MALWARE STATIC ANALYSIS TECHNIQUES USING A MULTIDISCIPLINARY APPROACH

by

Muhammad Aljammaz

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	Dr. Edward J. Wegman, Dissertation Director
_____	Dr. Duminda Wijesekera, Committee Member
_____	Dr. Jeremy E. Allnutt, Committee Member
_____	Dr. Donald Gantz, Committee Member
_____	Dr. Stephen Nash, Senior Associate Dean
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: _____	Summer Semester 2013 George Mason University Fairfax, VA

Malware Static Analysis Techniques Using a Multidisciplinary Approach

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Muhammad Aljammaz
Master of Science
George Mason University, 2004

Director: Edward J. Wegman, Professor
Department of Statistics

Summer Semester 2013
George Mason University
Fairfax, VA

DEDICATION

I dedicate this dissertation to my father and role model Abdullah Aljammaz, my lovely mother Salwa Al-Suwailem, to my advisor and mentor Prof. Edward Wegman, HRH Prince Bandar bin Sultan bin Abdulaziz, and to the respected Colonel Joe Ramsey.

ACKNOWLEDGEMENTS

It has been a long journey and completing my work could not have been possible without the support, patience, and guidance of my advisor and mentor Prof. Edward Wegman. Having a mentor is something quite special that most students don't experience, but I have been lucky enough to be mentored by my advisor who was so very kind to me and always took extra time to make sure that I was on the right path.

I would also like to thank my committee members Prof. Donald Gantz, Prof. Duminda Wijesekera, and Prof. Jeremy Allnutt for serving on my committee and sharing their time to advise me. I am especially thankful for Prof. Donald Gantz for his very useful comments on my dissertation and presentation.

I would like to thank my wonderful father Abdullah Aljammaz who has not only been my role model but also a source of inspiration for me throughout my life and especially in my academic endeavor. He helped shape the person who I am today, and for that I am very grateful. I would also like to thank my wonderful mother Salwa Al-Suwailem for all her support and comforting words of wisdom that kept me optimistic throughout the years. My parent's support was essential to me throughout my academic journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

MALWARE STATIC ANALYSIS TECHNIQUES USING A MULTIDISCIPLINARY APPROACH

Muhammad Aljammaz, Ph.D.

George Mason University, 2013

Dissertation Director: Dr. Edward J. Wegman

Most research discussing malware detection completely dismisses signatures as being a thing of the past, accusing signatures of suffering from a weak ability to detect zero-day malware. This indeed could be the case if we are still referring to the classic definition of signatures, which renders them specific to only a single malicious executable binary. But what if these signatures grouped more malicious executables under a single signature? They would then make a valuable defense towards the fight against malware. To create such signatures, we need to develop new methods and techniques to constantly advance the state of the art as malware gets more and more elusive under old methods and approaches. The methods I will discuss not only give a good chance of creating effective signatures for malware, but also provide something just as important giving the malware analyst an automated approach to understanding key characteristics of the analyzed malware.

This dissertation has many contributions. The main contribution is a fully automated malware analysis system that can create families of malware, each able to be classified into its appropriate family, including zero-day malware. Another contribution is a new pruning algorithm that tests cluster strength and ensures the tightness of a malware family. The dissertation also incorporates a novel application of blockmodeling to the problem of malware analysis, which takes the form of a visual component in the system. It also creates a novel malware family signature based on n-gram frequencies composed of instructions and API function calls. Two experiments were carried out testing the accuracy and scalability of the system. The experimental results show that this system is highly accurate and scalable.

1 INTRODUCTION

Malware is any type of software that has potentially malicious intent programmed by its creator. The name malware itself comes from the combination of the two words "malicious" and "software". There are many types of malware lurking around in the computing world today, like viruses, Trojan horses, worms, bots, spyware, rootkits, and various others that have been grouped under other names based on their behavior. This sudden explosion in malware creation in recent years is due to the fact that it has become a profitable business through stealing credit card and personal information, corporate spying, spam distribution, and almost anything else that can be imagined that involves gaining unauthorized access to computer systems. The year 2008 was especially a hectic year for the malware epidemic, as confirmed by an antivirus company called F-Secure. F-Secure had said it has seen somewhere around two times as much malware accumulated in 2008 compared to the sum of its 21 year history [1]. To show the extent of just how far malware has spread, a recent news report [2] indicates that malware has made its way into the International Space Station through one of the laptops used by the astronauts on board and has been officially confirmed by NASA. The financial impact of malware has been estimated to be in excess of $13 billion as of 2006 [3], and there is no doubt that malware will continue to be a growing problem in the future if the past is any indication

of things to come. Therefore, it is essential to continue researching anti-malware techniques to help reduce this growing threat.

The current and most widely deployed defense mechanisms used against malware are virus scanners. Current virus scanners rely on two methods, signature matching and heuristic analysis. Signature matching involves storing signatures of various individual malware locally in the machine and comparing each file against each signature. Heuristic analysis tries to identify suspicious behavior as it is happening on the system. The two basic differences between the two methods is that the signature based method relies on the creation of signatures of known malware, and has no false positives, while the heuristic analysis method does not rely on known malware and can potentially detect unknown malware but has a high false positive. Of course, in the real world good antivirus software is usually a hybrid of the two analysis methods, with each one complementing the other.

The traditional based signature matching systems are good at finding known malware, but not unknown or modified versions of known malware for which they have not generated a signature yet. Thus the need for a more flexible signature based system, which can have a better detection rate when it comes to polymorphic malware (different versions of the same malware), has risen. This research will be focused on the static analysis of malware and the creation of more robust signatures.

1.1 Background

In this section some I briefly discuss the various disciplines that have been used within this dissertation. Most of the topics that will be discussed have been applied to the goal of malware analysis in this dissertation.

1.1.1 Information Retrieval

Information retrieval is an interdisciplinary science primarily focusing on searching for documents. It is based on concepts taken from mathematics, computer science, information science, library science, and statistics [4]. Classical uses of information retrieval dealt with creating systems for libraries in order to give automated access to the content of stored documents ranging from books, journals, magazines, and articles. In recent years, information retrieval has gained more spotlight with the advent of online web searches through popular search engines. Information retrieval contains a wide variety of branches including text retrieval, which is commonly referred to as document retrieval. Document retrieval is what web search engines are focused on today. Typical document retrieval systems try to find similar documents to the user-inputted queries in accordance to their relevancy. There exist two classes of indexing, form based indexing and content based indexing. Form based indexing has to do with finding syntactic similarities of text, while content based indexing finds semantic associations between documents. An example of a semantic indexing algorithm is latent semantic indexing (LSI).

1.1.1.2 Latent Semantic Indexing

LSI is a way of indexing a collection of documents based upon the terms contained inside those documents. What makes LSI different from other algorithms is that it not only indexes terms to the documents they reside in but also looks at the document collection as a whole. LSI is motivated by an understanding that words used in the same contexts have similar meanings. For example, in analyzing a collection of documents, LSI will understand that "virus" and "malware" can be found in many documents having the same contexts, and that future queries having one of those terms can retrieve the documents attributed to the other. All of this happens automatically through the creation of a semantic space, which gives LSI the ability to retrieve relevant information even when the query shares no terms with the information itself. An important aspect to mention is that LSI creates the semantic spaces without manual input by the user or a predefined knowledge collection. This benefit allows LSI to be language independent. LSI can work with any type of data that can be written as text; this includes strings, words, and numbers. Benefits from LSI include its ability to be effective when dealing with noise in the text such as spelling mistakes, contradictory data, and ambiguous data, all of which stem from LSI's independence of the language used.

