San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Fall 2013

COMPRESSION-BASED ANALYSIS OF METAMORPHIC MALWARE

Jared Lee

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

Recommended Citation
Lee, Jared, "COMPRESSION-BASED ANALYSIS OF METAMORPHIC MALWARE" (2013). *Master's Projects*. 329.
http://scholarworks.sjsu.edu/etd_projects/329

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.

COMPRESSION-BASED ANALYSIS OF METAMORPHIC MALWARE

A Thesis

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

Jared Lee

December 2013

© 2013
Jared Lee
ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

COMPRESSION-BASED ANALYSIS OF METAMORPHIC MALWARE

by

Jared Lee

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2013

Dr. Thomas Austin	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science
Dr. Teng Moh	Department of Computer Science

ABSTRACT

Compression-based Analysis of Metamorphic Malware

by Jared Lee

Recent work has presented a technique based on structural entropy measurement as an effective way to detect metamorphic malware. The technique uses two steps, file segmentation and sequence comparison, to calculate file similarity. In another previous work, it was observed that similar malware have similar measures of Kolmogorov complexity. A proposed method of estimating Kolmogorov complexity was to calculate the compression ratio of a given malware which could then be used to cluster the malicious software. Malware detection has also been attempted through the use of adaptive data compression and showed promising results. In this paper, we attempt to combine these concepts and propose using compression ratios as an alternative measure of entropy with the purpose of segmenting files according to their structural characteristics. We then compare the segment-based sequences of two given files to determine file similarity. The idea is that even after malware is transformed using a metamorphic engine, the resulting variants still share identifiable structural similarities with the original. Using this proposed technique to identify metamorphic malware, we compare our results with previous work.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Thomas Austin, for his invaluable guidance. I would also like to thank my committee members, Dr. Sami Khuri and Dr. Teng Moh, for their unwavering support. Finally, I would like to thank Dr. Mark Stamp for his direction and encouragement, as this project would not have been possible without him.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 3

2.1 Malware 3

2.1.1 Types 3

2.1.2 Concealment Strategies 4

2.1.3 Metamorphic Techniques 5

2.2 Related Work 7

2.2.1 Hidden Markov Model 7

2.2.2 Structural Entropy 9

2.2.3 Compression-based Classification 10

3 Design and Implementation 13

3.1 Forming File Segments 13

3.1.1 Splitting a File into Byte Windows 13

3.1.2 Window Compression Ratios 14

3.1.3 Wavelet Transform Analysis 16

3.1.4 File Segment Creation 18

3.2 Sequence comparison 19

3.2.1 Levenshtein distance 19

3.2.2 Sequence alignment 20

3.2.3 Similarity calculation 22

4 Experiments and Results Analysis 23
4.1 Test Data 23
4.1.1 Second Generation Virus Generator 23
4.1.2 MWOR 24
4.1.3 Next Generation Virus Construction Kit 25
4.2 Receiver Operating Characteristic 25
4.3 Test Results 26
4.4 Setting Parameters 30
4.4.1 Cost Constants 30
4.4.2 Window Slide Size 30
4.4.3 Recursive Iterations of the Wavelet Transform 34
4.4.4 Compression Ratio Threshold 34
5 Conclusion and Future Work 36

APPENDIX

A MWOR Results 43
B Wavelet transforms on MWOR (0.5 padding ratio) 47
C Wavelet transforms on NGVCK 51
D Wavelet transforms on G2 55

LIST OF TABLES

1 Resulting File Segments 18

2 Edit matrix for strings "books" and "broom" 21

3 Cygwin Utility Files Used For Benign Data Set 23

4 Linux Binaries Used For Benign Data Set 25

5 Results Summary 30

6 Compression Ratio Threshold Calibration 35

LIST OF FIGURES

1 Generic Hidden Markov Model 8

2 File Segmentation 10

3 Kolmogorov Complexity Detection Framework 11

4 PPM-based Classification 12

5 Hexdump of sample file with addresses(Left), bytes in hex representation(Center), bytes in ascii representation(Right) 14

6 Window 1 of sample file 14

7 Window 2 of sample file 14

8 Compression ratios plot of sample file 15

9 Discrete wavelet transform using (a) 0 iterations, (b) 1 iteration, (c) 2 iterations, (d) 3 iterations 17

10 Example ROC Curve 26

11 G2 similarity 27

12 MWOR(0.5 padding) similarity 28

13 NGVCK similarity 29

14 ROC Curve Results 29

15 Window Slide Size vs. Total Execution Time 31

16 Window Slide Size vs. AUC 32

17 Window Slide Size vs. AUC 33

18 Window Slide Size vs. AUC 33

A.19 MWOR(1.0 padding) similarity 44

A.20 MWOR(1.5 padding) similarity 44

A.21 MWOR(2.0 padding) similarity 45

A.22 MWOR(2.5 padding) similarity 45

A.23 MWOR(3.0 padding) similarity 46

A.24 MWOR(4.0 padding) similarity 46

B.25 Wavelet Transform → 0 iterations (MWOR 0.5) 48

B.26 Wavelet Transform → 1 iteration (MWOR 0.5) 48

B.27 Wavelet Transform → 2 iterations (MWOR 0.5) 49

B.28 Wavelet Transform → 3 iterations (MWOR 0.5) 49

B.29 Wavelet Transform → 4 iterations (MWOR 0.5) 50

B.30 Wavelet Transform → 5 iterations (MWOR 0.5) 50

C.31 Wavelet Transform → 0 iterations (NGVCK) 52

C.32 Wavelet Transform → 1 iteration (NGVCK) 52

C.33 Wavelet Transform → 2 iterations (NGVCK) 53

C.34 Wavelet Transform → 3 iterations (NGVCK) 53

C.35 Wavelet Transform → 4 iterations (NGVCK) 54

C.36 Wavelet Transform → 5 iterations (NGVCK) 54

D.37 Wavelet Transform → 0 iterations (G2) 56

D.38 Wavelet Transform → 1 iteration (G2) 56

D.39 Wavelet Transform → 2 iterations (G2) 57

D.40 Wavelet Transform → 3 iterations (G2) 57

CHAPTER 1

Introduction

Malicious software, or malware, continue to be a threat in spite of advances in computer security [30, 35]. The detection of metamorphic malware, in particular, remains a challenging area of research due to various complexities involved [6, 33]. Metamorphic malware modify their internal structure at each infection, while remaining functionally equivalent [17]. This feature makes them very difficult to detect since the obfuscation used by metamorphic engines can allow them to defeat traditional malware detectors based on pattern matching [8, 35].

Metamorphic malware detection through static program analysis is an active area of research and many techniques have been developed that show good results [7]. For example, previous work has shown that despite extensive changes in internal structure, some metamorphic malware can be effectively detected using statistical based methods of similarity measurement [30]. It has also been shown that metamorphic malware can be clustered by using compression ratios as a measure of Kolmogorov complexity [29]. Unfortunately, there are a multitude of obfuscation techniques that render malware detection through static analysis either much less effective or highly resource intensive [6, 23, 33].

Recently, a novel approach utilizing structural entropy analysis has been developed and shows good resilience against obfuscation [3, 22]. This approach takes advantage of structural entropy to measure varying levels of data complexity throughout a file and uses these characteristics to calculate a similarity measure. The method involves two stages, file segmentation and sequence comparison. The

file segmentation stage uses entropy measurement along with wavelet analysis and the sequence comparison is done using Levenshtein distance. Levenshtein distance, or edit distance, is a similarity measure of two strings [26].

Good results were shown when the structural entropy technique was applied to various families of metamorphic malware [3]. However, some cases proved particularly difficult to detect. In this project, we extend the previous research by proposing the use of compression ratios as an alternative measurement of entropy. We also consider different adjustments to the file segmentation step and analyze the effects on file similarity scores. Finally, we compare our experimental results with those obtained in previous research.

CHAPTER 2

Background

2.1 Malware

Malware continues to be a major security threat in information systems. Computer viruses, worms, spyware, Trojan horses, rootkits, and other intentionally harmful software all fall under this category [12].

2.1.1 Types

A computer virus is a malicious software that attempts to copy itself into other executable code [2]. The now infected executable code can then be expected to infect new code when run. As a result, viruses can be defined by both their self-replicating and parasitic nature. In addition to other executable programs, viruses can also be commonly found in the boot sector and in memory [16].

Worms, although similar to viruses, differ in that they are standalone and do not require external executable code in order to run [2]. They can be expected to execute automatically on victim machines without the need for user interaction [25]. While viruses try to self-replicate to different host files within the local filesystem, worms try to self-replicate to different hosts across networks.

A Trojan horse is considered to be a piece of software that attempts to hide its malicious intent under the guise of benign behavior [2]. It distinguishes itself from other malware through its standalone nature and masquerading attempts [16]. Unlike viruses and worms, Trojan horses depend heavily on users for the purpose of dissemination. Backdoor Trojans, Distributed Attacks Trojans, and Remote

Administration Trojans are just some of the types that can be encountered.

Spyware is malware created with the intent to collect user activity on a victim machine, without the knowledge and consent of the user, and send that data to a third party [2]. While spyware does not share the self-replicating nature of viruses and worms, they can potentially cause greater financial loss due to stolen passwords, credit card numbers, and other sensitive information. Adware and key loggers are common types of spyware used by malicious attackers.

Rootkits are commonly considered to be tools that allow attackers to gain unauthorized administrative access to other systems [16]. Modern incarnations of rootkits also attempt to efficiently hide all traces that the system has been infiltrated and compromised. They can be categorized into user mode and kernel mode rootkits, with both being able to cause irreparable damage.

2.1.2 Concealment Strategies

Previously, signature-based detection schemes were employed against computer malware with great success. As time went on, however, attackers developed a number of advanced obfuscation techniques to morph their code that made traditional forms of detection much less effective.

One early technique used by malware writers was encrypting the body of the malware code [2]. The encryption method was not required to be complex and often involved a simple XOR with a fixed key [5]. The rationale was that by encrypting the malware body, the appearance of the malware code would drastically change thus evading signature detection. However, this technique was eventually defeated by looking for signatures in the malware decryptor rather than the body [2].

Techniques were then implemented to retain the ability to encrypt the malware body while simultaneously varying the decryption code across generations [18]. These polymorphic malware were effective since the number of possible unique decryptors proved to be too numerous for malware detectors to account for them all [2]. With a near infinite amount of possible decryptors, it quickly became infeasible for anti-virus tools to search for every possible signature. Emulation-based techniques proved effective against polymorphic malware by emulating the decryptor instructions in order to have the malware decrypt its own body making it vulnerable to signature detection once again [28]. However, emulation is typically very slow [2] which prevents it from being done in practical situations where time efficiency is a factor. This has caused polymorphism to remain a feasible and popular way to evade detection.

As a logical progression, metamorphic malware arose where polymorphic techniques were applied to the entire virus code. These malware change their entire internal structure with each generation while retaining functional equivalency [17]. This eliminated the need for encryption since signature-based detection already proved ineffective when sufficient morphing occurred. Consequently, detecting metamorphic malware with high accuracy is a challenging problem. Although truly effective metamorphic engines are very difficult to implement in practice, many already exist and we can only expect more in the future. This is the class of malware that the technique presented in this paper attempts to detect.

2.1.3 Metamorphic Techniques

Awareness of various morphing techniques is useful in order to help counter the obfuscation techniques employed by malware writers. We now consider an

overview of what techniques are commonly used.

Register swapping is a technique that involves using the same code but changing which registers are used in each generation in order to help avoid signature-based detection. It is often considered one of the simplest metamorphic techniques and one such implementation of this technique is Vecna's Win95/Regswap virus [16]. Although this technique was initially somewhat successful, it was quickly mitigated through the introduction of wildcards in signatures.

Instruction substitution replaces a set of instructions with a different set having equivalent functionality. For example, an implementation of this technique could be to replace instances of the instruction "`XOR EAX, EAX`" with the instruction "`SUB EAX, EAX`" [16]. Although the two previous instructions have different opcodes, the resulting functionality is equivalent. This morphing technique helps evade not only signature detection, but also certain detection techniques based on static program analysis.

Instruction reordering takes advantage of the ability to change the order of a sequence of instructions that have no dependencies [18]. For example, if two adjacent instructions are "`ADD EAX, 1`" and "`SUB RAX, 4`", they can be placed in any order without affecting functionality. By making such re-orderings on a larger scale, detection is made more difficult for detection techniques that factor in positional information such as ones that use state transition graphs.

Subroutine permutation is a natural extension to instruction reordering that reorders sections of code but then executes them in the original order at run-time through the use of jump statements at the end of each section [6]. This technique is

useful since splitting a program into n sections would yield $n!$ possible variations of the same malware.

Garbage code insertion is a simple, yet effective, technique used by many metamorphic engines to insert instructions have no impact on the execution of the program but makes the code look very different [16]. Inserting various amounts of garbage code into malware is an effective way to counter detection techniques that utilize op-code based statistical analysis. Furthermore, inserting parts of benign software as garbage code has proved to be particularly effective.

Formal grammar mutation is another obfuscation technique and begins by rewriting existing morphing techniques into a formal grammar [4, 13, 34]. After this transformation, various formal grammar rules can be applied to an input sequence to create a multitude of variations. In this case, the input sequence would be instructions in the malware program.

2.2 Related Work

The following section pertains to previous research in calculating file similarity. We give an overview of the techniques and review their effectiveness in detecting metamorphic malware.

2.2.1 Hidden Markov Model

A Hidden Markov Model (HMM) is a machine learning technique that models a system, treating the system as a Markov process with hidden states [20]. The assumption is made that the observations are probabilistically based on the hidden states and the only information given is the output sequence of the system. The model consists of initial state probabilities, state transition probabilities, and

probability distributions for all possible observations for each state. By training an HMM on a given set of observations, it is then possible to score another set of observations against the model and observe how well the second set of observations fit [20, 24]. As a result, Hidden Markov models are useful for applications that deal with statistical pattern analysis.

