) *N*-gram-roc

(d) Cosine-roc

Figure A.27: Zeroaccess substitution experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) *N*-gram-roc

(d) Cosine-roc

Figure A.28: Zeroaccess transposition experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) *N*-gram-roc

(d) Cosine-roc

APPENDIX B

Zbot experiments

Similarity Method	Morph Vs Benign scores	Morph Vs Morph
opcode graph	30 .40 = 900	30 choose 2 = 436
<i>N</i> -gram	30 .40 = 900	30 choose 2 = 436
Cosine	30 .40 = 900	30 choose 2 = 436

HMM	Morphing ratio	Total no of files	No. of iterations
Deletion	all	200	1000
Substitution	all	200	1000
Transposition	all	200	1000
Insertion	all	90	1000

Method	Morphed data %	AUC values for morphed variants-Zbot			
		Cosine	<i>N</i> -gram	OpcodeGraph	HMM
Deletion	37.91%	0.603	0.977	0.579	0.917
Deletion	57.89%	0.606	0.777	0.640	0.947
Deletion	70.18%	0.612	0.560	0.682	0.919
Deletion	78.59%	0.572	0.535	0.752	0.816
Insertion	54.93%	0.621	0.970	0.998	0.504
Insertion	110.05%	0.638	0.555	1	0.750
Insertion	164.99%	0.660	0.519	1	0.831
Insertion	219.76%	0.693	0.654	1	0.867
Substitution	10%	0.618	0.982	0.909	0.971
Substitution	20%	0.616	0.653	0.983	0.995
Substitution	30%	0.612	0.694	0.999	1
Substitution	40%	0.630	0.623	1	1
Transposition	52%	0.615	0.989	0.525	0.922
Transposition	71%	0.615	0.937	0.520	0.923
Transposition	81%	0.615	0.931	0.522	0.924
Transposition	87%	0.615	0.918	0.521	0.924

Table B.11: AUC for Zbot

Figure B.29: Zbot deletion experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-Gram

(f) N-Gram roc

(g) Cosine

(h) Cosine-roc

Figure B.30: Zbot insertion experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-Gram

(f) N-Gram roc

(g) Cosine

(h) Cosine-roc

Figure B.31: Zbot substitution experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-Gram

(f) N-Gram roc

(g) Cosine

(h) Cosine-roc

Figure B.32: Zbot transposition experiments

(a) HMM

(b) HMM-roc

(c) OpcodeGraph

(d) OpcodeGraph-roc

(e) N-Gram

(f) N-Gram roc

(g) Cosine

(h) Cosine-roc

Figure B.33: Zbot n-gram insertion

Figure B.34: Zbot n-gram insertion

Figure B.35: Zbot n-gram insertion

APPENDIX C

Winwebsec experiments

Figure C.36: Cosine mean scores

Figure C.37: Opcodegraph mean scores

Figure C.38: Ngram mean scores

Similarity Method	Morph Vs Benign scores	Morph Vs Morph
opcode graph	30 .30 = 900	30 choose 2 = 436
N -gram	30 .30 = 900	30 choose 2 = 436
Cosine	30 .30 = 900	30 choose 2 = 436

HMM	Morphing ratio	Total no of files	No. of iterations
Deletion	all	250	1000
Substitution	all	250	1000
Transposition	all	250	1000
Insertion	10%	250	1000
Insertion	20%	60	1000
Insertion	30%	50	1000
Insertion	40%	40	1000

Figure C.39: HMM mean scores

Figure C.40: Winwebsec deletion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-Gram roc

(d) Cosine-roc

Figure C.41: Winwebsec insertion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-Gram roc

(d) Cosine-roc

Figure C.42: Winwebsec substitution experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-Gram roc

(d) Cosine-roc

Figure C.43: Winwebsec transposition experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-Gram roc

(d) Cosine-roc

		AUC values for Morphed variants-Winwebsec			
Method	Morphed data %	Cosine	<i>N</i> -gram	OpcodeGraph	HMM
Deletion	37.91%	0.663	0.602	0.791	0.936
Deletion	57.89%	0.619	0.511	0.808	0.933
Deletion	70.18%	0.639	0.616	0.831	0.928
Deletion	78.59%	0.634	0.643	0.862	0.923
Insertion	54.93%	0.646	0.563	0.994	0.676
Insertion	110.05%	0.665	0.583	1	0.522
Insertion	164.99%	0.689	0.685	1	0.515
Insertion	219.76%	0.717	0.666	1	0.547
Substitution	10%	0.642	0.578	0.827	0.841
Substitution	20%	0.640	0.568	0.958	1
Substitution	30%	0.654	0.647	0.996	1
Substitution	40%	0.671	0.678	1	1
Transposition	52%	0.637	0.632	0.757	0.939
Transposition	71%	0.637	0.599	0.755	0.938
Transposition	81%	0.637	0.599	0.754	0.938
Transposition	87%	0.637	0.566	0.752	0.938