The engine behind LSI is a mathematical method called singular value decomposition (SVD). SVD is applied to LSI as an automatic and powerful statistical method that finds associations among the terms in a large collection of unstructured documents [5]. The way LSI works is by creating a weighted term-document matrix and then applying a singular value decomposition of it, creating a matrix representing the

final document retrieval grid [6]. This final matrix has had its dimensions significantly reduced, making it more compact, and in that sense LSI functions as a dimensionality reducer.

The way to create a weighted term-document matrix is by finding all the occurrences of each term in each document then applying local and global weighting functions. Then by applying the SVD of the term-frequency matrix represented as A, three different matrices are derived: a term vector matrix T, a diagonal matrix S having diagonal singular values of A, and a document vector matrix D.

The singular value decomposition of the weighted term-document matrix, A. Where:

$$A = TSD^T$$
$$A_{t\ x\ d} = T_{t\ x\ m}\, S_{m\ x\ m}\, D^T_{d\ x\ m}$$
$$t = terms$$
$$d = documents$$
$$m = is\ the\ rank(A), \leq min(t, d)$$
$$T^T T = D^T D = I_m$$
$$TT^T = I_t$$
$$DD^T = I_d$$
$$S = diagonal\ matrix\ with\ decreasing\ singular\ values\ of\ A$$

LSI then further reduces the rank m to $k < m$, where the value of k is based on its strength of performance in retrieval, which was estimated at around 200 - 300 in one study [7]. Thus the equation becomes $A \approx T_{t\ x\ k}\, S_{k\ x\ k}\, D^T_{d\ x\ k}$ with matrix A turning into a dimensionally reduced matrix, having filtered out noise in the form of un-needed

semantic information from the original text. Now when a user submits a query in the form of a vector of terms, the newly reduced matrix will act as a lookup matrix. The terms that were a part of the query are looked up for each document and a score is given to the documents with a cutoff point, so that only relevant documents are returned.

1.1.2 Social Network Analysis and Blockmodeling

Social network analysis is the study of a network of social ties between entities. It provides a collection of methods and techniques to analyze various social networks. Social network analysis was originally created as an analytical tool for researchers in the social sciences [8]. It has been used in economics, sociology, anthropology, biology, and various other disciplines where networks can be created from collected data. The science of social network analysis is based upon graph theory, which is the study of graphs composed of vertices and the connections between them, called edges. The main objective of social network analysis is to reveal relationships between entities in a social network through the analysis of structural patterns and formations of their networks [9]. The entities in the social network are referred to as actors, and the social ties are the social relationships between the actors.

One of the more powerful techniques in social network analysis is blockmodeling. Blockmodeling can generally be described as "a coherent approach to identifying fundamental structures of social networks" [10]. Blockmodeling's usefulness also lies in its ability to describe those structures found within a social network. Many structures are often non-apparent at first inspection and, hence, blockmodeling has the ability to offer

more than one structure to describe the same social network, giving analysts different perspectives into the social network.

In [11] they define a blockmodel as consisting of two parts:

i. A partition of actors in the network into discrete subsets called positions.
ii. For each pair of positions a statement of the presence or absence of a tie within or between the positions on each of the relations.

In blockmodeling it is assumed that a matrix is used to represent the social ties within the network, and that a one-mode network is being used. The first step of blockmodeling requires the clustering of the actors into classes or clusters where the members of a cluster have similar structural features. This requires having a clear definition of equivalence and a measurement based on that definition to perform the clustering on. The two most widely used equivalences are structural equivalence and regular equivalence. Structural equivalence means that two actors are structurally equivalent if they have exact ties to and from the same other actors, which means that the labels of those other actors must be the same. Two measurements used in structural equivalence are the Euclidean distance:

$$d_{ij} = \sqrt{\sum_{k=1}^{a} [(x_{ik} - x_{jk})^2 + (x_{ki} - x_{kj})^2]}$$

Where,

d_{ij} = The Euclidean distance between actors *i* and *j*.

a = The number of actors in the relational matrix.

x_{ij} = The element in the relational matrix.

And the correlation coefficient:

$$c_{ij} = \frac{\sum(x_{ki} - M_{*i})(x_{kj} - M_{*j}) + \sum(x_{ik} - M_{i*})(x_{jk} - M_{j*})}{\sqrt{\sum(x_{ki} - M_{*i})^2 + \sum(x_{ik} - M_{i*})^2}\sqrt{\sum\left(x_{kj} - M_{*j}\right)^2 + \sum\left(x_{jk} - M_{j*}\right)^2}}$$

Where,

c_{ij} = The correlation coefficient of actors i and j.

M_{i*} = The mean of the values in row i.

M_{*i} = The mean of the values in column i.

x_{ij} = The element in the relational matrix.

Unlike structural equivalence, regular equivalence does not require exact ties to the same other actors (they can have different labels). It does not require having the same number of incoming and outgoing ties. Thus, two actors are regularly equivalent if they have incoming and outgoing ties to and from equivalent actors. The measurement used for regular equivalence is called REGE. REGE is an algorithm that uses an iterative

procedure that keeps estimating the regular equivalence between two actors using the changes of equivalence in the actors adjacent to them [11].

Now clustering can be performed on the computed equivalences. There are many types of clustering algorithms but the most popular are Concor, which stands for CONvergence of iterated CORelations, and hierarchical clustering. Once the clusters are created, the actors are aligned to their appropriate rows and columns so that they are grouped by cluster affiliation, and then the matrix takes on a different arrangement than it had originally. As defined earlier the resulting clusters in social network analysis are called positions. After the positions have had their actors aligned, the matrix is segmented into many different blocks. Blocks are different regions of the matrix that have been created based upon the relation of two positions. Therefore, a blockmodel that has k positions will have k^2 blocks, and from these blocks there will be k diagonal blocks and $k\,(k-1)$ non-diagonal blocks [10]. After the blocks have been created the blockmodel of the initial matrix relation is represented as an image matrix, which has as vertices the positions that were created and as edges the relational ties between the positions.

To determine the relational ties in the image matrix between the positions, one must use a specific criteria for deciding if there is a tie represented as '1' or a '0' if there is not. The following specified criteria are listed in [11]:

1. Perfect fit: This criterion is used for dichotomous relations. For a perfect fit, a block must either be all 0's or all 1's. Hence the relational tie in the image matrix is a 1 only if the block is all 1's and 0 if it is all 0's.

2. Zeroblock: This criterion is used for dichotomous relations. The relational tie in the image matrix is a 0 if the entire block is filled with 0's; otherwise, it is a 1.
3. Oneblock: This is a criterion used for dichotomous relations. It is the opposite of a zeroblock in that the relational tie in the image matrix is a 1 if the entire block is filled with 1's; otherwise, it is 0.
4. α Density criterion: This is a criterion used for dichotomous relations. The previous criteria rely on the blocks being either fully 1's or 0's and real world data is rarely in that form. Therefore, a density threshold is defined so that if the density of the block is smaller than the threshold, its corresponding relational tie in the image matrix is a 0 or else it is a 1. This threshold is usually the calculated density of the whole relation.
5. Maximum value: This is a criterion used for multi-valued relations, where blocks that contain only small values in them are considered zeroblocks, and blocks that contain any large values are considered oneblocks. This is similar to the zeroblock criterion, but for multi-valued relations.
6. Mean value: This also is a criterion used for multi-valued relations, where the mean value of the entire relation is calculated. Then the mean of each block is calculated, and if it is greater than or equal to the relation’s mean then it is a oneblock or else it is a zeroblock. It is similar to the density criterion, except that instead of using a density value it uses the mean value.

As can be seen blockmodeling takes in an arguably large amount of information and compresses it into a much more simple and general form. Therefore, blockmodeling can be thought of as a dimensionality reducer because the image matrix is a reduced form of the original relational matrix. In blockmodeling, clustering is a central component because blockmodeling's whole goal is to analyze blocks of a matrix, blocks describe the relations between positions (clusters of actors), and so it can be thought of as analyzing clusters of actors.