Figure 1 taken from [24] illustrates a generic Hidden Markov model. X_t represents a hidden state at time t with state transition probability A. Each observation O_i gives some information about the hidden state with regards to probability distribution B. The overall idea is that with a sufficiently large amount of observations, we can attempt to uncover the underlying Markov process and build a model that most closely represents the data. Previous work has analyzed

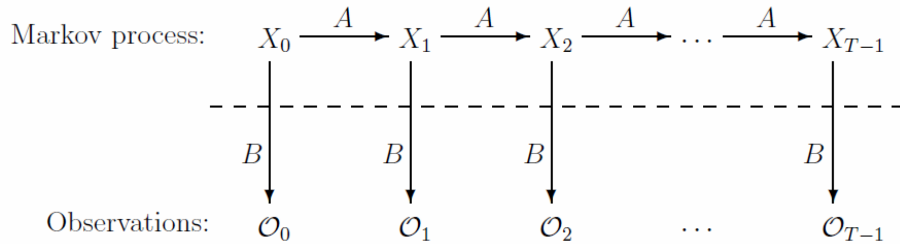


Figure 1: Generic Hidden Markov Model

using HMMs as a tool to detect metamorphic malware [1, 30]. Opcode sequences are extracted from a set of malware belonging to the same metamorphic family and are fed as input observations to the HMM. Unknown files are then disassembled to retrieve their respective opcode sequences and the sequences are scored against the trained HMM model. File similarity is then determined by how well the opcode sequences of the test files score against the trained model. By then setting a scoring threshold, which is determined by experimental means, a binary classifier is

effectively produced. The previous work produced good results and demonstrates that HMM-based detection using opcodes is an effective method of detecting metamorphic malware. However, it was shown in [23] that malware with sufficient amounts of dead code insertion are able to defeat this technique. This is due to dead code causing statistical inconsistencies in the opcode sequences.

2.2.2 Structural Entropy

Recent research [3] further developed the concept of using structural entropy calculations to identify file similarity. The structural entropy technique, originally introduced in [22], produced good results when applied to polymorphic malware. As a logical next step, the technique was adapted by [3] to apply it to metamorphic malware. As opposed to several previous detection techniques, structural entropy analysis examines the raw bytes of files rather than analyzing the disassembled opcode sequences. The intuition appears to be that entropy and size characteristics alone can uniquely identify families of metamorphic malware.

We now describe the technique presented in [3]. The proposed technique can be divided into two main stages, file segmentation and sequence comparison. File segmentation is performed by first calculating entropy using Shannon's formula. Immediately afterwards, a wavelet transform is applied to the resulting entropy values. Figure 2 illustrates the overall file segmentation process. The sequence comparison is then performed using the edit distance algorithm with a unique cost function described in the paper. The result of the algorithm is plugged into a final similarity formula which produces a score that can be compared against a pre-determined threshold.

The technique produced good results. Structural entropy analysis was able to

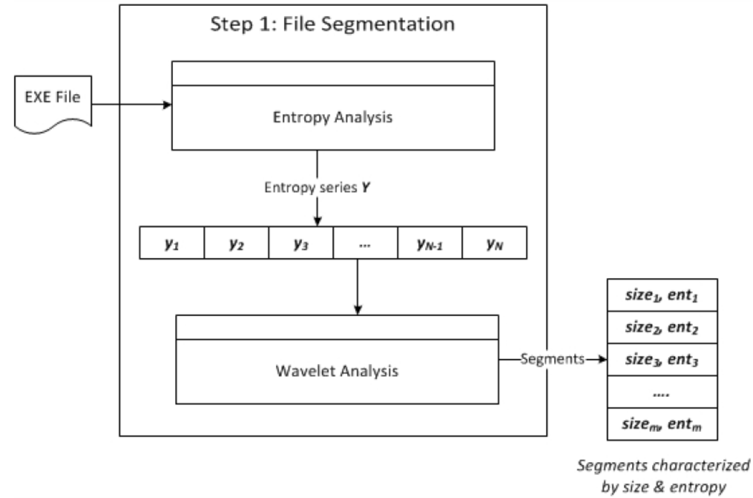


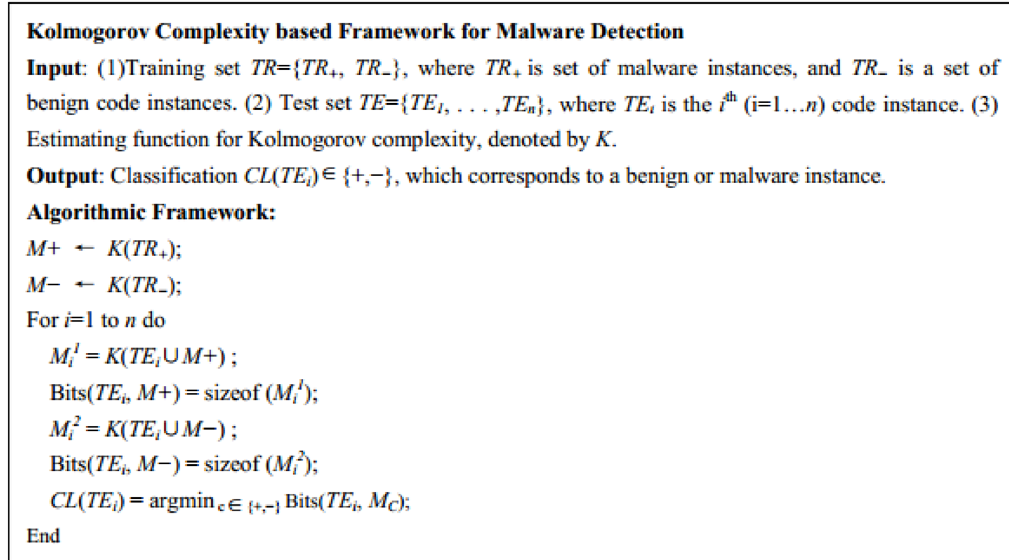
Figure 2: File Segmentation

even detect malware with large amounts of dead code insertion, mitigating a weakness of the Hidden Markov Model approach. However, results were less than ideal for the NGVCK metamorphic family as it proved particularly challenging for this technique. The authors in [3] attributed this phenomenon to the variation in file sizes of the NGVCK viruses and its effect on the sequence comparison process.

2.2.3 Compression-based Classification

Previous research has also been done involving compression in the development of malware detection methods. For example, the paper [12] presents a malware detection framework based on Kolmogorov complexity. Kolmogorov complexity, an information measurement, is the length of the minimal description of a specific string s. Using this measure in a malware detection framework is based on the following principal: given two strings, the more similar they are to each other, the more they can be compressed when concatenated together as opposed to being compressed separately. Treating the byte sequence of a file as a string, an unknown

string can be compared with several known strings. Each known string represents a different class of file, and whichever known string the unknown string compresses best with can be considered the closest in file similarity. The detection framework, described in [12], is shown in Figure 3.



Kolmogorov Complexity based Framework for Malware Detection

Input: (1)Training set $TR=\{TR_+, TR_-\}$, where TR_+ is set of malware instances, and TR_- is a set of benign code instances. (2) Test set $TE=\{TE_1, \ldots, TE_n\}$, where TE_i is the ith (i=1…n) code instance. (3) Estimating function for Kolmogorov complexity, denoted by K.

Output: Classification $CL(TE_i) \in \{+,-\}$, which corresponds to a benign or malware instance.

Algorithmic Framework:

$M+ \leftarrow K(TR_+)$;

$M- \leftarrow K(TR_-)$;

For i=1 to n do

$M_i^1 = K(TE_i \cup M+)$;

Bits(TE_i, $M+$) = sizeof (M_i^1);

$M_i^2 = K(TE_i \cup M-)$;

Bits(TE_i, $M-$) = sizeof (M_i^2);

$CL(TE_i)$ = argmin$_{c \in \{+,-\}}$ Bits(TE_i, M_C);

End

Figure 3: Kolmogorov Complexity Detection Framework

This detection framework provided a high rate of success. However, a drawback mentioned in [12] was that the amount of memory usage increases exponentially as the size of the malware code increases and thus could be a problem in some cases where the malware instances are too large.

Another paper [36] describes a compression-based classification technique and presents a detection framework utilizing a learning engine to train on sets of malware and benign code. By using prediction by partial matching (PPM), an adaptive data compression model, two compression models are built with one representing the malware code and the other representing benign code. For each new file that needs to be classified, the average number of bits required to encode the file is computed

using the two compression models. The file is then classified by determining which model gives a greater compression rate. The algorithm is shown in Figure 4.

```
Algorithm PPM Classifier
Input: Training set T = T+ ∪ T−, test set
       P = {X1, ..., Xn}, and the order of the
       Markov model in PPM, k
Output: Classification c(Xi) ∈ {+, −} of
        Xi ∈ P, for i = 1, ..., n.
M+ ← CreatePPM(T+);
M− ← CreatePPM(T−);
forall X ∈ P do
  | Bits(X, M+) = ComputeBits(X, M+) ;
  | Bits(X, M−) = ComputeBits(X, M−);
  | c(X) = arg min_{c∈{+,−}} Bits(X, Mc);
end
```

Figure 4: PPM-based Classification

The results in [36] appeared promising and it was noted that many techniques that malware writers use to avoid detection by signature based scanning are largely ineffective against compression-based analysis.

CHAPTER 3

Design and Implementation

The file similarity method presented in this paper is primarily derived from [3]. Unlike in [30], for example, we examine the raw bytes of a file without the need for code disassembly. We deviate from [3] by considering compression ratios as an alternative measurement of entropy. Finally, our technique differs from [12, 36] in that while we use compression to analyze files, we determine final similarity through a technique based on Levenshtein distance.

3.1 Forming File Segments

3.1.1 Splitting a File into Byte Windows

The first step in our technique involves splitting files into windows. We consider a window to be a string of consecutive bytes and each window should contain the same number of bytes. When dividing a file into a series of windows, the windows should overlap some amount since we are treating a file as a single stream of continuous data. Therefore, we shift, or slide, some amount of bytes before allocating bytes to the next window. These window sizes and window slide sizes are determined experimentally in order to achieve optimal definition of a given family of metamoprhic malware. Consider Figure 5 which shows a hexdump of a sample file containing 103 bytes.

As an example, if we set the window size to 10 bytes and the window slide size to 5 bytes, then the first window would be as shown in Figure 6 with the second window being shown in Figure 7. Subsequent windows are determined in the same way. If the final window contains less than window size bytes, null bytes are

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  c0 0a 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  a0 5c 00 00 00 00 00 00  |@........\......|
00000030  00 00 00 00 40 00 38 00  08 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  c0 01 00 00 00 00 00                              |.......|
00000067
```

Figure 5: Hexdump of sample file with addresses(Left), bytes in hex representation(Center), bytes in ascii representation(Right)

appended.

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  c0 0a 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  a0 5c 00 00 00 00 00 00  |@........\......|
00000030  00 00 00 00 40 00 38 00  08 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  c0 01 00 00 00 00 00                              |.......|
00000067
```