Table C.12: AUC for Winwebsec

APPENDIX D

MWOR experiments with linux utilities

Similarity Method	Morph Vs Benign scores	Morph Vs Morph
opcode graph	100 .100 = 10000	100 choose 2 = 4951
<i>N</i> -gram	100 .100 = 10000	100 choose 2 = 4951
Cosine	100 .100 = 10000	100 choose 2 = 4951

Similarity Method	Total No of files	CrossValidation	No. of iterations
HMM	100	k = 5	1000

Figure D.44: Distribution of opcodes in linux files

		AUC values for Morphed variants-MWOR			
Method	Morphed data %	Cosine	<i>N</i> -gram	OpcodeGraph	HMM
Deletion	37.91%	1	0.990	0.986	1
Deletion	57.89%	1	0.871	0.986	1
Deletion	70.18%	1	0.721	0.984	1
Deletion	78.59%	1	0.619	0.988	1
Insertion	54.93%	1	0.780	0.902	1
Insertion	110.05%	1	0.694	0.546	1
Insertion	164.99%	1	0.867	0.948	1
Insertion	219.76%	1	0.858	1	1
Substitution	10%	1	0.883	0.918	1
Substitution	20%	1	0.639	0.864	1
Substitution	30%	1	0.815	0.770	1
Substitution	40%	1	0.810	0.700	1
Transposition	52%	1	0.997	0.987	1
Transposition	71%	1	0.989	0.985	1
Transposition	81%	1	0.984	0.983	1
Transposition	87%	1	0.979	0.983	1

Table D.13: AUC for MWOR

Figure D.45: MWOR deletion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-gram-roc

(d) Cosine-roc

Figure D.46: MWOR insertion experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) *N*-gram roc

(d) Cosine-roc

Figure D.47: MWOR substitution experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) N-gram -roc

(d) Cosine-roc

Figure D.48: MWOR transposition experiments

(a) HMM-roc

(b) OpcodeGraph-roc

(c) *N*-gram -roc

(d) Cosine-roc

APPENDIX E

Zbot combination morphing experiments

Morphing 1	Morphing 2	No. of training files	No. of iterations
Deletion	Substitution	250	1000
Deletion	Transposition	250	1000
Substitution	Transposition	250	1000
Insertion	Deletion	250	1000
Insertion	Substitution	200	1000
Insertion	Transposition	200	1000

Similarity Method	Morphed files	Benign files
opcode graph	500	214
N -gram	50	50
Cosine	500	214

Figure E.49: Deletion vs Substitution vs AUC for HMM

Figure E.50: Deletion vs Transposition vs AUC for HMM

Figure E.51: Insertion vs Deletion vs AUC for HMM

Figure E.52: Substitution vs Transposition vs AUC for HMM

Figure E.53: Insertion vs Transposition vs AUC for HMM

Figure E.54: Insertion vs Substitution vs AUC for HMM

Figure E.55: Deletion vs Substitution vs AUC for Cosine

Figure E.56: Insertion vs Transposition vs AUC for Cosine

Figure E.57: Deletion vs Transposition vs AUC for Cosine

Figure E.58: Substitution vs Transposition vs AUC for Cosine

Figure E.59: Insertion vs Deletion vs AUC for Cosine

Figure E.60: Insertion vs Substitution vs AUC for Cosine

Figure E.61: Insertion vs Substitution vs AUC for Opcodegraph

Figure E.62: Deletion vs Substitution vs AUC for Opcodegraph

Figure E.63: Insertion vs Transposition vs AUC for Opcodegraph

Figure E.64: Deletion vs Transposition vs AUC for Opcodegraph

Figure E.65: Substitution vs Transposition vs AUC for Opcodegraph

Figure E.66: Insertion vs Deletion vs AUC for Opcodegraph

Figure E.67: Insertion vs Substitution vs AUC for Ngram method

Figure E.68: Deletion vs Substitution vs AUC for Ngram method

Figure E.69: Insertion vs Transposition vs AUC for Ngram method

Figure E.70: Deletion vs Transposition vs AUC for Ngram method

Figure E.71: Substitution vs Transposition vs AUC for Ngram method

Figure E.72: Insertion vs Deletion vs AUC for Ngram method