1.1.3 Odds Ratio

Before giving a definition of what odds ratio is, it is important to define 'odds' in statistical terms. Odds are used in relation to some event occurring, which is the probability or likelihood of the event happening divided by the probability or likelihood of the event not happening. The formula is [12]:

$$\frac{P}{1-P}$$

Where variable 'P' is the probability of the event happening.

The odds ratio, as its name implies, is the ratio of odds of an event occurring in two different groups or sets. It can be thought of as a comparison of odds of an event happening in two different groups. The formula is given below [13]:

$$\frac{P1/(1-P1)}{P2/(1-P2)}$$

P1 is the probability of the event occurring in group1 and P2 is the probability of the event occurring in group2. Having a result of 1 from the odds ratio shows that the event is just as equally likely to happen in both of the two groups. A result that is larger than 1 shows that the event is more likely to happen to the first group. A result that is smaller than 1 shows that the event is less likely to happen to the first group.

1.1.4 Malware Types

Before the word malware was ever coined, malicious software had always been referred to as a virus. As time progressed, different names were created for malicious software based on their specific behavior. Therefore, the word malware represents the general name given to all malicious types of software including viruses, Trojan horses, worms, bots, rootkits, spyware, and various others. In this section, explanations of each type of malware will be given.

1. Viruses: A computer virus is a type of malware that replicates itself onto other files and infects the host computer. Viruses are usually attached to executable files and are activated as soon as the executable is run on the host pc; if the executable is not run, then the virus stays dormant and cannot cause any harm on the system. Once the file is run the virus will usually put a copy of itself into other executables on the system and

depending on what the author of the virus has programmed into it, it may do anything from outputting annoying messages to destroying data on the computer system. Viruses can spread by downloading the executable that contains the virus either by email attachments, website downloads, or USB drive file transfers. Viruses cannot propagate from network to network. They rely on users (unknowingly) to do the propagation for them [14]. The term virus was first used by Fred Cohen in an academic paper written in 1984, but the first virus to spread in the wild was the Elk Cloner Virus [15].

2. Worms: Computer worms are a type of malware that do not need a host executable to infect other computer systems. A computer worm is a completely independent executable program that can spread from computer to computer through the use of vulnerabilities and exploits in the target systems. Once the worm has gained access to the target system, it creates copies of itself and propagates to other systems. This continues until a vast number of computer systems are infected. Worms use the network as a medium, which makes the propagation possible without the use of any human manual intervention. The first worm to ever be released into the wild or internet was the Morris worm [16]. The worm quickly spread to academic, military, and commercial systems in a matter of hours.
3. Bots: The name bot was taken from the name robot, which emphasizes the automation aspect of it. Traditionally bots were and still are used as web

crawlers that gather information as well as being used in IRC (internet relay chat) chat rooms where the bots accept commands and output relevant information. The bots would do automated tasks like run trivia games and give operator privileges. In 1993, an IRC bot named Eggdrop had a special feature called a botnet that linked many bots together and used their strength in numbers against chat room attackers [17]. Malware authors utilized the botnet concept and incorporated it into their malware. The malicious use of bots have properties of worms in that they can independently propagate without the use of a host program, but they are much more intelligent and organized than a worm. Once a bot infects a computer system host, it reports back to a central server that is the command and control center for all of the bots in its network. This network is called a botnet. Each bot represents an infected and compromised computer system that is under control of a single entity.

4. Trojan Horses: Trojan horses are a type of malware that have been named after the mythical wooden horse that the Greeks used as a ploy to gain access into the walls of Troy [18]. What distinguishes the Trojan horse from the other malware is that it is designed to look like a legitimate program when in fact it hides malicious software within it. The malicious code can even harbor other different types of malware like worms, viruses, and bots. It can also just create damage on its own like creating backdoors, erasing information, and much more.

5. Rootkits: The name rootkit is derived from the word 'root', which is the highest privileged account on a UNIX operating system, and 'kit' referring to the set of tools. Rootkits are often installed on the target system once it has been compromised, and the attacker (in this case a malware) has achieved privileged access like root. The rootkit hides any known evidence of the compromise. Because it usually resides at the kernel it can thwart anti-virus software that tries to detect it. Removing rootkits is an almost impossible task because it could have modified any type of kernel-level information, which might cause irreplaceable damage [19].
6. Spyware: Spyware is a type of malware that is primarily based on tracking the Internet activity of the compromised system it is on. It may have a keylogger that logs the information typed. This can be particularly dangerous if the compromised users access bank accounts and other personal finance sites. Usually spyware will use deceiving tactics to install themselves through vulnerabilities of the web browser or by using a Trojan horse [20].

These naming mechanisms provide a way to categorize the vast amount of malware by behavior, but today there is malware that encompasses many behaviors of each of the previous malware types, creating a very sophisticated type of malware. If the past is any indication of the future then malware will only continue to get more and more sophisticated as time goes on.

1.2 Review of the Literature

This section discusses previous work in static analysis. Within the static analysis field there is binary based detection and instruction based detection. Most static analysis research started in binary based detection and then started to shift to instruction based detection.

1.2.1 Static Based Detection

The whole idea behind malware static analysis is to try and detect if a particular binary is a malware without the need of actually executing it. This aspect of static analysis reduces the risk of exposure to the malicious behavior of the malware because it cannot bring harm to the system on which it is currently analyzed. Through static analysis, one can obtain many general characteristics of the malware with little effort as compared to dynamic analysis, which requires much more effort and time in observing the malware's behavior as it is running in some type of environment. Thus, the advantages of static analysis are speed and safety.

1.2.1.1 Binary Based Detection

The vast majority of malware that are run on computer systems are in binary form, and that includes executables used in the Windows operating system families. In binary detection, they use features collected in binary form without actually knowing what the contents of the file are. One of the first endeavors in creating a method that automatically creates signatures for malicious executables was the work done at IBM by

[21]. They created a statistical method for extracting signatures automatically from the binary of a malicious executable. They developed a technique that is akin to the speech recognition technique. The result was an algorithm based on speech recognition that extracted the signatures from a relatively large number of files with a comparable result to that of a manual extraction by a human expert. In [22] they create an automated technique that generates multiple neural network classifiers along with a voting procedure to reduce the amount of false positives that are created when classifying a file as a virus or not. Their work is similar to the work of [23], using neural networks for detecting boot sector viruses, except that they applied it on all win32 binaries instead of just boot sector viruses. Although they did try to reduce the false positives through a voting procedure, they concluded that their results show that they cannot reduce the number of false positives enough to make it a reliable virus detector when used on newly unseen viruses.

In [24] they created a framework based on data mining algorithms. Their dataset consists of both malicious and benign executables for the windows operating system. They split their datasets into two subsets composed of a training set and a testing set. The data mining algorithms were used on the training set to create classifiers, which had their accuracy tested by using them on the testing set. For features they extracted three different types of features from the collected binaries. The first feature is actually a collection of information extracted from the executable's PE header. They include a list of DLLs used by the binary, a list of function calls made by the binary, and the number of function calls in each DLL. The second feature is a collection of strings taken from the

binary. The third feature is byte sequences taken from the binary in hexadecimal format using a utility called hexdump [25]. The third feature is considered to be the most robust feature compared to the other features because it gives more information about the binary itself. The data mining algorithms that were used on each are RIPPER, Naive Bayes, and a Multi-Classifier system. The results were that the Naive Bayes applied to the strings feature generated the highest accuracy of 97.11%. The second highest was Multi-Naive Bayes applied to the byte sequences generating a classification accuracy of 96.88%. The third feature was a collection of three types of data taken from the PE header and had the least accuracy at 83.62%, 89.36%, and 89.07% respectively. Even though the strings features had the highest accuracy, it is probably the easiest to defeat. The second highest was the byte sequences feature, which is quite close to the strings accuracy but is much harder to defeat.