Figure 6: Window 1 of sample file

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  c0 0a 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  a0 5c 00 00 00 00 00 00  |@........\......|
00000030  00 00 00 00 40 00 38 00  08 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  c0 01 00 00 00 00 00                              |.......|
00000067
```

Figure 7: Window 2 of sample file

Although in this example our window size is very small, window sizes must normally be relatively large in order for compression analysis to derive any significant information from them. However, at the same time, the window sizes cannot be made too large either as it would allow attackers to mask the unique portions of their malware.

3.1.2 Window Compression Ratios

Given our series of windows, we then calculate the compression ratio of each one. The key notion is that we expect windows containing low entropy data to produce high compression ratios and windows containing high entropy data to

produce low compression ratios. Therefore, the series of compression ratio values gives insight into the underlying structure of the file without ever needing to inspect the actual code.

In our implementation, we use the software application, gzip, to calculate the compression ratios. Gzip utilizes Lempel-Ziv (LZ77) coding as its main algorithm [15]. Implementations of LZ77 compress data by replacing repeated instances of data with references to their earlier occurrences in the data stream. Therefore, by compressing the data in these windows, we are essentially measuring the distribution of unique byte sequences in each window. Figure 8 illustrates a plot of compression ratios derived from an example file.

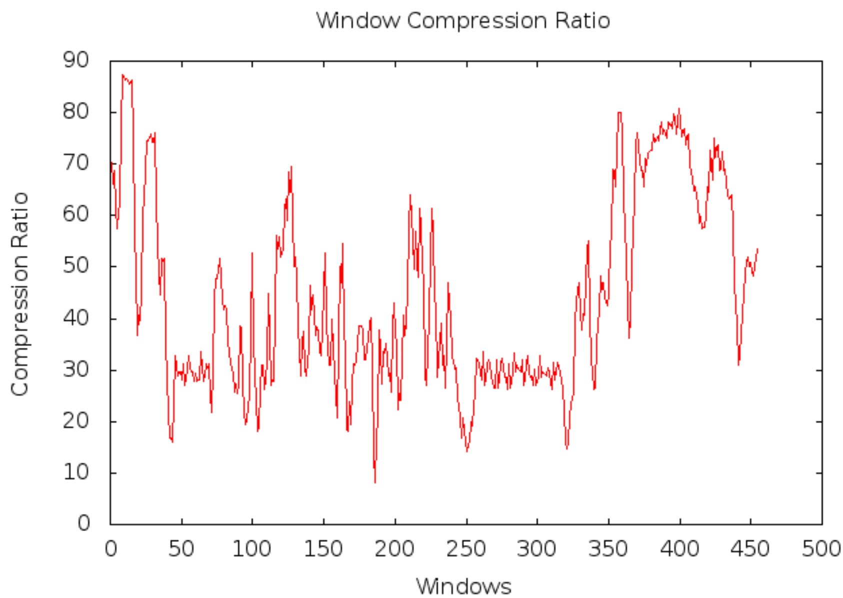


Figure 8: Compression ratios plot of sample file

3.1.3 Wavelet Transform Analysis

The next step involves applying a wavelet transform to our series of compression ratios. Observing Figure 8, we see that the data can be very volatile at times and may make trivial differences seem overly significant when comparing two sets of plots. By applying a wavelet transform, our data is smoothed, especially in places where a high frequency of variation occurs.

Although there are several different wavelet transforms, we choose to use the Discrete Haar Wavelet Transform in our implementation. This approach follows previous work [3, 22]. The Haar Transform is both simple and efficient. If ideal results are achieved by using the simplest wavelet transform, then using more complicated transforms would only incur a performance penalty with little to no benefit.

Suppose we are provided with N values,

$$\mathbf{x} = (x_1, x_2, ..., x_N) \quad \text{where } N \text{ is even.} \tag{1}$$

Let s_k and d_k be defined as,

$$s_k = \frac{x_{2k-1} + x_{2k}}{2}, \quad \text{where } k = 1, ..., N/2 \tag{2}$$

$$d_k = \frac{x_{2k} - x_{2k-1}}{2}, \quad \text{where } k = 1, ..., N/2 \tag{3}$$

Then the Discrete Haar Wavelet Transform can be defined as the following transformation [14],

$$\mathbf{x} = (x_1, ..., x_N) \rightarrow (\mathbf{x} \,|\, \mathbf{d}) = (s_1, ..., s_{N/2} \,|\, d_1, ..., d_{N/2}) \tag{4}$$

The set of values s_k are also known as pair-wise averages. By recursively applying the Discrete Haar Wavelet Transform on the pair-wise averages, we can

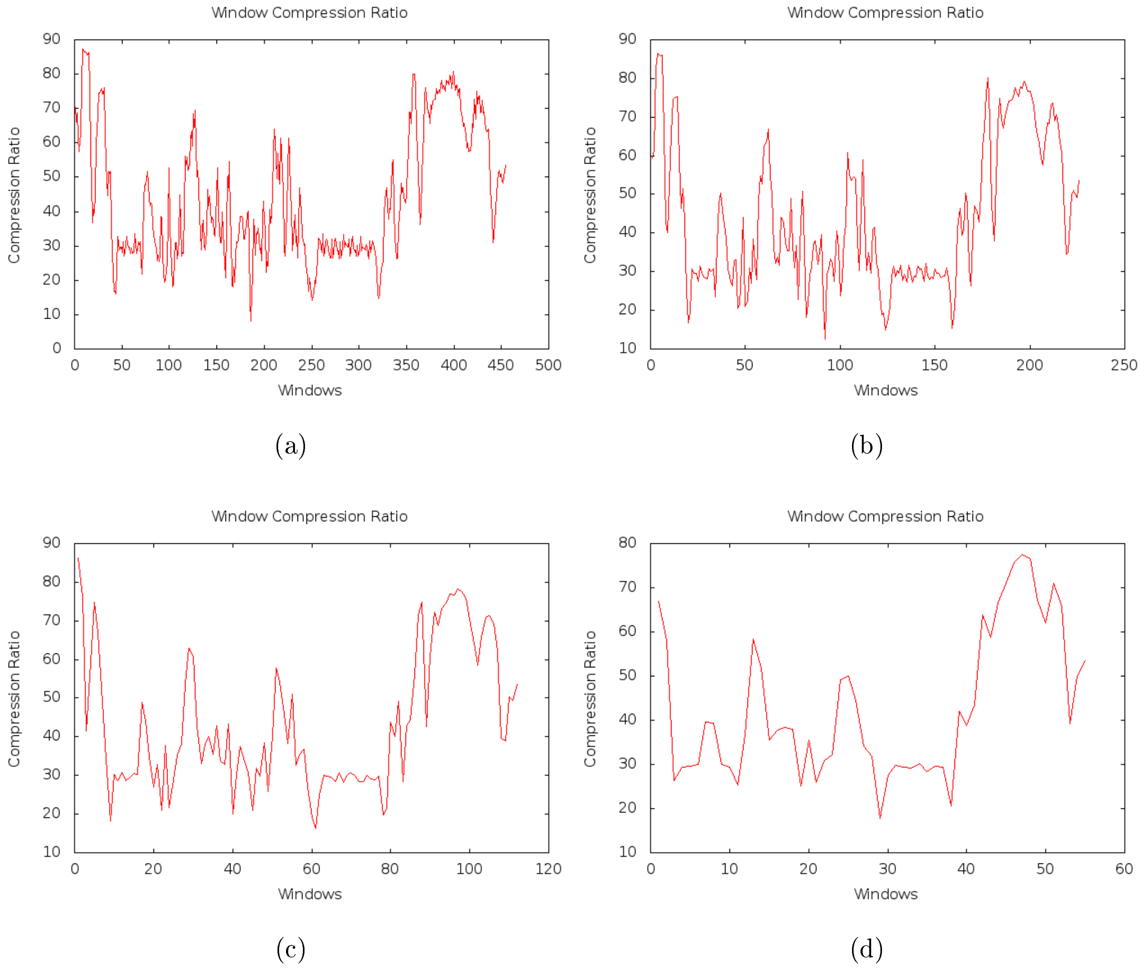


Figure 9: Discrete wavelet transform using (a) 0 iterations, (b) 1 iteration, (c) 2 iterations, (d) 3 iterations

perform an arbitrary number of iterations of the transform. However, the transform can only be applied to number sets that contain even amounts of values. Therefore, for cases where our set contains an odd amount, we pad the set with a copy of the last value to best preserve the original data. Figure 9 demonstrates the transform's effect on the data up to three iterations.

Table 1: Resulting File Segments

Segment #	Segment Length	Segment Value
1	1	0.820
2	1	0.640
3	3	0.897
4	2	0.575
5	3	0.903

3.1.4 File Segment Creation

Our next goal is to form file segments from the transformed compression ratios. To do this, we first set a threshold that determines what is considered high entropy and low entropy. In our implementation, our threshold is set to a ratio of 0.65 such that compression ratio values greater than or equal to 0.65 are considered to indicate low entropy and ratios below 0.65 are considered to indicate high entropy. We decided to use 0.65 after a calibration experiment described in Section 4.4.4. Each segment boundary is then made when two adjacent values are on opposite sides of the threshold.

As a result, each segment has both a length and a value associated with it. The segment length is the number of compression ratio values that contribute to that particular segment and the segment value is the mean of the associated ratios.

As an example, suppose our resulting wavelet transformed values are

$$0.82, 0.64, 0.79, 0.90, 1.00, 0.60, 0.55, 0.93, 0.88, 0.90$$

Using a threshold of 0.65, the resulting segments are shown in Table 1.

3.2 Sequence comparison

The resulting sequence of segments becomes the representation of that file. This is useful because we now have reduced the problem of file similarity to a problem of sequence comparison. We do the comparison using an algorithm based on the Levenshtein distance. The resulting distance between the two sequences is then used to determine their corresponding file similarity. This technique is derived from [3, 22].

3.2.1 Levenshtein distance

Levenshtein distance, or edit distance, is a string metric for measuring how much two sequences differ from each other [26]. More precisely, the Levenshtein distance between two sequences a and b is the minimum number of insertions, deletions, and substitutions of elements required to transform a into b [10]. The smaller the number of operations needed to perform the transformation, the more we consider the first sequence to be similar to the second.

The following example compares the string "books" to "broom" and calculates the distance between them assuming a cost of 1 for each insertion, deletion, and substitution.

1. books → b**r**ooks (insertion of "r")
2. broo**k**s → broo**m**s (substitution of "k" for "m")
3. broom**s** → broom (deletion of "s")