1.2.1.1.1 Byte Sequences and N-grams

In [26] they expanded on the work of [24] by combining techniques from machine learning and data mining. They call their system the Malicious Executable Classification System (MECS). The feature they extracted from the binary is a collection of byte hexadecimal sequences that have been converted into n-grams. The n-grams are created by combining each four byte sequence into one n-gram. Through this process they created around 255,904,403 distinct n-grams from 476 malicious executables and 561 benign executables. Each n-gram is viewed as a Boolean attribute (it is either present or not). Since the amount of n-grams generated was so large they decided to use the most

relevant n-grams by computing the information gain (IG) for each. The top 500 n-grams were then selected as the final number of attributes. The reduction of n-grams heavily reduces the processing overhead on the classification methods. The classifiers that were used on the n-grams in the experiments were ibk, tfidf, Naive Bayes, support vector machines (SVMs), decision trees, boosted Naive Bayes, boosted svms, and boosted decision trees. Boosting is a method that combines multiple classifiers. The three classifiers that were boosted were svms, J48 decision tree, and Naive Bayes because they had an acceptable addition in computational expense. After creating the classifiers they used receiver operating characteristics (ROC) analysis to evaluate them. The best performing classifier turned out to be the boosted J48 decision tree. In comparison to [24], they credit their better results in how they processed byte sequences. By using overlapping sequences of four bytes they kept useful features, which did not get split across boundaries. They reduced the 255 million n-grams into a set of 500 easy to manage n-grams.

Similar to [26] the work of [27] also uses n-grams, but within the Common N-Gram analysis (CNG) method to detect malicious code. The results obtained are 98% accuracy using a 3-fold cross validation.

In recent work [28] created a system and compared their results to [26] and [24]. The architecture of their proposed technique is divided into four different modules—a block-generator module, a feature extraction module, a classification module, and a correlation module. The block generating module is fed a file that is then split into blocks of size 1k. Then the 1-gram, 2-gram, 3-gram, and 4-gram are generated for each block.

Frequency histograms for the 1-gram, 2-gram, 3-gram, and 4-gram of each block are calculated. The feature extraction module then takes in the previously calculated histograms, and 13 feature sets, which are Simpson's Index, Canberra Distance, Minkowski Distance of Order, Manhattan Distance, Chebyshev Distance, Bray Curtis Distance, Angular Separation, Correlation Coefficient, Entropy, Kullback-Leibler Divergence, Jensen-Shannon Divergence, Itakura-Saito Divergence, and Total Variation, are calculated for each of the 4 frequency histograms that represent each block. This leaves us with a total of 52 features for each individual block. The 52 features are fed into a classification module that is comprised of a boosted J48 decision tree that has been boosted using an AdaBoostM1 algorithm. The classification module classifies each block as either malicious or benign. The final correlation module sees how many blocks for each file were labeled with malicious or benign and decides if the whole file is malicious or not based on some threshold. There were a total of 1,800 benign files and were comprised of 300 of the following file types: DOC, EXE, JPG, MP3, PDF and ZIP. There were 10,311 win32 malware samples and were comprised of the following types—backdoor, Trojan, virus, worm, constructor, and miscellaneous. Their results show that they achieve a higher detection rate than both [26] and [24]. They attribute their success to the way they select features by computing them on a per block n-gram analysis and the correlation between the blocks classified.

In [29] they deviated from the previously discussed methods of classification and focused on benign profiling. The motivating factor behind their method was to try and emulate a biological immune system, in which the immune system has to differentiate

between its own cells and other foreign invaders. The focus was on profiling benign executables by creating frequencies for single byte patterns as potential features in the profile. The malicious executables are recognized as the outliers that have significant deviation from the normal benign profile. Two models were compared to each other in creating the profile. The first was a multivariate Gaussian likelihood model fit with a principal component analysis (PCA), and the second was a support vector machine (SVM) model. In their results they showed that the Gaussian model had a better accuracy rate in its ability to differentiate between malicious and benign executables.

1.2.1.1.2 Embedded Malware

Embedded malware offers a challenge to normal signature based anti-virus detection because it can go undetected when hidden inside normal looking files. In [30] through the use of statistical content analysis techniques, they were able to supplement signature based detection of embedded malware. They obtained 3 different collections of files, including 31 normal application executable files, 45 spyware files, 331 normal executable System32 files, and 571 virus files. Three different modeling methods are applied to the set of normal files, which are one centroid method, multi-centroids method, and exemplar files as centroids method. From the normal file collection they created 1-gram and 2-gram feature vectors as well as distributions of the n-grams. After the set of models were computed from the normal collection, they used a test file to measure how similar its content is to the normal models. This measurement was computed through calculating the Mahalanobis distance between the test file and the normal centroid models

that were computed beforehand. The tests were done on three pairs of groups from the previously listed collection, normal executable vs. spyware, normal application vs. spyware, and normal executable vs. viruses. Their results were relatively good to fair for the case of normal executable vs. spyware and normal executable vs. virus. The main problem was with the high false positive rate, and they concluded that their comparison methods were not sufficient enough for detecting malicious code in a reliable manner. Similar work has also been done by [31].

1.2.1.1.3 Feature Selection

An important aspect of malware detection and analysis is being able to find attributes that can be used as features in the analysis. In [32] they collected various static attributes of a binary executable for the purpose of detecting malicious binaries. The dataset that was used included 1410 viruses, 160 worms, 2520 Trojan horses, and 1,200 benign samples taken from Windows XP and Windows 2000. The attributes were taken from various places in the binary like the DOS header, COFF header, PE optional header, data directories, import table, sections, and resources information. In all there were 42 different attributes, including:

1. DOS checksum. 2. Number of sections. 3. Time stamp. 4. Number of symbols. 5. Size of code. 6. Size of initialized data section. 7. Size of un-initialized data section. 8. Major OS version. 9. Minor OS version. 10. Major image version. 11. Minor image version. 12. Major subsystem version. 13. Minor subsystem version. 14. Entry point. 15. Image file

checksum. 16. Size of exports. 17. Size of imports. 18. Size of resources. 19. Size of exceptions. 20. Size of attribute certificate table. 21. Size of base relocations. 22. Size of copyright. 23. Size of thread local storage. 24. Size of bound imports. 25. Size of import table. 26. Dynamic link libraries used. 27. Registry references made. 28. Cursors. 29. Bitmaps. 30. Icons. 31. Menus. 32. Dialogs. 33. Strings. 34. Font directories. 35. Fonts. 36. Accelerators. 37. RC data. 38. Message tables. 39. Group cursors. 40. Group icons. 41. Version. 42. Entropy of code/text section

Each attribute is modeled as a discrete random variable having more than 1 outcome, and for each random variable a histogram was computed of each outcome. To obtain the probability mass functions (PMFs) of the attributes the histograms must be normalized. For each attributed random variable, the difference needs to be quantified between the PMF derived from the benign executables and the PMFs derived from the malicious samples. The measuring methods of quantification used are based on Kullback-Leibler divergence. The Kullback-Leibler divergence shows the statistical difference between two different PMFs of a discrete random variable. But because Kullback-Leibler imposes rigid conditions to ensure a good difference is calculated, another divergence measure based on Kullback-Leibler is used instead. The resistor average (RA) quantifies the difference between two attributes' PMFs taken from different genres (benign, virus, worm, Trojan). Since RA can only quantify divergence across file genres, another measure called differential RA divergence (DiffRA) is used to quantify how similar the samples are from the same genre. After seeing the results from the RA and DiffRA on the attributes from the different genres, 13 of the 42 attributes were deemed good enough for

usage in a classifier. For their classifier they classify a file as malicious if either cross-correlation or log-likelihood classifiers flag it as malicious. To compare their classifier they use a support vector machine classification. The results show that this statistical model is less computationally expensive, faster, and more accurate than typical machine language classification. The methods created are in themselves not enough to replace signature based detection, but can be used as a supplement to it in the cases of zero-day malware.

In [33] they created a feature selection and evaluation scheme to be used in computer virus detection. For features they created n-grams using the binary's byte sequence. The n-grams are scanned through, and their frequencies within each virus family are recorded. A list of features is constructed from the previously calculated frequencies for each feature that meets a given support threshold within each virus family. The final step is feature elimination, which removes features that occur frequently in one virus family and are exclusive to that family. This step retains features that meet a threshold of inter-family support. They compared their method to a traditional model that examines 16-byte sequences in all viruses and kept those that had a support of at least 1%. In their newly created model, they used an intra-family support threshold of 40% (with a limit of 500 features) and an inter-family support of 3 out of 110 families. They ended up with 12% more features than the traditional model, and when they inputted both feature sets into classifiers, the newly developed method had an overall accuracy of 93.65% compared to around 50% accuracy for the traditional model. In their final experiment, they tried to optimize the feature selection by using ranges of n-grams from

3-8 and 3-6 for the inter-family support. What was found is that the shorter the n-gram was, the more accurate the detection, and the smaller the support for inter-family, the better the detection was as well. But what happens when the n-gram gets smaller is that the number of features is magnified which creates more processing. Other similar work has been done by [34], [35], [36].

1.2.2 Instruction Based Detection

In instruction based detection, the malware itself is no longer treated as a byte sequence of binary code, but instead as a sequence of assembly instructions that make up the binary. There is still no need to execute the binary to obtain the instructions, but the binary must be scanned through to obtain the set of assembly instructions that compose it.

One of the first works dealing with instruction assembly based malware detection was [37]. They created a system that generates phylogeny models for malware based upon the assembly instructions that make up the malware. Phylogeny is a type of network used in the area of biology, and the discipline of bioinformatics tries to automatically create phylogeny models based on biological information such as gene sequences. When applying this method to malware, they had to choose which information to use as features. They did a simple experiment to see which would be more effective to base their phylogeny models on; they used both byte code as their information and assembly code. The results showed that the assembly code was more reliable than byte code in creating an accurate phylogeny model. The system itself that they created is a four part module system setup in the following order tokenizer, feature occurrence matrix extractor,

similarity metric calculator, and clusterer. The tokenizer takes in programs and outputs sequences of assembly code that are converted into n-grams and n-perms. The feature occurrence matrix extractor creates two feature occurrence count matrices for each of the n-grams and n-perms features, such that each matrix holds the number of times a feature occurs in each program. The similarity metric calculator takes in as input the feature occurrence count matrices created in the previous step and creates a symmetric similarity matrix between all the programs. The similarity metric used was the TFxIDF weighting and cosine similarity. The final part is the clusterer, which clusters the programs by their similarity measures calculated previously. The clustering software used is a program called CLUTO [38] that has a UPGMA clustering criterion function, which is used to generate biological phylogeny models. There were two different experiments done. The first was to see the weaknesses of the n-gram and n-perm approaches in finding similarities in programs that are different because of permutation operations. The samples used consisted of permutated and unrelated samples, and their similarity scores were compared. The results showed that n-perms have a better similarity score for permutated programs and create a fair phylogeny model. The second experiment showed how well three major anti-virus software systems fared when they compared their name classifications with their own phylogeny model. It showed some naming inconsistencies between the anti-virus software, which shows that a unified naming scheme through an automated phylogeny model would be beneficial.

A more recent work dealing with instruction based malware detection comes from [39]. They created a data mining framework for the detection of malware. Concepts are

taken from information retrieval and classification techniques and applied to their research. For representing the programs, they used vector space models, where every program is a vector in n-dimensional space with n being the number of distinct instruction sequences. The framework itself consists of two different experiments. The first is a supervised learning experiment that trains, validates, and tests many different classifiers on the collection of thousands of malicious and clean program samples. The second experiment uses sequential association analysis to select features and create signatures. In the first experiment a total of 410 malware and 410 benign files were used. The malware consisted of Trojans, viruses and worms. The features were extracted from the programs in the form of variable length instruction sequences. The unary variable removal method was performed to reduce the sequences by 98.2%. A term document matrix was made from the extracted features, where the features represented terms and the programs represented documents. The matrix contained the term frequencies in each program. From the remaining features, 50% were reduced after applying the Chi-Square test of independence having a final set of 633 features that were reduced from an initial collection of 62,608. Three models logistic regression, neural networks, and decision trees were fitted with the training data (30% of original data). The neural network achieved the best overall accuracy rate with the lowest false positive and second but very close detection rate. The first in detection accuracy was the decision tree, but it had a higher false positive rate, which reduced its overall accuracy to second. In the second part of their work, they conducted sequential association analysis that extracted instruction sequences that were found in malware collections but were not specific to any particular

malware, therefore creating a generalized set of signatures. The algorithm that was designed achieved an 86.5% detection rate with a 1.9% false positive rate.

In 2007 a detection method for malicious code through the use of statistical analysis of assembly instruction distributions was introduced by [40]. In the work they gathered 20 windows XP executables and split them into 4 different size groups. For the malware 67 malware executables and dlls were taken from seven different classes of malware. A list of opcodes was then created from the files and augmented with more opcodes found from Wikipedia totaling 398 IA-32 opcodes. In the benign samples there were around 1.5 million opcodes and 192 unique opcodes. Overall 72 opcodes accounted for 99.8% of all opcodes found, 14 opcodes had 90%, and the top 5 were around 64%. The malware samples produced similar results. The frequencies of each opcode were calculated. The 14 most frequent opcodes and 14 least frequent were put in a contingency table. Then using Pearson's Chi Square procedure followed by a post-hoc standardized residual testing of individual cells it showed the statistical differences in opcode frequency between the benign samples and the malicious seven class samples. Then an association strength between the malware classes and opcodes was calculated using Cramer's V. The results showed that the 14 common opcodes were weak predictors, explaining only 5-15% of the frequency variation, while the 14 rarer opcodes had a 12-63% variation. The malware frequency distribution deviates significantly from the benign samples, and the rarer opcodes explain more of the variation than common ones.

Recently [41] did more of the same malware detection techniques that were discussed previously. The primary focus was on information retrieval by using weighted

term frequency to create a weighted opcode sequence frequency and applying a cosine similarity measure to detect malware variants. There is not anything particularly new in this method, but for their mining of opcode relevance they applied a mutual information measure as a measure of how statistically dependent two variables are. This measure helped in computing the relevance of the opcodes based on their frequencies in both the malware dataset and benign data set, which reduced noise produced from irrelevant opcodes.

An interesting approach was created by [42] where they combined a collection of techniques from bioinformatics, data mining, and information retrieval. Taking much of their inspiration from bioinformatics, they treated a binary program as a genome, and the features that were acquired from the binary were the genes. They gathered around 3548 malicious executables and 200 benign files. The first step was to disassemble the files into low-level assembly and extract the opcodes only, leaving each file represented by a sequence of opcodes. There exists somewhere in the region of two hundred opcodes in the Intel instruction set. These opcodes were then grouped into categories of similar function, which Intel has already grouped into 13 groupings. The groupings were reduced even more and made into 11 groups. Now each of the opcodes in the sequence will be converted into one of these 11 groups. The file was now represented as a sequence of s-opcodes, which is the newly termed word for the grouped opcodes. The next step was to create malicious families based on the s-opcodes that were generated. To achieve this they used clustering. For clustering they took the s-opcode sequence representing the malware and made many subsequence terms out of it using a sliding window method,

creating a vector of terms that represents the malware. The vector is taken and the TF-IDF is calculated (over all subsequence terms) for each term. Then the cosine similarity measure was used to create a score matrix representation of the similarity between the vectors. For clustering they used the DBSCAN algorithm along with the TF-IDF-Cosine similarity matrix. After having created the families they retrieved features from each family as local subsequences from each family class that appear in every member of that family. They implemented the feature extraction through the use of a multiple alignment algorithm. In the detection phase they have a set of features, one per family, that they use inside their 'bioinformatics-based' classifier that was implemented using a pair-wise local alignment algorithm. The pair-wise local alignment algorithm locally aligns the incoming s-opcode sequence with all the features in the database. They applied their method at the network layer in simulated network traffic and achieved an identification rate of 83%.