Since three operations are the minimum number of edits required to transform "books" to "broom", the Levenshtein distance of these two strings is considered to

be three. Other sets of three operations can also complete the conversion.

Given two sequences $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_m)$ along with a defined cost function, we can compute the elements of the matrix $D_{(n+1)\times(m+1)} = \{d_{i,j}\}$ using the following recursion which is defined in [3],

$$d_{i,j} = \begin{cases} 0 & \text{if } i = j = 0 \\ d_{0,j-1} + \delta_Y(j) & \text{if } i = 0 \text{ and } j > 0 \\ d_{i-1,0} + \delta_X(i) & \text{if } i > 0 \text{ and } j = 0 \\ d_{i-1,j-1} & \text{if } x_i = y_j \\ \min \begin{cases} d_{i,j-1} + \delta_Y(j) \\ d_{i-1,j} + \delta_X(i) \\ d_{i-1,j-1} + \delta_{X,Y}(i,j) \end{cases} & \text{if } x_i \neq y_j \end{cases} \tag{5}$$

where $\delta_Y(j)$ is the cost of insertions, $\delta_X(i)$ is the cost of deletions, and $\delta_{X,Y}(i,j)$ is the cost of substitutions.

Using the cost function $\delta_Y = \delta_X = \delta_{X,Y} = 1$ in conjunction with (5) to calculate the Levenshtein distance in the previous example, we obtain the matrix shown in Table 2 where $d_{n,m}$ provides the final distance score.

3.2.2 Sequence alignment

Let X and Y represent two files under analysis for similarity calculation. We obtain their respective segments x_i for $i = 1, 2, ..., n$ and y_j for $j = 1, 2, ..., m$ using the segmentation process detailed in Section 3.1. We then apply the following cost function taken from [3, 22] to account for size differences,

$$\text{cost}_\sigma(x_i, y_j) = \frac{|\sigma(x_i) - \sigma(y_j)|}{\sigma(x_i) + \sigma(y_j)}, \tag{6}$$

Table 2: Edit matrix for strings "books" and "broom"

		b	**o**	**o**	**k**	**s**
	0	1	2	3	4	5
b	1	0	1	2	3	4
r	2	1	1	2	3	4
o	3	2	1	1	2	3
o	4	3	2	1	2	3
m	5	4	3	2	2	3

where $\sigma(x_i)$ is the size of segment x_i and $\sigma(y_j)$ is the size of segment y_j. The range of possible values for this cost function is between 0 and 1 inclusive. With respect to compression ratio differences, we again utilize the following cost function from [3, 22],

$$\text{cost}_\epsilon(x_i, y_j) = \frac{1}{1 + e^{-4\cdot|\epsilon(x_i)-\epsilon(y_j)|+6.5}} - 0.001501, \tag{7}$$

where $\epsilon(x_i)$ and $\epsilon(y_j)$ are the compression ratios of the corresponding segments. The constants, 6.5 and 0.001501, in (7) bound cost_ϵ between 0 and 1 [3]. Combining equations (6) and (7) results in the final version of the cost function,

$$\text{cost}(x_i, y_j) = c_\sigma \cdot \text{cost}_\sigma(x_i, y_j) + c_\epsilon \cdot \text{cost}_\epsilon(x_i, y_j), \tag{8}$$

where c_σ and c_ϵ are constants used weight the size and entropy costs appropriately. Section 4.4.1 defines the values for these constants.

This cost function is then applied to the Levenshtein distance based sequence alignment algorithm. We use dynamic programming to create a two-dimensional array similar to Table 2 and retrieve the last element as our final cost calculation between the two segment sequences. A good reference on dynamic programming can be found in [9]. In order to utilize equation (5) to calculate the elements of the

array, we set $\tau = 0.3$ and define the following functions,

$$\begin{aligned} \delta_Y(j) &= \tau \, \log \, \sigma(y_{j-1}) \\ \delta_X(i) &= \tau \, \log \, \sigma(x_{i-1}) \\ \delta_{X,Y}(i,j) &= \text{cost}(x_{i-1}, y_{j-1}) \cdot \log\left(\frac{\sigma(x_{i-1}) + \sigma(y_{j-1})}{2}\right) \end{aligned}, \tag{9}$$

which were derived in [3].

3.2.3 Similarity calculation

After calculating the edit distance using equation (5) with penalty functions (9), the similarity between files X and Y is determined by applying the following equation,

$$\text{similarity} = 100\left(1 - \frac{d_{n,m}}{\text{cost}_{max}}\right), \tag{10}$$

where cost_{max} is the worst-case penalty. As in [3, 22], this penalty is calculated in a special way and is defined as,

$$\text{cost}_{max} = d'_{0,m} + d'_{n,0} \tag{11}$$

where $d'_{0,m}$ and $d'_{n,0}$ are determined by using equation (5) with the penalty functions,

$$\begin{aligned} \delta'_Y(j) &= \delta_Y(j) \\ \delta'_X(i) &= \delta_X(i) \\ \delta'_{X,Y}(i,j) &= 2\tau(\log \, \sigma(x_{i-1}) + \log \, \sigma(y_{j-1})). \end{aligned} \tag{12}$$

CHAPTER 4

Experiments and Results Analysis

To evaluate how well the compression-based detection technique described in this paper fares against metamorphism, we first establish representative data sets with which to apply our technique to. We then define a common metric to quantify the technique's effectiveness and provide our findings. Finally, we provide the reasoning used to determine the values for the technique's critical parameters.

4.1 Test Data

4.1.1 Second Generation Virus Generator

In our first set of data, we utilize 50 virus files, which were retrieved from [31], that were generated by the Second Generation Virus Generator (G2). G2 viruses are one of several well-known metamorphic families. The benign files we use to compare against the G2 viruses are 16 specific Cygwin utility files [11] chosen for their representation as non-virus files in previous papers such as [3, 21, 30]. The exact files included in the benign data set are shown in Table 3.

Table 3: Cygwin Utility Files Used For Benign Data Set

1	ascii.exe	9	mkshortcut.exe
2	banner.exe	10	msgtool.exe
3	conv.exe	11	putclip.exe
4	cygdrop.exe	12	readshortcut.exe
5	cygstart.exe	13	realpath.exe
6	dump.exe	14	semstat.exe
7	getclip.exe	15	semtool.exe
8	lpr.exe	16	shmtool.exe

4.1.2 MWOR

In our second set of data, we utilize metamorphic worms that were introduced in [23]. These worms, also known as MWOR, were designed to defeat statistical based classifiers, particularly techniques that rely on opcode based analysis such as the Hidden Markov Model detection technique described in Section 2.2.1. One of the defining features of the MWOR worm generator is its ability to insert arbitrary amounts of dead code into the generated malicious files. By copying code from various benign files and placing it within the MWOR worms in a non-executable manner, MWOR worms completely retain their functionality while being able to significantly alter their statistical properties. The amount of dead code inserted into the worm is denoted and quantified as a padding ratio. A padding ratio of 4, for example, denotes that in the overall file there is 4 times as much dead code as actual worm code. We choose 700 MWOR files divided evenly among padding ratios of 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, and 4.0 in our experiments. Each set of 100 worm variants containing the same padding ratio are grouped together and tested separately effectively creating 7 distinct data sets.

For our benign data set, we depart from the Cygwin utility files here since the MWOR generator produces Linux based worms whereas the Cygwin files are Windows executables. We instead choose 30 representative standard Linux binaries shown in Table 4 to better compare against the MWOR worms. The benign data set and MWOR data set we use here contain the exact same files experimented with in [3].

Table 4: Linux Binaries Used For Benign Data Set

1	/bin/date	11	/usr/bin/as	21	/usr/bin/nm
2	/bin/dmesg	12	/usr/bin/at	22	/usr/bin/nm-tool
3	/bin/grep	13	/usr/bin/dig	23	/usr/bin/objdump
4	/bin/kill	14	/usr/bin/file	24	/usr/bin/oclock
5	/bin/mknod	15	/usr/bin/funzip	25	/usr/bin/readelf
6	/bin/mount	16	/usr/bin/killall	26	/usr/bin/rpl8
7	/bin/rm	17	/usr/bin/last	27	/usr/bin/shuf
8	/bin/sleep	18	/usr/bin/ld	28	/usr/bin/size
9	/bin/sync	19	/usr/bin/msgcat	29	/usr/bin/strip
10	/bin/touch	20	/usr/bin/namei	30	/usr/bin/sum

4.1.3 Next Generation Virus Construction Kit

In our final set of data, we utilize 50 virus files generated by the Next Generation Virus Construction Kit (NGVCK) [31]. NGVCK viruses attempt to infect Win32 PE-Executables and posses both encryption and anti-debugging capabilities [32]. Since these viruses target Windows machines, we compare them against the 16 Cygwin utility files specified in Table 3. NGVCK viruses are considered highly metamorphic and have been used in previous research such as [3, 30].

4.2 Receiver Operating Characteristic

To evaluate our results, we choose to use Receiver Operating Characteristic (ROC) curves. An ROC curve serves as a graphical representation of a binary classifier system and plots the fraction of true positives against the fraction of false positives at various thresholds. See Figure 10, which was taken from [27], for an example plot of superimposed ROC curves. By then calculating the area under the curve (AUC), we have a single metric by which to compare our results with other

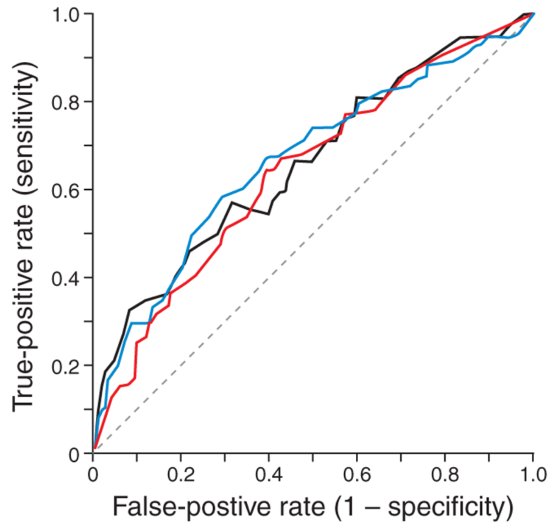


Figure 10: Example ROC Curve

experiments. If the AUC is near 0.5, it indicates that the evaluated classifier has poor performance. If the AUC is near 1.0, the classifier is performing very well. An AUC of exactly 1.0 represents optimal performance.

4.3 Test Results

For each data set, all unique pairs of malicious files and benign files are compared. Unique pairs of malicious files within the same data set are also compared. Ideally, comparing a malicious file against a benign file should produce low similarity while comparing a malicious file against a variant within the same metamorphic family should produce high similarity.

In our experiments with the G2 viruses, we obtain the results shown in Figure 11. Here, we use a window size of 128 bytes and a window slide size of 64 bytes. We can see a clear separation between the similarity scores of *virus versus virus* and *virus versus benign* pairs. This indicates that our technique is able to clearly distinguish between the G2 generated viruses and the benign files.

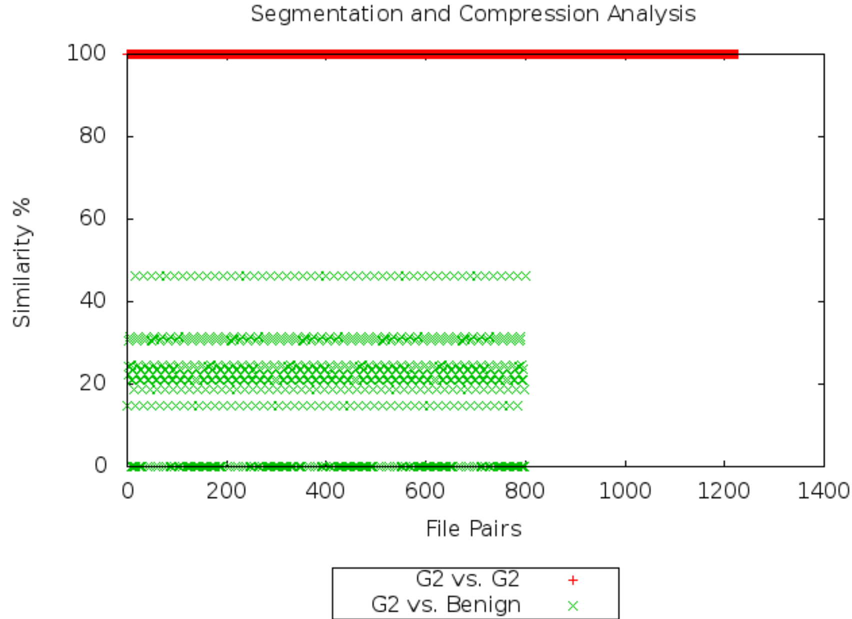


Figure 11: G2 similarity

We then begin testing against the MWOR generated worms, beginning with those that contain a 0.5 padding ratio, and obtain the results in Figure 12. We use a window size of 256 bytes and window slide size of 64 bytes. Looking at the results, we can again set a clear threshold such that there is ideal separation between the *worm versus worm* pairs and *worm versus benign* pairs. Repeating our experiments with padding ratios 1.0, 1.5, 2.0, 2.5, 3.0, and 4.0, we find that we are able to obtain full separation at every level. The results for these padding ratios can be found in Appendix A.

Finally, we experiment with the NGVCK generated viruses and produce the results in Figure 13. Using a window size of 128 bytes and window slide size of 2 bytes, we again obtain full separation with *virus versus virus* pairs producing higher similarity and *virus versus benign* pairs producing lower similarity. Although our technique produced no false positives and no false negatives here, the NGVCK

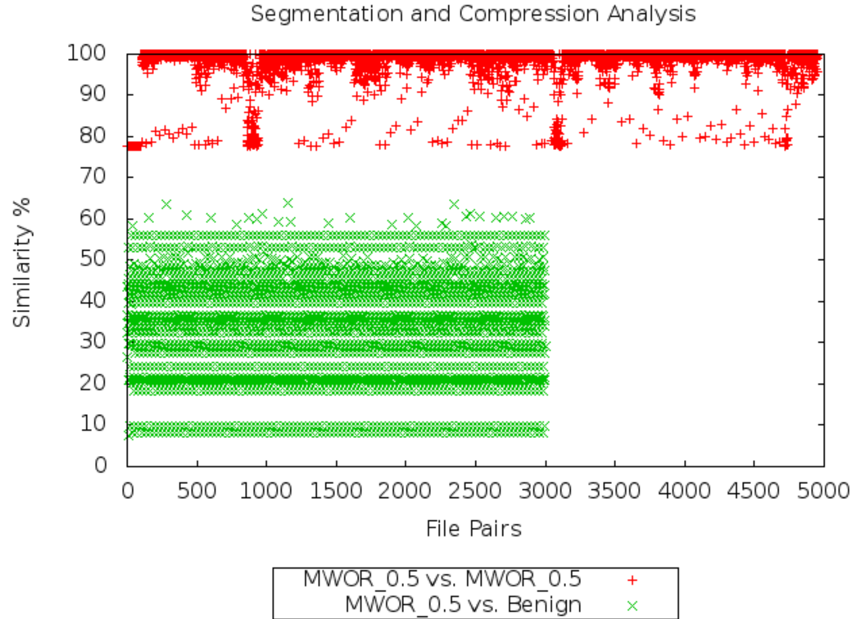


Figure 12: MWOR(0.5 padding) similarity

generated viruses were by far the most difficult malware to clearly detect in our experiments. This is likely attributed to the fact that the NGVCK generator contains encryption capabilities [32]. Code that is obfuscated through encryption would produce byte distributions that are much less compressible, thus directly inhibiting our ability to extract information during our segmentation step. However, it appears that despite these challenges, our compression-based analysis is still able to identify sufficient unique characteristics of the NGVCK files.

Overall, our compression-based analysis technique is able to fully distinguish between all sets of metamorphic malware tested and their benign file counterparts. Ideal separation is achieved in all cases and is illustrated in Figure 14 where ROC curves are drawn for each set.

We obtain an AUC of 1.0 for all curves. Previous research utilizing structural entropy analysis also achieved an AUC of 1.0 for the G2 and MWOR generated

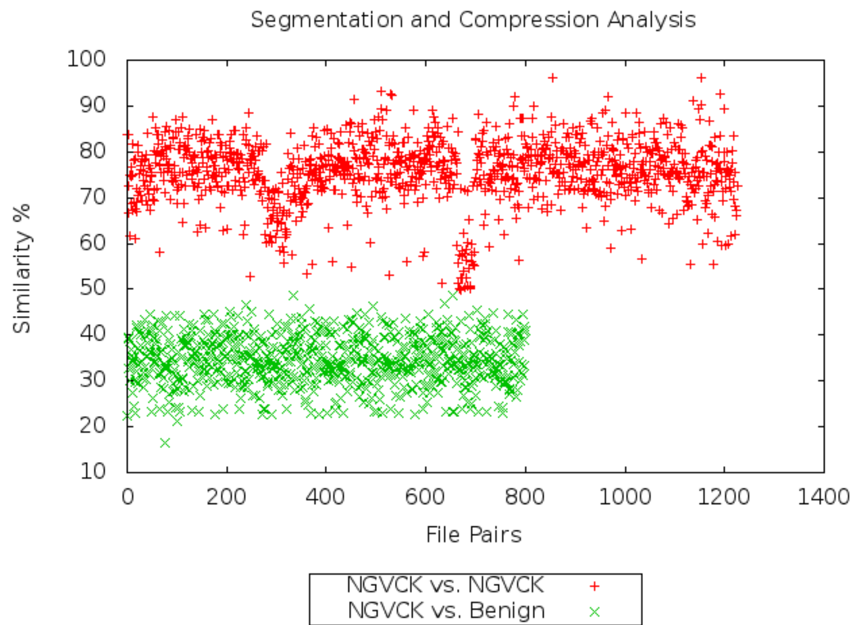


Figure 13: NGVCK similarity

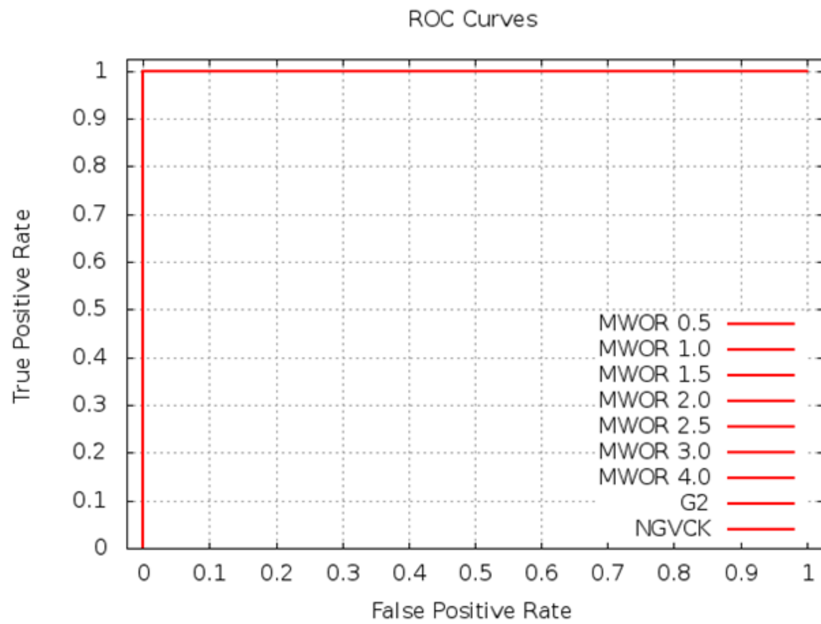


Figure 14: ROC Curve Results

malware [3]. However, they were unable to obtain ideal separation for the NGVCK case which we have here. Table 5 summarizes our findings.

Table 5: Results Summary

Metamorphic Family	Window Size	Window Slide Size	AUC
G2	128	64	1.00
NGVCK	128	2	1.00
MWOR(0.5 padding)	256	64	1.00
MWOR(1.0 padding)	256	64	1.00
MWOR(1.5 padding)	256	64	1.00
MWOR(2.0 padding)	256	64	1.00
MWOR(2.5 padding)	256	64	1.00
MWOR(3.0 padding)	256	64	1.00