1.3 Problem Statement

Through the combination of social network analysis, information retrieval, and statistical analysis methods, a static malware analysis system can discover and identify key characteristics of malware, thereby creating well defined families, which assist in malware detection and understanding. Some of the challenges faced include:

1. Development of a fully automatic system capable of clustering the malware into families using various created algorithms.

2. Automatically creating stable and unique signatures for the different families created in the clustering phase.
3. Choosing which type of features to use in the signature phase.
4. Creating visual aids to see the malware families and how they relate to other families.

2 METHODOLOGY

This chapter provides a description of the general framework of my malware analysis system. I introduce the way malware is perceived from the point of view of this research, how the malware is collected, an explanation of various components of the system, and how I evaluated the performance of the system.

2.1 Malware Perception

The way I view each malware is as a large sequence of low-level assembly opcode instructions that are intended to carry out the specific tasks that its author had programmed it to do. In this dissertation I have gathered a sample of malware and obtained their disassembled instructions, which are the building blocks of the malware, and created various types of social networks from them. But instead of using each instruction independently as actors in the social network, I created n-grams from the individual instructions to obtain a set of grouped instructions of various lengths. These n-grams represent the fingerprints of each malware, and can capture many hidden meanings that might be left out if I only looked at the instructions independently. Figure 1 shows an example of the process.

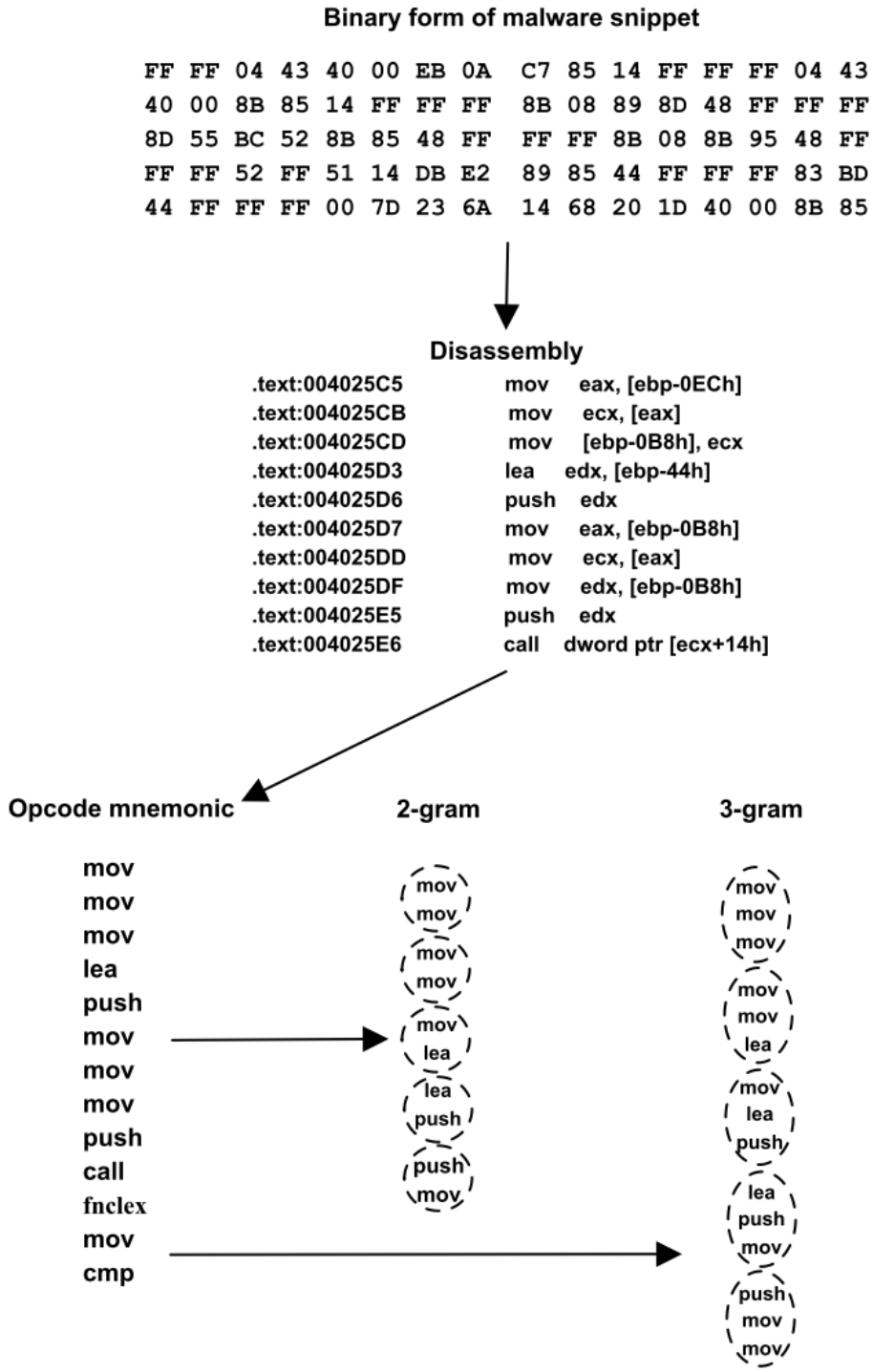

Figure 1: Malware Perception

Figure 1 first starts out with a snippet of a malware still in binary form but displayed in hexadecimal. It is then disassembled into recognizable assembly instructions with their parameters. From the disassembled form only the instructions are taken and the rest thrown out. This leaves the data in a sequence of opcode mnemonics. From this sequence, the opcode mnemonics will be made into various different n-gram lengths, ranging from 2-grams to 10-grams.

2.2 System Architecture

In this section I discuss the architecture of the malware analysis system as shown in Figure 2. A short general description will be given, but each section will be discussed in the following chapters. The right portion of the figure starts with the training set using a malware collection. It passes through the n-gram extractor, clustering into families, visualization module, signature creation, and finally the signature repository where the signatures are stored. The left portion of the figure shows how to classify an unknown executable. It only goes through the n-gram extractor and then to the classifier, where it is compared against the signatures in the repository. If the unknown executable is not classified under a particular family, then the closest family to that unknown executable is displayed. If the unknown executable is classified as being a part of a malware family then it will be considered a member of that family. The unknown executable joins that malware family and goes through a pruning stage in the clustering phase to see if it will be permanently inserted in that malware family, or else it will be only considered as a

member of the family but will have no effect on the signature. If it passes the pruning stage, then a new signature is created and inserted in the repository.