MWOR(4.0 padding)	256	64	1.00

4.4 Setting Parameters

Although compression-based analysis can identify unique features across metamorphic malware variants, there are several critical parameters in the technique that must be correctly calibrated in order to achieve ideal detection.

4.4.1 Cost Constants

Recall cost function (8) that is used during the sequence comparison step where c_σ and c_ϵ are adjustable parameters in order to assign appropriate weights to our two components. In all our experiments, we chose $c_\sigma = 1.6$ and $c_\epsilon = 0.4$. Previous research [3] has experimentally found these values to be ideal and so we use these values here as well. Likewise, we set $\tau = 0.3$ in (9), which is the same value used for τ in [3, 22].

4.4.2 Window Slide Size

Another critical parameter is the window slide size. Recall that the window slide size is the number of bytes we shift, or slide, before analyzing the next window

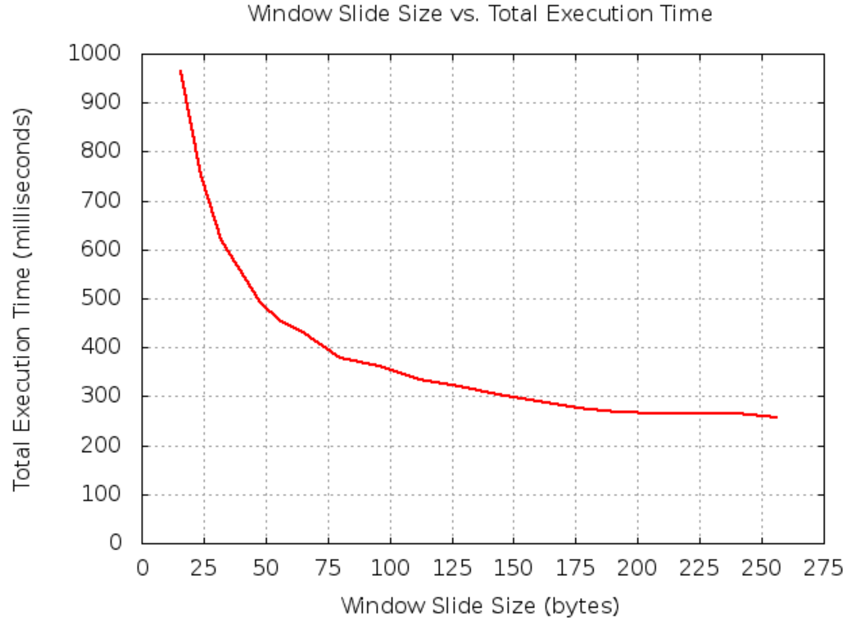


Figure 15: Window Slide Size vs. Total Execution Time

in that file. Sensibly, the minimum possible window slide size is 1 byte and the maximum possible window slide size is the size of the window itself. However, there is a trade-off. If the window slide size is set too large, too much information is missed during the calculation of compression ratios and the accuracy of detection falls. On the other hand, if the window slide size is set too small, a performance penalty is incurred. The number of windows formed, and hence the number of compression ratio calculations that must be done (the most expensive part of the technique), should be minimized while still retaining ideal detection accuracy. In order to illustrate the window slide size's effect on performance, we vary the window slide size while keeping all other parameters constant and plot the resulting execution times. Figure 15 shows the case where, using our technique, similarity is calculated between two MWOR variants both with padding ratio 0.5. From this plot, it can be seen that execution time is inversely proportional to the window slide size.

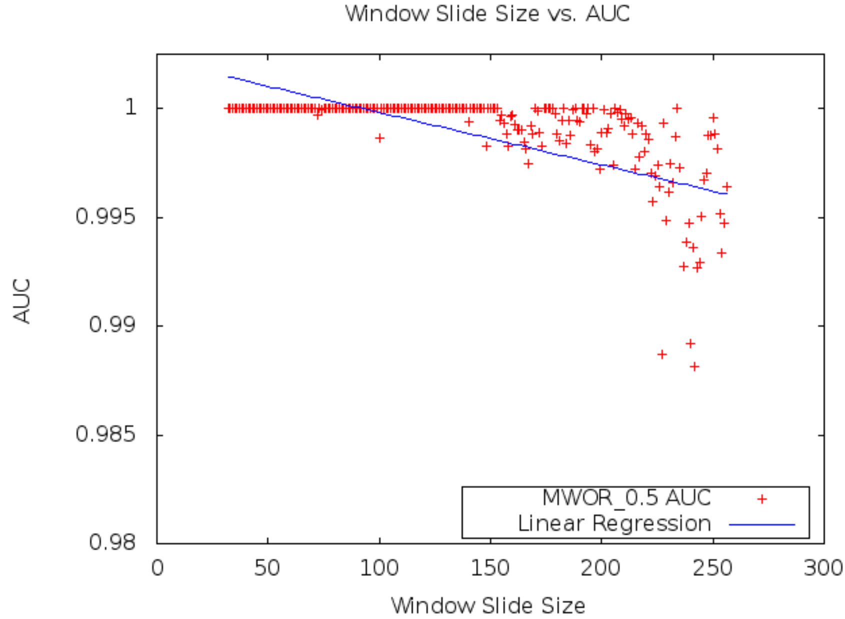


Figure 16: Window Slide Size vs. AUC

In order to illustrate the window slide size's effect on detection accuracy, we vary the window slide size used in our technique while keeping all other parameters constant and plot the resulting AUC. Figure 16 shows the results for the MWOR case with padding ratio 0.5 where the window slide size is again measured in bytes. There will always be some oscillation in the AUC due to the structural variation of the file under analysis. However, we can identify a downward trend in the AUC, and hence the accuracy of detection, as the window slide size increases. The results have also been plotted for the NGVCK and can be seen in Figure 17. The decrease in detection accuracy is more drastic for the NGVCK case. However, for the G2 case, detection accuracy remains optimal throughout and the results can be seen in Figure 18. These results are again likely attributed to the fact that G2 is the least effective metamorphic engine tested in our experiments while NGVCK is the most effective. We use a window slide size of 2 bytes for the NGVCK malware and a window slide size of 64 bytes for the G2 and MWOR malware in all other

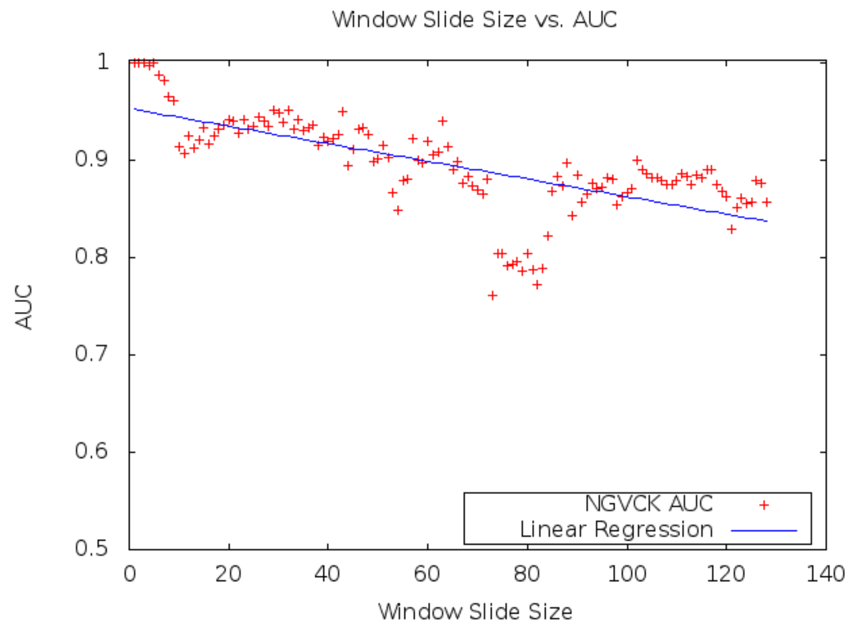


Figure 17: Window Slide Size vs. AUC

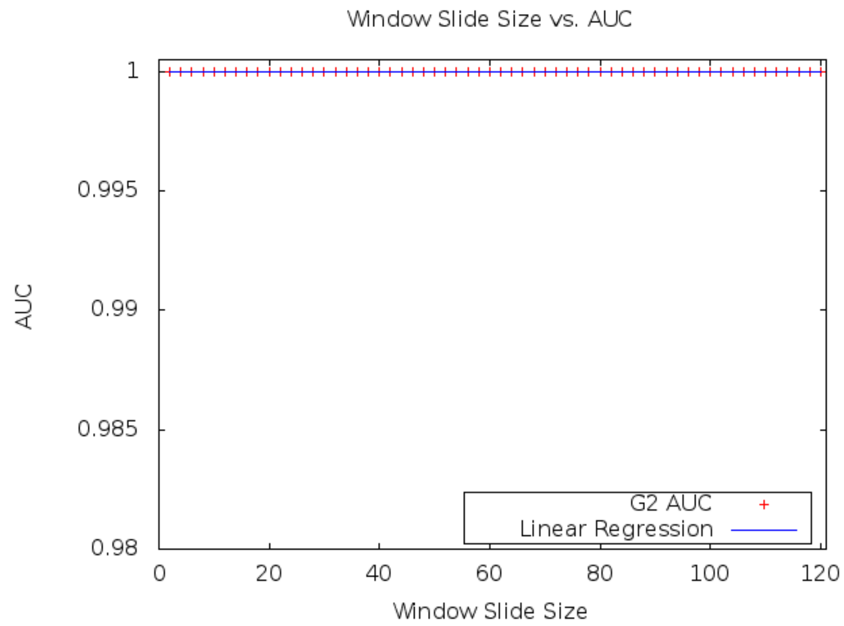


Figure 18: Window Slide Size vs. AUC

experiments based on the results found here.

4.4.3 Recursive Iterations of the Wavelet Transform

We observe another parameter that must be carefully set in order to obtain ideal detection and that is the number of recursive iterations of the wavelet transform. Recall that the purpose of the transform is to mitigate the effect of high frequency changes on the segment formation process. Refer to Figure 9 to see an illustration of this effect. If no transform is applied or too few iterations of the transform are applied, then the high frequency changes could cause an excess of segments to be formed which may cause metamorphic malware variants to appear increasingly dissimilar. However, if too many iterations of the transform are applied, then we lose too much information from the original compression analysis and all segment sequences, malicious and benign, begin to look the same. For all our tests, we found success with setting the number of recursive iterations of the transform to 3. However, other metamorphic families will have to be looked at in a case by case basis. Appendix B shows the effect of varying wavelet transform iterations for MWOR with padding ratio 0.5, Appendix C shows the results for NGVCK, and Appendix D shows the results for G2. For MWOR with padding ratio 0.5 and NGVCK, we tested up to 5 recursive iterations. However, for G2, we can only test up to 3 recursive iterations due to the G2 malware having smaller file sizes.

4.4.4 Compression Ratio Threshold

Setting an appropriate compression ratio threshold is also important in the technique. Recall from Section 3.1.4 that the compression ratio threshold determines what windows we consider to have high entropy and low entropy. In order to calibrate this threshold, we experiment with the MWOR malware with padding ratio 0.5 and vary the threshold while keeping all other parameters

Table 6: Compression Ratio Threshold Calibration

Compression Ratio Threshold	AUC
0.05 or lower	N/A
0.10	0.49154
0.15	0.46793
0.20	0.94633
0.25	0.99985
0.30	0.99952
0.35	0.99358
0.40	0.99992
0.45	0.99999
0.50	0.99934
0.55	0.99814
0.60	1.00000
0.65	1.00000
0.70	1.00000
0.75	1.00000
0.80	1.00000
0.85 or greater	N/A

constant. The results are shown in Table 6.

No byte window produced a compression ratio of 0.85 or greater meaning the files could not be segmented and as a result, no similarity score could be determined for those cases. Similarly, no byte window produced a compression ratio of 0.05 or lower. The compression ratio thresholds between 0.60 and 0.80 produced the best results. The large range of ideal thresholds suggests a somewhat sizable leniency. We chose to use 0.65 in all other experiments based on this calibration.

CHAPTER 5

Conclusion and Future Work