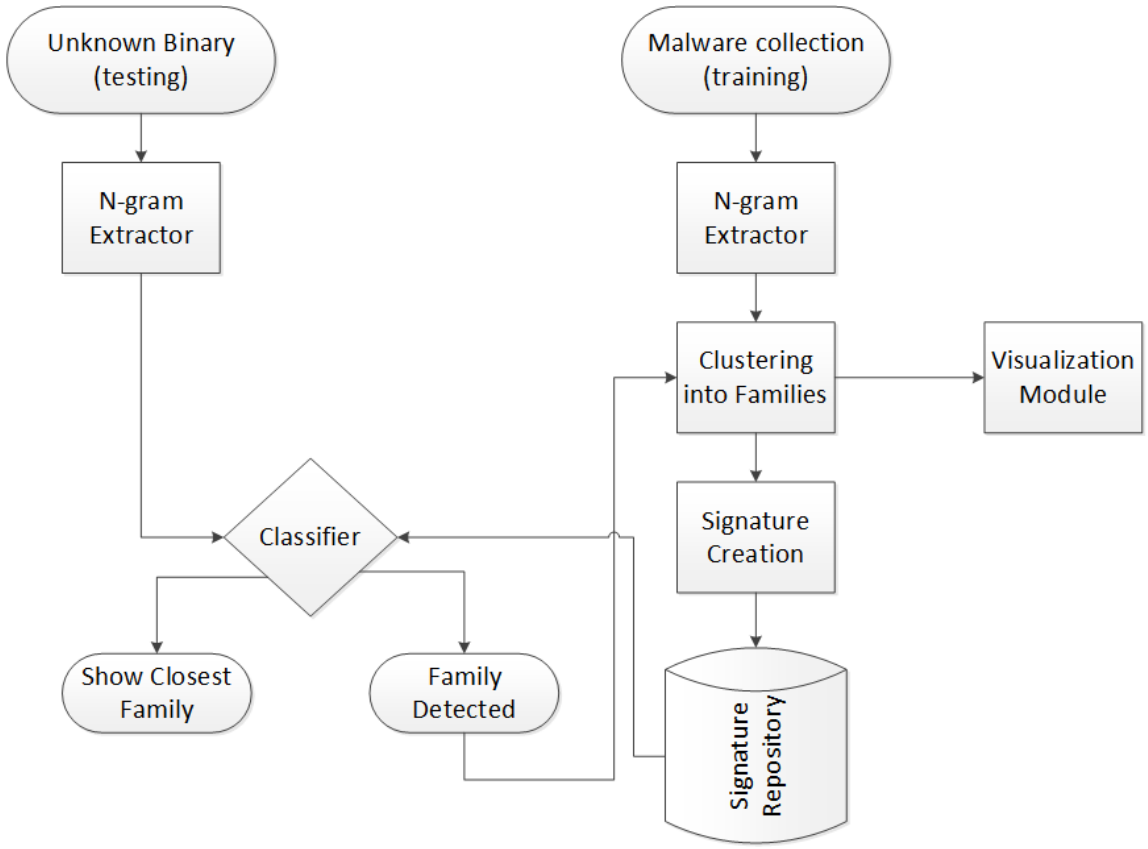

Figure 2: System Architecture

2.3 Collecting Malware

Since over 70% of enterprise work PCs today are running windows XP, as reported from Microsoft [43], I chose to specifically focus on the particular set of win32 binaries given its widespread popularity. Therefore, even though this method can work for any operating system, for the purpose of experimentation I choose to setup the

experiments using Windows XP. This is the process that I used to collect the data for verifying the effectiveness of my malware analysis system. First a set of malware viruses is collected from the VX Heavens website [44]. The initial set is first screened for viruses that are win32 based executables, and all others are removed. The second screening consists of going through the malware viruses and making sure that none were compressed or encrypted. The reason is that if the malware viruses were compressed or encrypted, then the extraction of the opcode instructions inside those binaries would amount to nothing but meaningless data ruining any subsequent analysis conducted. Both the win32 check and the check for uncompressed/unencrypted executables have been screened using a tool called PEiD [45]. The final set of malware should contain various viruses that have been randomly chosen and have passed both checks, as shown in Figure 3. After the set is complete, all the malware viruses are run through the n-gram extractor and the various n-grams are generated for each virus.

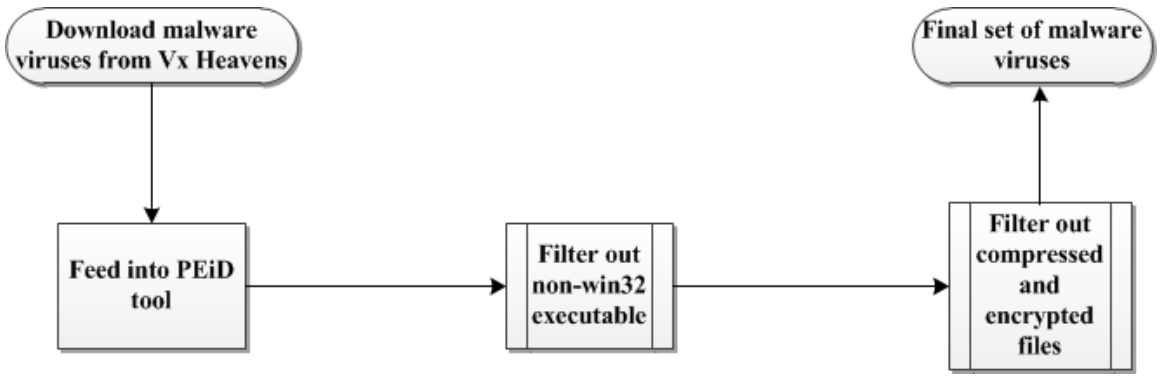

Figure 3: The Malware Collection Process

2.4 The N-gram Extractor

Traditionally n-gram extraction from documents that are in the form of consecutive text was fairly straight forward to extract, but since we are dealing with a different kind of document in the form of computer program code, the extraction becomes more complicated. The structure of code in a computer program is completely different than the structure of a linear document such as a newspaper article that contains text in the form of sentences that start and end from left to right. Programs contain many branches to other parts of the program and thus create a very large number of possible paths that may be taken or executed. The figure is so large that it is not worth pursuing from a computational and efficiency point of view. If it were for one file then it might be justified, but if we are sampling hundreds or even thousands of computer programs then that would be too computationally intensive.

What I propose is to group each program into a set of basic blocks. A basic block is a segment of code that has one entry point (i.e., no code within the segment is the destination of a jump instruction), one exit point, and no jump instructions contained within it (i.e., a sequence of instructions that are all executed if the first one in the sequence is executed). Figure 4 shows an example of a basic block of code taken from a malware. The beginning of the basic block starts at the 'add' instruction and ends at the 'jnz' instruction without any jumps out of or into the basic block. A method presented by [46] defines the general rules of creating basic blocks:

1. The first instruction is a leader.

2. Any instruction that is the target of a conditional or an unconditional jump instruction is a leader.
3. Any instruction that immediately follows a conditional or an unconditional jump instruction is a leader.

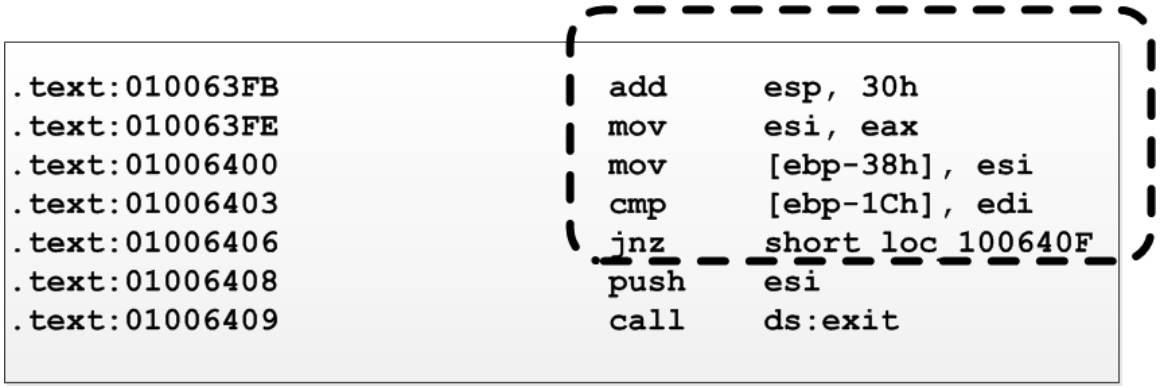

Figure 4: Basic Block Example

After the program has been transformed into many individual basic blocks, I calculated how large each basic block was and created that many different n-grams of it. For example if the basic block consists of 5 instructions then we created 2-grams, 3-grams, 4-grams, and 5-grams for that specific basic block. One might lose some connections between basic blocks since the last instruction of a basic block can continue to the beginning instruction of another basic block to create more n-grams. Therefore, I included an extra n-gram calculation that creates more n-grams at these locations, which I call n-gram joints. These n-grams are named n-gram joints because they are composed of more than 1 basic block.