We propose the application of compression-based structural analysis to detect metamorphic malware. We create a binary classification scheme, where we use file compression ratios to develop representative sequences of each file. These representative sequences are then compared against each other using the edit distance algorithm to determine file similarity. After setting a scoring threshold, file similarity scores can be used to classify files as either benign or malicious.

Despite the many intricate techniques utilized in metamorphic engines, our experiments indicate that file compression ratios remain a distinguishing feature across all variants of a given metamorphic malware family. Of all the metamorphic families we tested, the G2 family were the easiest to detect with all G2 variants having 99% similarity or higher with each other and low similarity with all tested benign files. The MWOR metamorphic family displayed higher variability but were still clearly distinguishable from benign files, even at very high padding ratios. The NGVCK metamorphic family proved the most difficult to detect, as expected. Regarded as highly metamorphic and possessing a large array of obfuscation techniques, the NGVCK malware have been analyzed in a lot of previous research. However, very few non-emulation based techniques are able to achieve ideal detection rates.

Overall, we are able to achieve ideal detection, that is, 0% false positives and 0% false negatives, for all metamorphic malware tested. A strength of our technique is that it is applied directly to binary files without the need for expensive

pre-processing steps such as code disassembly. Our technique is primarily based on previous work that utilized structural entropy analysis to detect metamoprhic malware [3]. Although there are several adjustments made here, the core difference is that we propose using compression ratios as an alternative measure of entropy.

A possible limitation of the technique, however, is that it is vulnerable to obfuscation involving heavy use of compression or packing. Since the segmentation step relies on compression ratio calculations, code that has already been compressed or packed may render the technique ineffective. However, previous work has shown that entropy analysis can effectively identify encrypted and packed malware [19]. Therefore, if metamorphic malware use extensive compression or packing for obfuscation to escape detection by our compression-based analysis, it would only make them easier to detect using the aforementioned entropy analysis. For future work, a combination of the two techniques into a single detection framework might be considered.

Additionally, further experimentation with various parameter settings for the technique presented in this paper could prove to be useful. Although we are able to achieve ideal detection, different parameter settings may allow the same level of detection but with improved efficiency. A more detailed look at using different compression methods and sequence comparison algorithms may also prove fruitful. We experimented using Hidden Markov Models as an alternative to the edit distance algorithm in comparing file sequences. However, we were unable to achieve the same level of success using that method.

Another path of future work that may prove interesting would be to apply compression-based analysis to network anomaly detection. It has been shown that measuring nonextensive entropy is an effective method to detect anomalies in

network traffic within an Autonomous System [37]. Given the success shown here using compression ratios as an alternative measure to entropy, it would indicate that compression-based analysis may also prove similarly effective in other domains where entropy is a distinguishing feature.

Finally, future work aimed at finding ways to defeat this technique may also prove worthwhile. For example, the MWOR metamorphic malware make strong use of dead code insertion, but the dead code is taken from parts of legitimate programs in order to defeat op-code based similarity detection techniques. If a careful selection of bytes to manipulate compression ratios are used instead, the MWOR malware may defeat detection by compression-based analysis as well.

LIST OF REFERENCES

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

[2] J. Aycock, *Computer Viruses and Malware*. Springer, 2006.

[3] D. Baysa, R. M. Low, M. Stamp, "Structural Entropy and Metamorphic Malware", *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, 2013.

[4] P. Beaucamps, "Advanced Metamorphic Techniques in Computer Viruses", *International Conference on Computer, Electrical, and Systems Science, and Engineering (CESSE'07)*, 2007.
http://vxheavens.com/lib/pdf/Advanced%20Metamorphic%20Techniques%20in%20Computer%20Viruses.pdf

[5] P. Beaucamps, "Advanced Polymorphic Techniques", *International Journal of Computer Science*, 2(3).
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.4560&rep=rep1&type=pdf

[6] J. Borello, "Code obfuscation techniques for metamorphic viruses", *Journal in Computer Virology*, 4(3):211–220, 2008.

[7] S. Cesare, *Survey in Static Detection of Malware*, 2011.
http://www.foocodechu.com/?q=node/42

[8] M. Christodorescu, S. Jha, "Testing malware detectors", *In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, 34–44, 2004.
http://www.eecs.berkeley.edu/~sseshia/pubdir/oakland05.pdf

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd Edition. The MIT Press, 2009.

[10] G. Cormode, S. Muthukrishnan, "The string edit distance matching problem with moves", *ACM Trans. Algorithms* 3, 1, Article 2, 2007.
http://dimacs.rutgers.edu/~graham/pubs/papers/editmoves.pdf

[11] Cygwin. Cygwin utility files, 2013.
http://www.cygwin.com/

[12] W. Deng, et al, “A Malware Detection Framework Based on Kolmogorov Complexity”, *Journal of Computational Information Systems*, 7(8):2687–2694, 2011.
http://www.jofcis.com/publishedpapers/2011_7_8_2687_2694.pdf

[13] E. Filiol, “Metamorphism, Formal Grammars and Undecidable Code Mutation”, *International Journal of Electrical and Computer Engineering*, 2(1), 2007.
http://www.waset.org/journals/ijece/v2/v2-1-9.pdf

[14] P. Fleet, “The Discrete Haar Wavelet Transformation”, *Joint Mathematical Meetings*, 2007.
http://cam.mathlab.stthomas.edu/wavelets/pdffiles/NewOrleans07/HaarTransform.pdf

[15] gzip, “manpagez: man (manual) pages & more”, 2013.
http://www.manpagez.com/man/1/gzip/

[16] E. Konstantinou, “Metamorphic Virus: Analysis and Detection”, *Technical Report, RHUL-MA-2008-02*, 2008.
http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf

[17] F. Leder, B. Steinbock, P. Martini, “Classification and Detection of Metamorphic Malware using Value Set Analysis”, *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 13(14):39–46, 2009.
http://net.cs.uni-bonn.de/fileadmin/user_upload/leder/metamorphvsa.pdf

[18] D. Lin, “Hunting for Undetectable Metamorphic Viruses”, Master’s Projects, Paper 144, 2009.
http://scholarworks.sjsu.edu/etd_projects/144

[19] R. Lyda, J. Hamrock, “Using Entropy Analysis to Find Encrypted and Packed Malware”, *IEEE Security and Privacy*, 5(2):40–45, March 2007.

[20] L. R. Rabiner, “A tutorial on hidden Markov models and selected applications in speech recognition”, *Proceedings of the IEEE*, 77(2), 1989.
http://www.cs.ucsb.edu/~cs281b/papers/HMMs\%20-\%20Rabiner.pdf

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

[22] I. Sorokin, “Comparing files using structural entropy”, *Journal in Computer Virology*, 7(4):259–265, 2011.

[23] S. Sridhara, M. Stamp, “Metamorphic worm that carries its own morphing engine”, *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, May 2013.

[24] M. Stamp, “A Revealing Introduction to Hidden Markov Models”, 2012. http://cs.sjsu.edu/~stamp/RUA/HMM.pdf

[25] M. Stamp, *Information Security: Principles and Practice*, second edition, Wiley, 2011.

[26] R. A. Wagner, M. J. Fischer, “The String-to-String Correction Problem”, *Journal of the ACM (JACM)*, 21(1):168–173, 1974. http://www.inrg.csie.ntu.edu.tw/algorithm2013/homework/Wagner-74.pdf

[27] D. Warnock, C. Peck, “A roadmap for biomarker qualification”, *Nature Biotechnology*, 28, 444–445, 2010.

[28] R. Warrior, “Guide to improving Polymorphic Engines”, *VX Heavens*. http://vxheaven.org/lib/static/vdat/tumisc17.htm

[29] G. Wicherski, “peHash: A Novel Approach to Fast Malware Clustering”, *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more (LEET'09)*, 2009. https://www.usenix.org/legacy/event/leet09/tech/full_papers/wicherski/wicherski.pdf

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

[31] Virus files, Department of Computer Science, San Jose State University, 2013. http://cs.sjsu.edu/~stamp/viruses/

[32] VX Heaven, Next Generation Virus Construction Kit, 2013. http://vxheaven.org/vx.php?id=tn02

[33] I. You, K. Yim, “Malware obfuscation techniques: A brief survey”, *2010 International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, 297–300, November 2010.

[34] P. Zbitskiy, “Code mutation techniques by means of formal grammars and automatons”, *Journal in Computer Virology*, 5(3):199–207, August 2009.

[35] Q. Zhang, D. Reeves, “MetaAware: Identifying Metamorphic Malware”, *ACSAC*, 411–420, IEEE Computer Society, 2007. http://www.acsac.org/2007/papers/81.pdf

[36] Y. Zhou, M. Inge, “Malware Detection Using Adaptive Data Compression”, *AISec '08 Proceedings of the 1st ACM workshop on Workshop on AISec*, 53–60, 2008.

[37] A. Ziviani, A. Gomes, M. Monsores, P. Rodrigues, "Network Anomaly Detection using Nonextensive Entropy", *IEEE Communications Letters*, 11(12):1034–1036, December 2007.

APPENDIX A

MWOR Results

Figures A.19, A.20, A.21, A.22, A.23, and A.24 show the similarity plots of the MWOR metamorphic family with varying padding ratios. The set of benign files used in each experiment shown here can be found in Table 3 of Section 4.1.1.

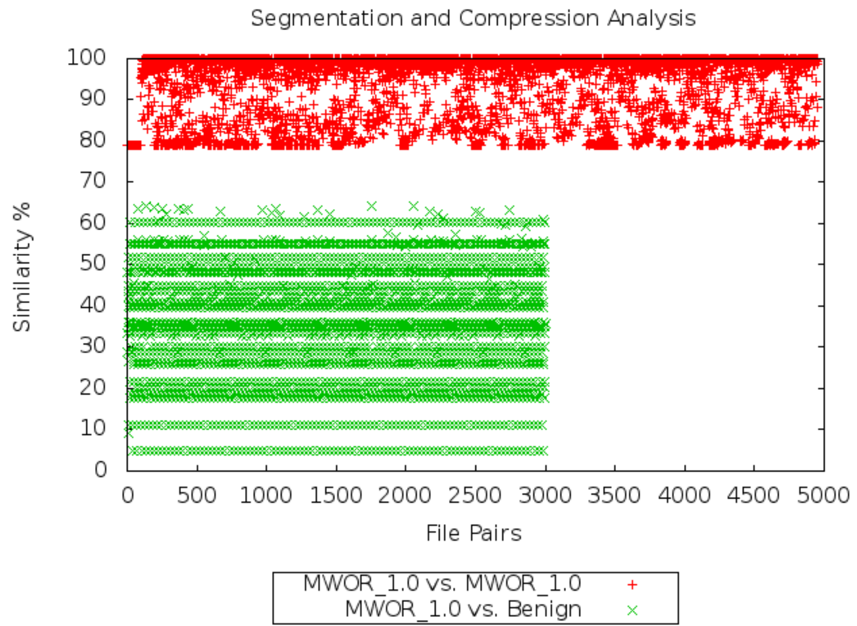


Figure A.19: MWOR(1.0 padding) similarity

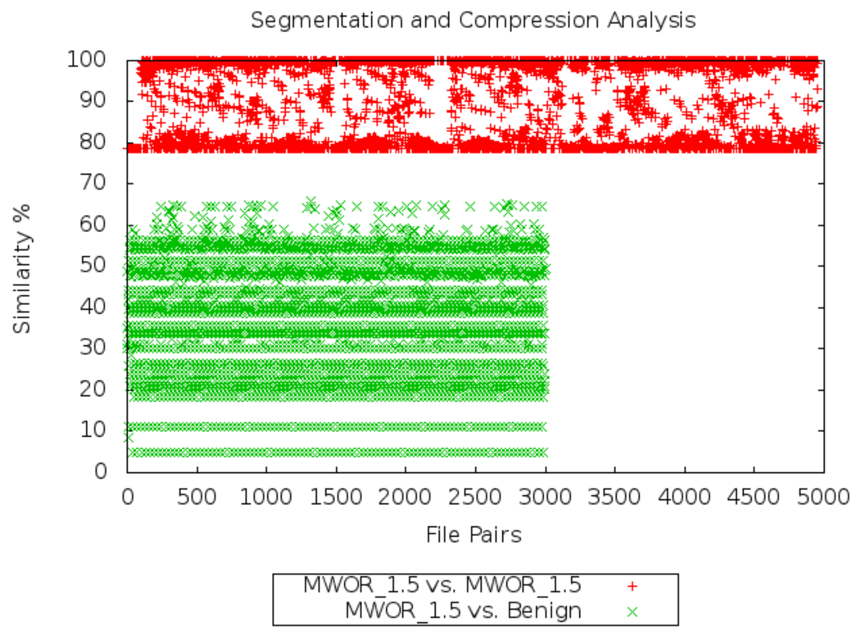


Figure A.20: MWOR(1.5 padding) similarity

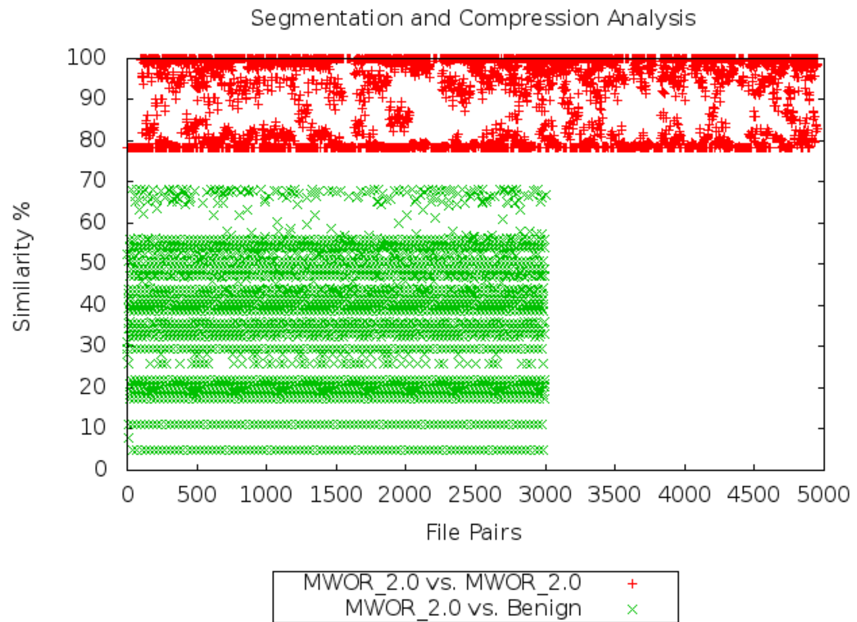


Figure A.21: MWOR(2.0 padding) similarity

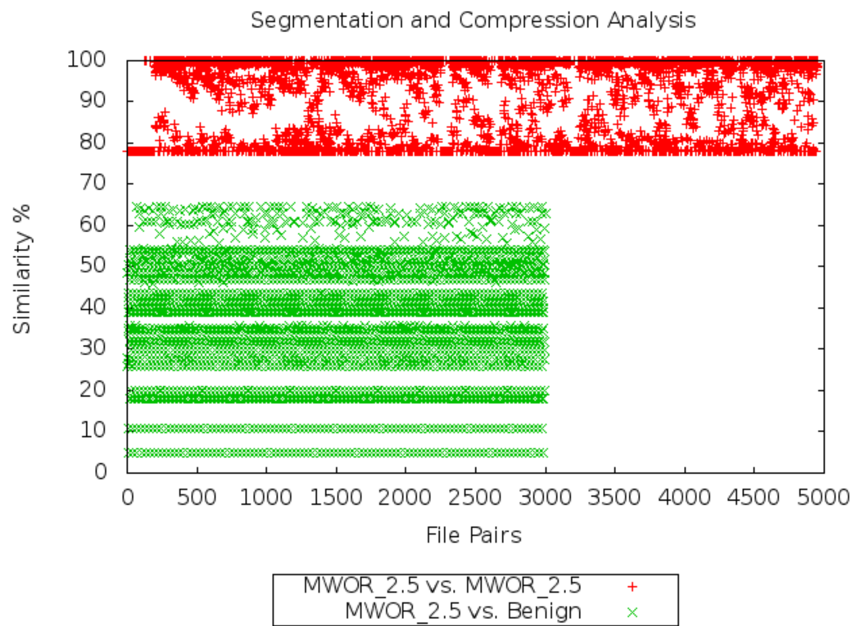


Figure A.22: MWOR(2.5 padding) similarity

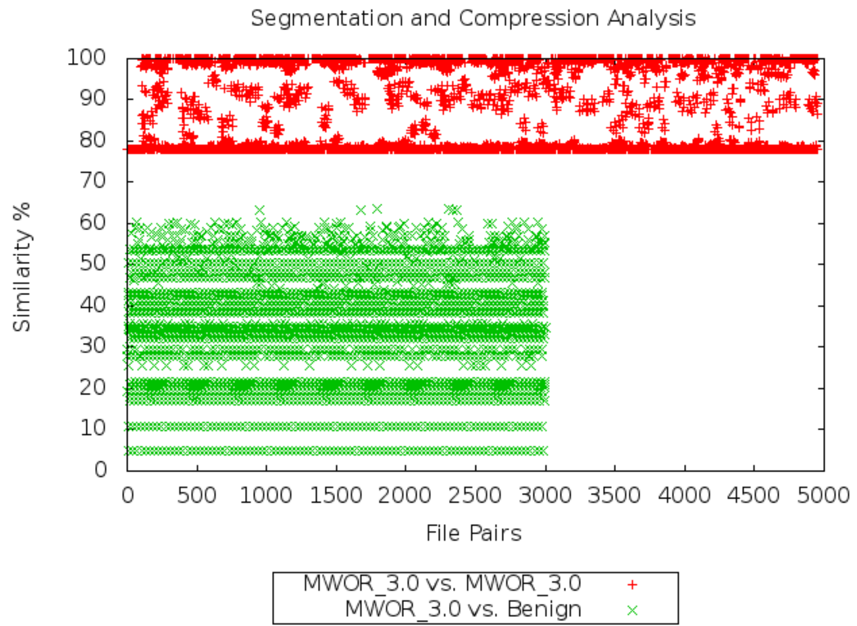


Figure A.23: MWOR(3.0 padding) similarity

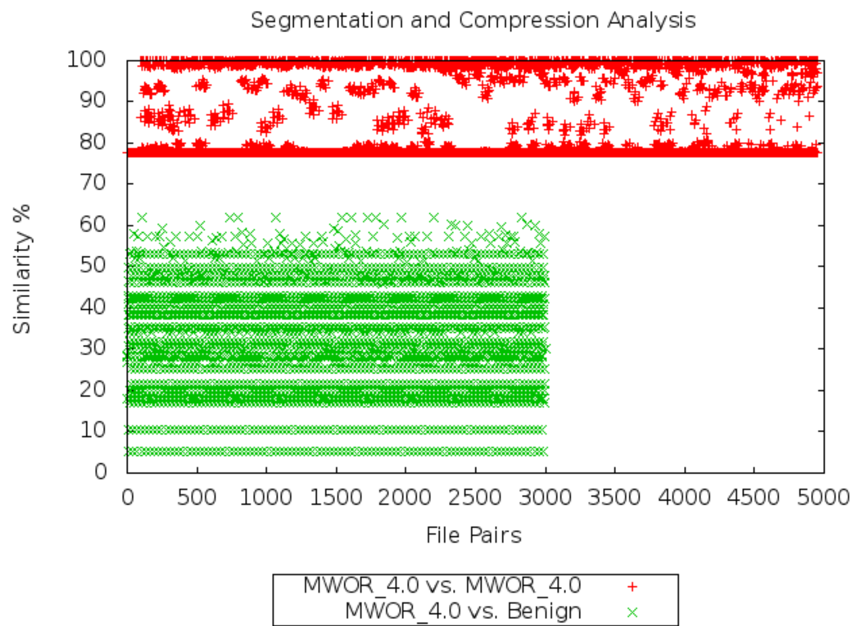


Figure A.24: MWOR(4.0 padding) similarity

APPENDIX B

Wavelet transforms on MWOR (0.5 padding ratio)

The effect of varying wavelet transform iterations for MWOR with padding ratio 0.5 is plotted in Figures B.25, B.26, B.27, B.28, B.29, and B.30.

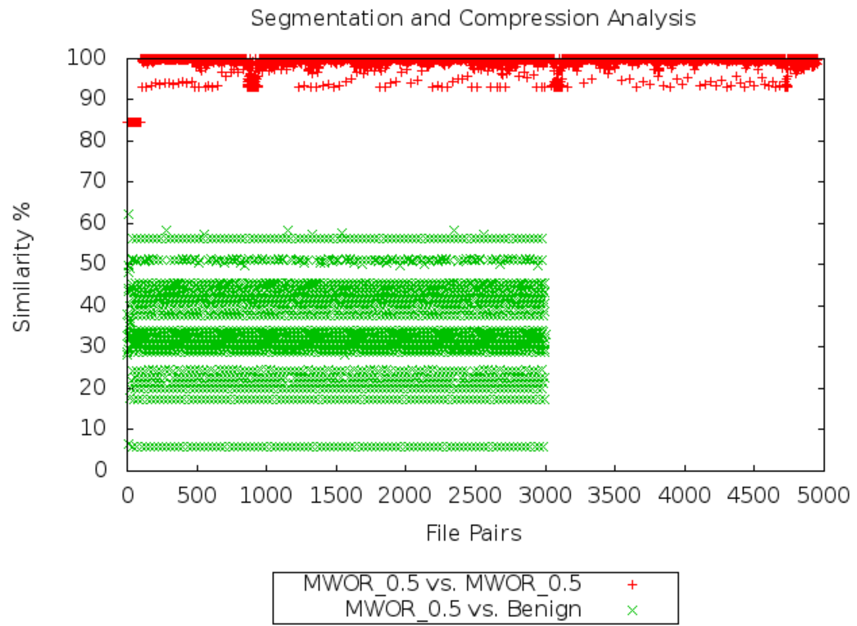


Figure B.25: Wavelet Transform → 0 iterations (MWOR 0.5)

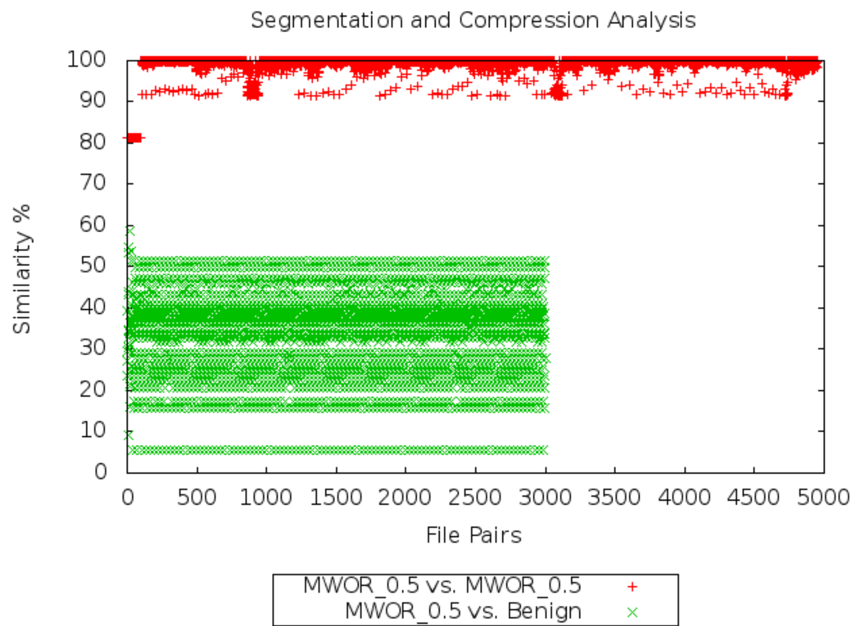


Figure B.26: Wavelet Transform → 1 iteration (MWOR 0.5)

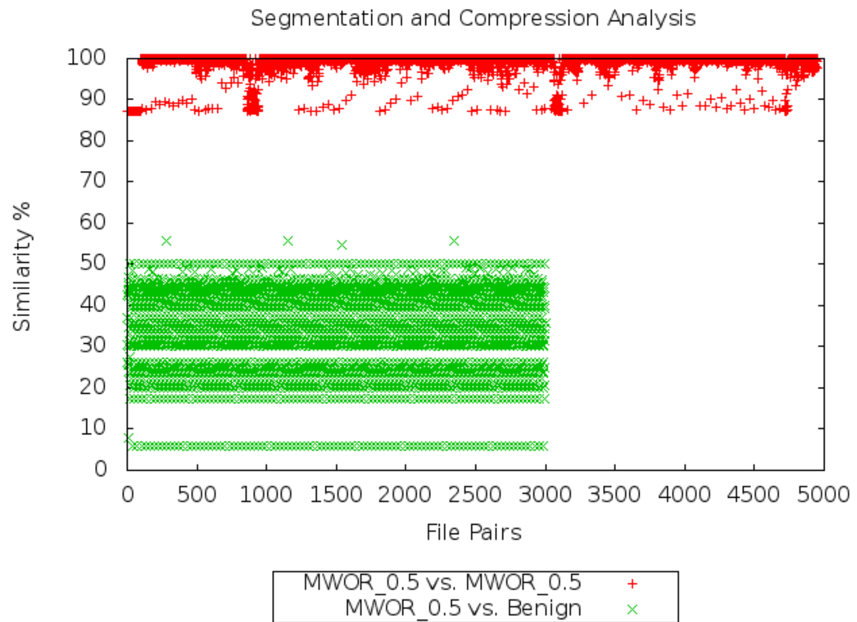


Figure B.27: Wavelet Transform → 2 iterations (MWOR 0.5)

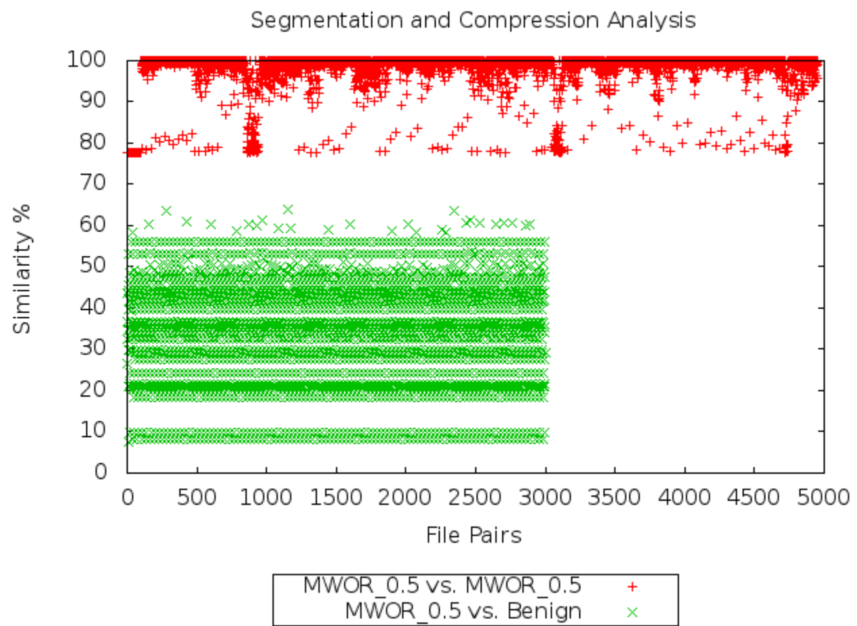


Figure B.28: Wavelet Transform → 3 iterations (MWOR 0.5)

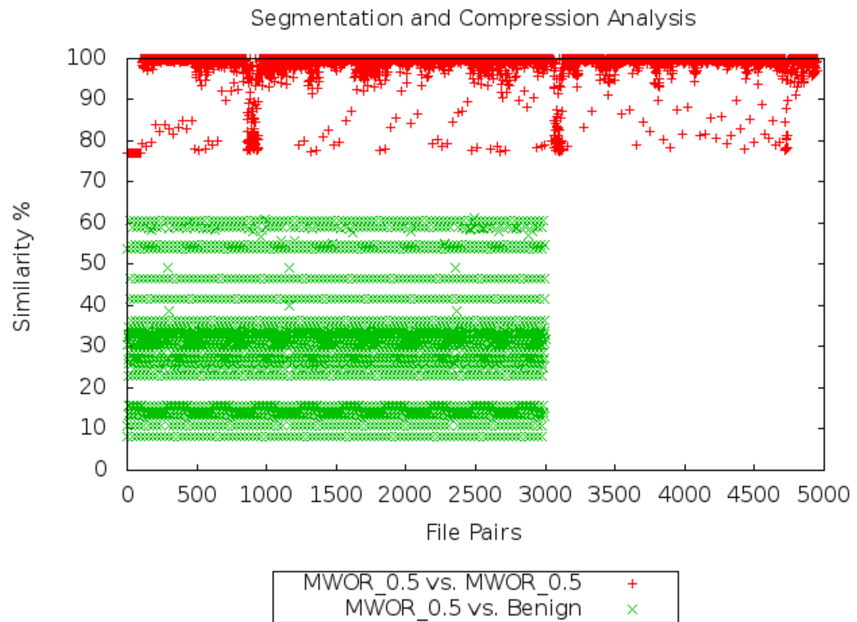


Figure B.29: Wavelet Transform → 4 iterations (MWOR 0.5)

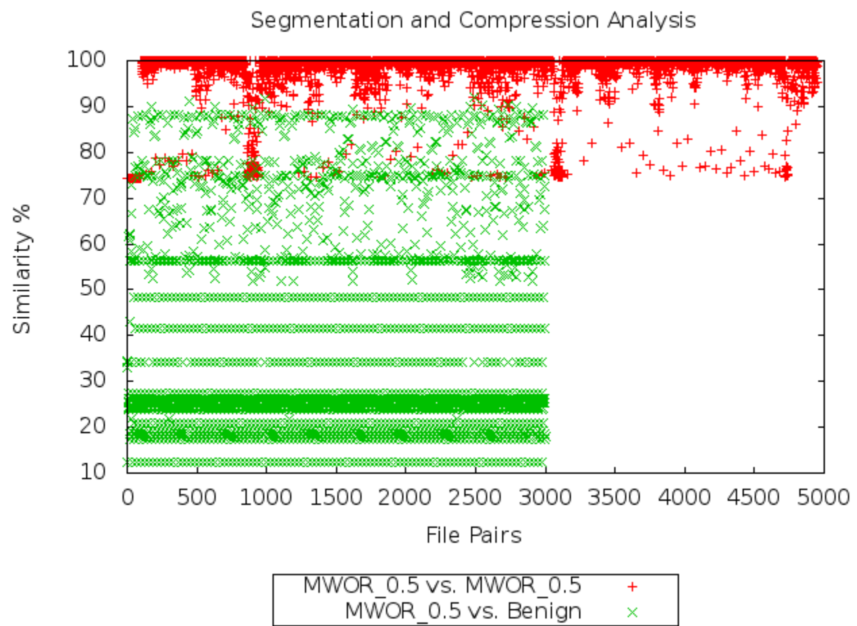


Figure B.30: Wavelet Transform → 5 iterations (MWOR 0.5)

APPENDIX C

Wavelet transforms on NGVCK

The effect of varying wavelet transform iterations for NGVCK is plotted in Figures C.31, C.32, C.33, C.34, C.35, and C.36.

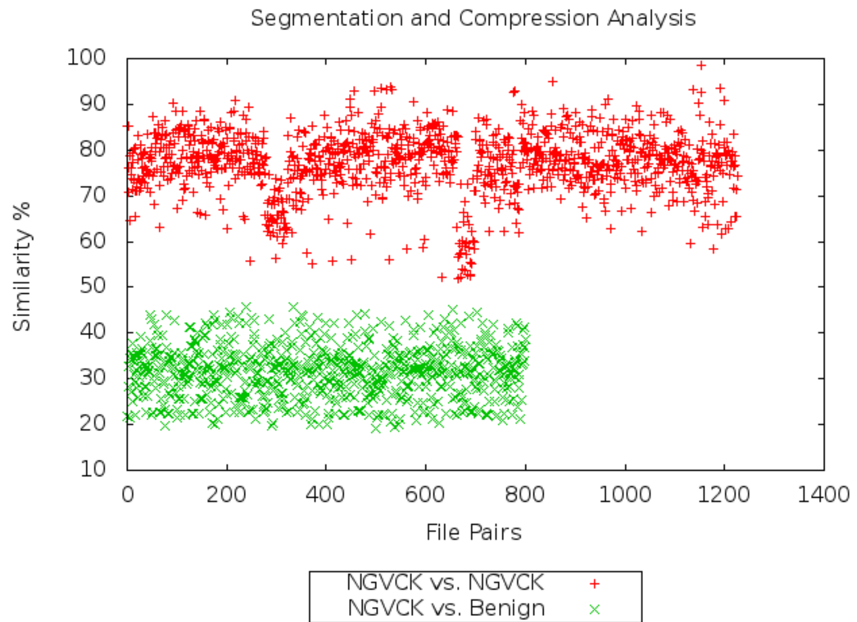


Figure C.31: Wavelet Transform → 0 iterations (NGVCK)

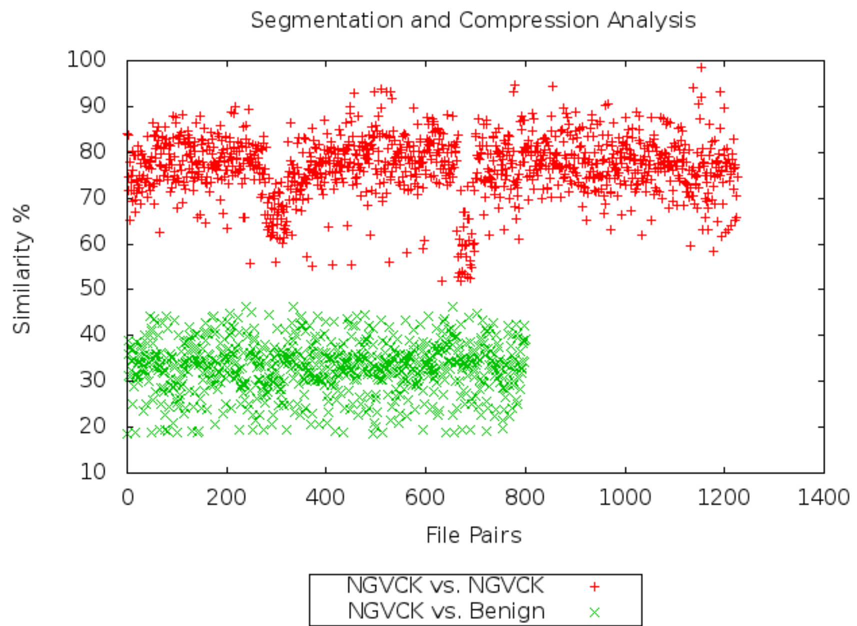


Figure C.32: Wavelet Transform → 1 iteration (NGVCK)

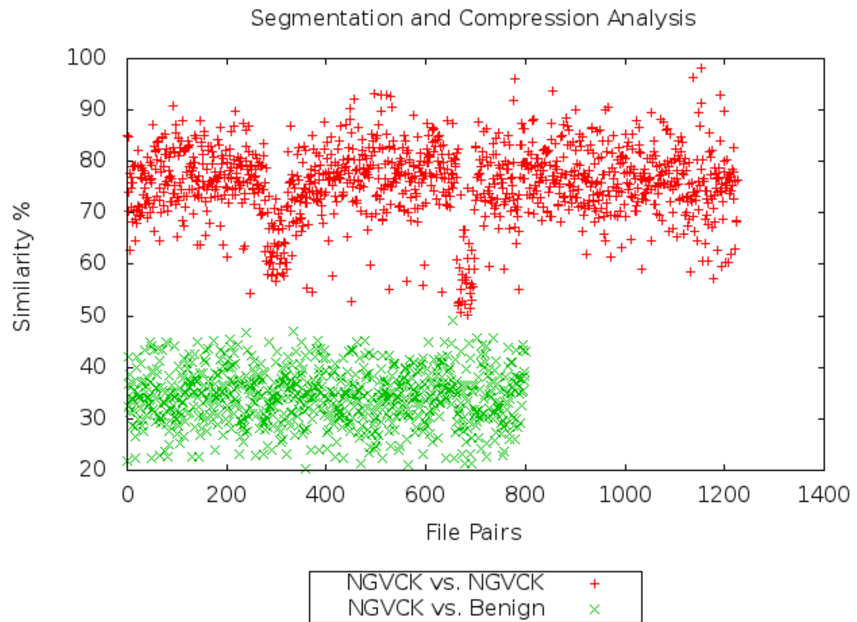


Figure C.33: Wavelet Transform → 2 iterations (NGVCK)

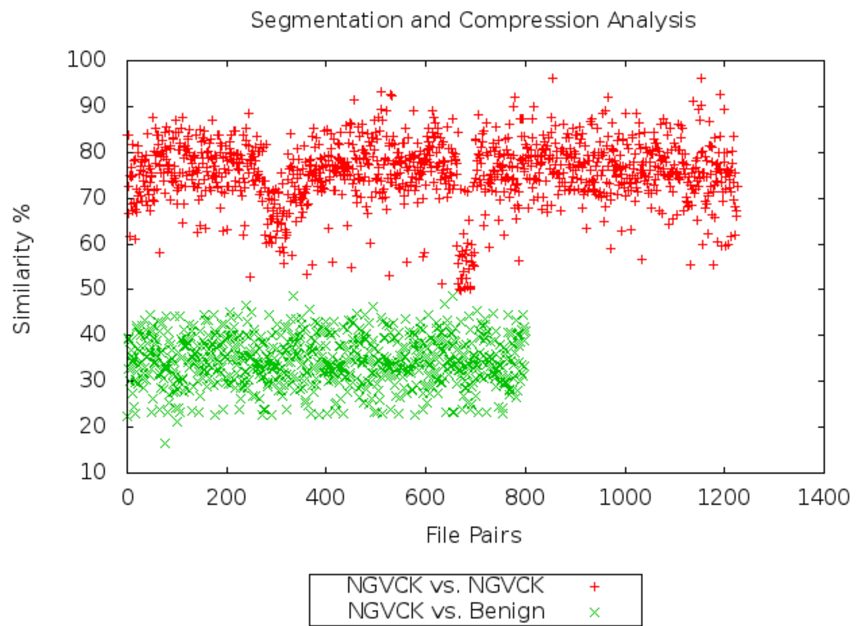


Figure C.34: Wavelet Transform → 3 iterations (NGVCK)

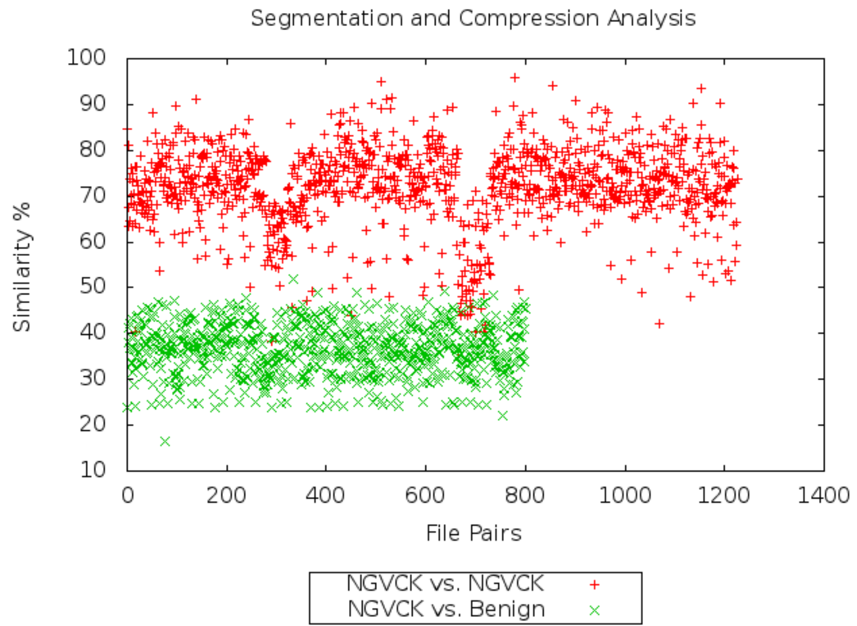


Figure C.35: Wavelet Transform → 4 iterations (NGVCK)

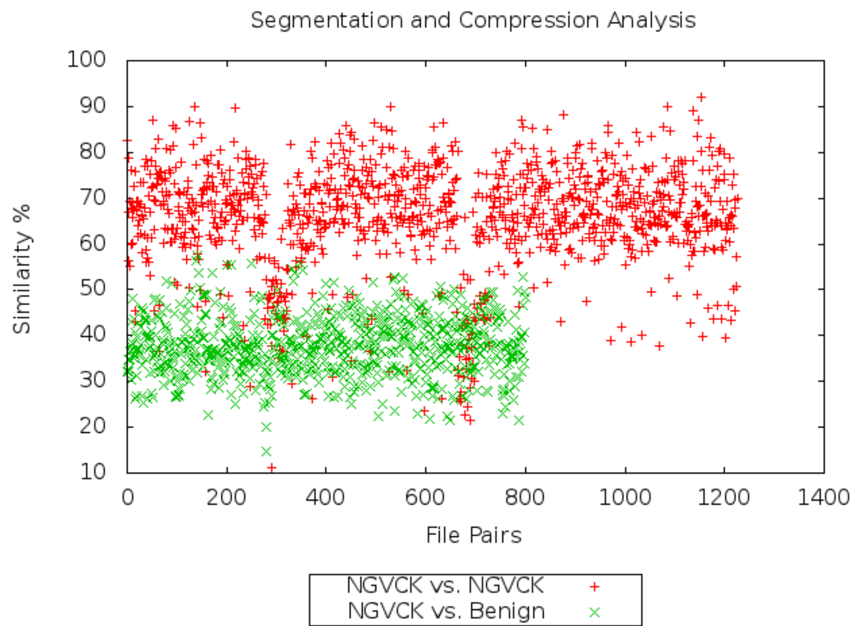


Figure C.36: Wavelet Transform → 5 iterations (NGVCK)

APPENDIX D

Wavelet transforms on G2

The effect of varying wavelet transform iterations for G2 is plotted in Figures D.37, D.38, D.39, and D.40.

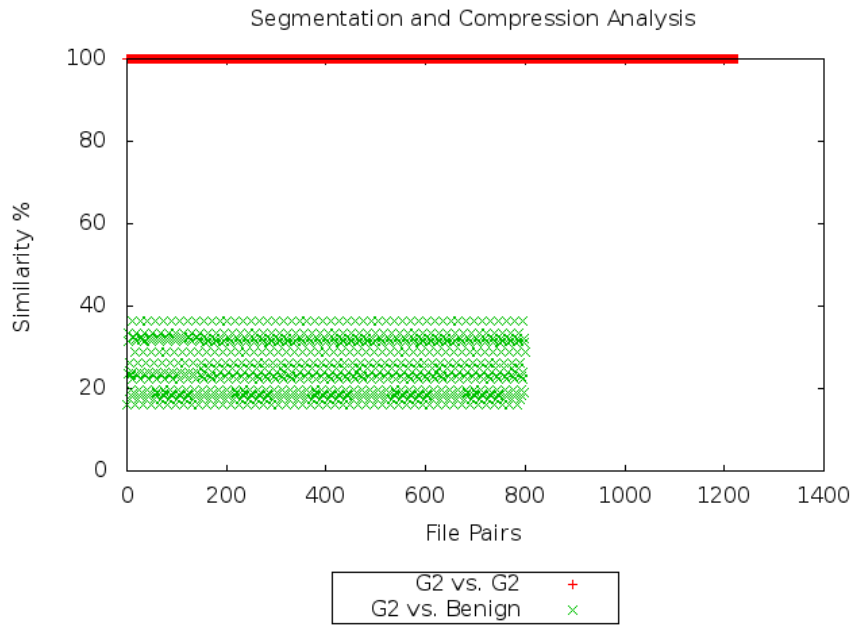


Figure D.37: Wavelet Transform → 0 iterations (G2)

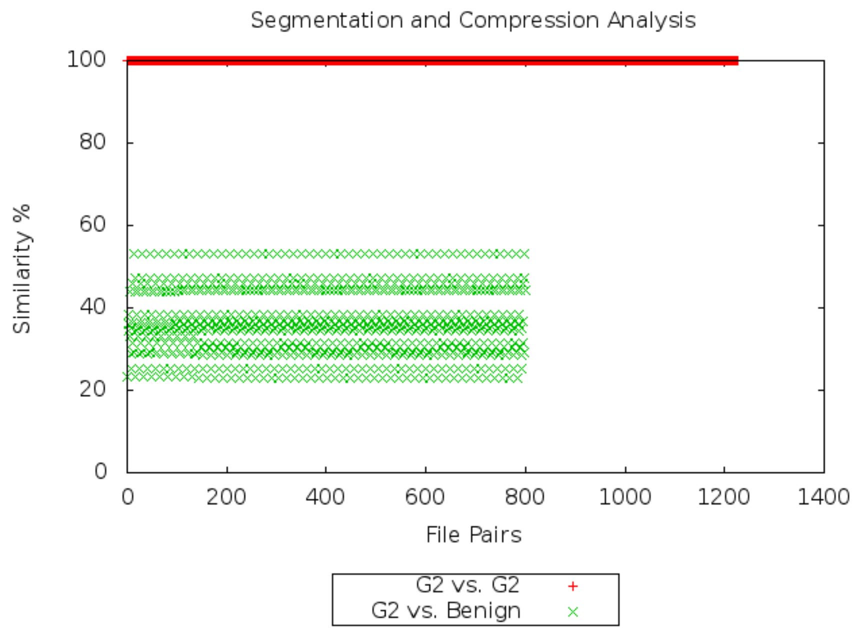


Figure D.38: Wavelet Transform → 1 iteration (G2)

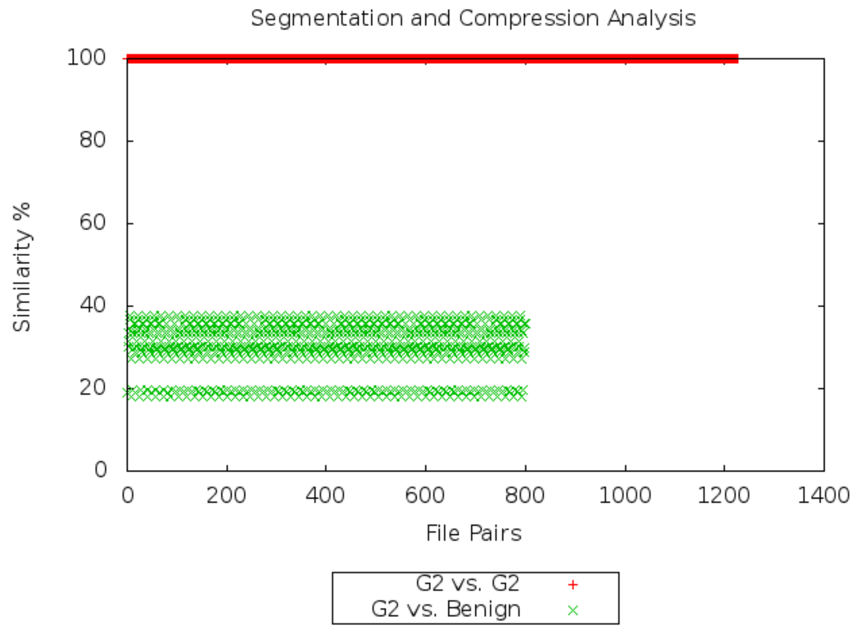


Figure D.39: Wavelet Transform → 2 iterations (G2)

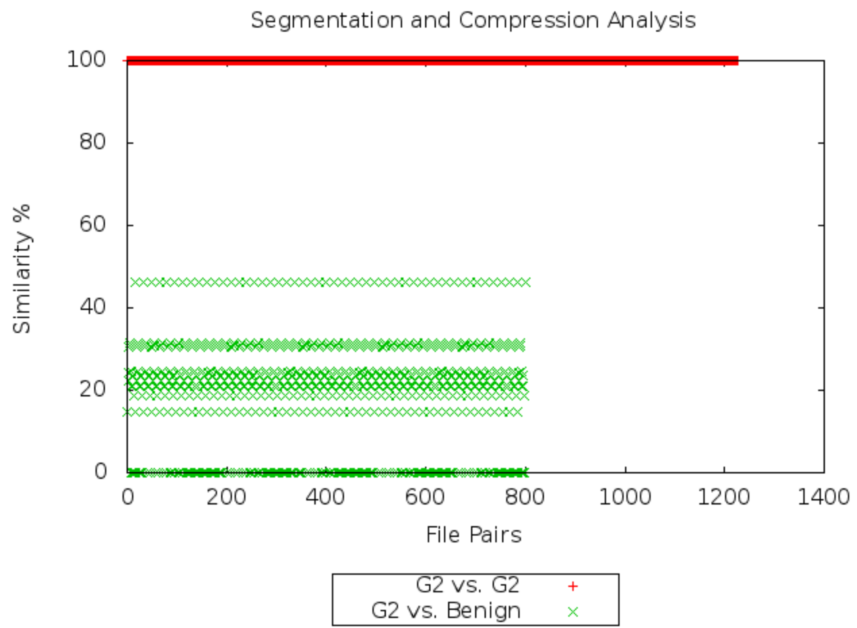


Figure D.40: Wavelet Transform → 3 iterations (G2)