To extract the n-grams I used software called IDA Pro [47], which is a disassembler that offers its users the ability to create custom made programs and scripts to extract all the needed information from a particular binary executable. Even though there are many disassemblers like [48], [49], [50] I chose IDA for its overall completeness and its strong API package. Figure 5 shows a snapshot of IDA Pro. The language I used was C++ for extracting the n-gram instruction counts and the API function counts. The total amount of code written was around 1,500 lines in C++. For the statistical and analysis part of the dissertation, I used the R statistical software package and wrote around an additional 1,800 lines of code.

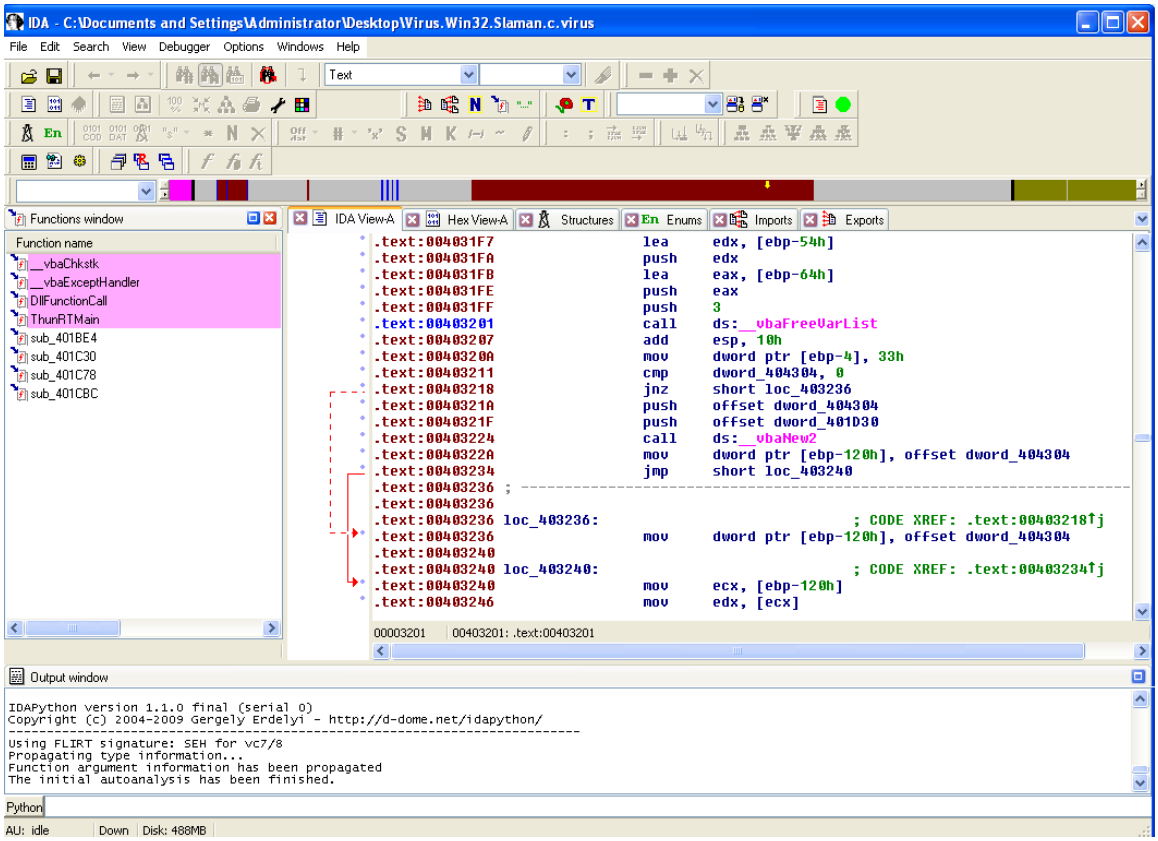

Figure 5: IDA Pro Snapshot

2.5 API Function Call Extractor

To extract the API function call counts in the malware, I had to access the import table of the malware executable. The import table contains all of the functions that the executable will need to access from their respective DLL files [51]. The import table will list the function names, its associated DLL, and its address in memory. IDA Pro gives the programmer access to the import table. Through IDA Pro's functions, I obtained the address of each API function in the import table and made sure to catch any far jump to any of those API addresses in the import table and count how many times each API

address was called. I discovered that there are indirect calls, where a call is made to another instruction that calls the API address in the import table, and I had to adjust my code to take that into account to create an accurate count of each API address called in the malware executable.

2.6 Performance Evaluation

I performed the following measurements to evaluate my proposed techniques:

1. True Positive (TP): The number of correctly classified malware.
2. True Negative (TN): The number of correctly classified goodware.
3. False Positive (FP): The number of goodware incorrectly classified as malware.
4. False Negative (FN): The number of malware incorrectly classified.

To see the number of correctly detected malware I used the True Positive Rate (TPR):

$$\frac{TP}{TP + FN}$$

To see the number of goodware incorrectly classified as malware I used the False Positive Rate (FPR):

$$\frac{FP}{TN + FP}$$

For an accuracy measure of the system I used:

$$\frac{TP + TN}{TP + TN + FP + FN}$$

3 CLUSTERING MALWARE INTO FAMILIES

After having obtained the malware files and running them through the n-gram extractor, the next step was to cluster the malware into families. While this may seem to be a straightforward clustering task, it is far from that, having many unforeseen challenges. The challenges faced include choosing an appropriate similarity measurement, clustering algorithm, selecting parameters, pruning the results for better accuracy, and creating a visual output of the results. These challenges were further increased by making it an essential point that everything must be fully automated without any manual intervention.

3.1 Social Network Analysis of Malware

To create malware social networks I discuss 1 network setup that is the most intuitive, which will be the core of various methods created in this dissertation. A two-mode social network consisting of n-gram opcode instructions and the malicious programs they were extracted from as the two classes of actors. Having a network composed of two actors in the form of opcode instructions and malware is the most intuitive representation. It gives the ability to mathematically represent the network as a matrix. Matrices are easily visualized through graphical representations and are less computationally intensive to apply measurements and functions on.

In this setup I take the previously described two-mode network and convert it into a one-mode network for the sole purpose of applying blockmodeling, which only accepts one-mode networks as input. The ability to reduce a two-mode network into a one-mode network is due to the duality between graphs and their adjacency matrices [52]. For example, in a two-mode binary network composed of terms and documents as actors, the matrix representation is $A_{t\ x\ d}$. Where t is the number of terms and d is the number of documents. Converting this two-mode network into a one-mode network of terms relative to the set of documents, we obtain $B_{t\ x\ t} = A_{t\ x\ d}\ A^T_{d\ x\ t}$. Matrix B represents the one-mode network of terms having the following properties: b_{ij} is the strength of ties between terms i and j where $i \neq j$, and $b_{ii} = \sum_{j=1}^{d} a_{ij}$ is the strength of ties to all terms in i's social network. Matrix A can also be converted into another one-mode network composed of documents relative to the set of terms through the following equation $C_{d\ x\ d} = A^T_{d\ x\ t}\ A_{t\ x\ d}$. Matrix C represents the one-mode network of documents having the following properties: c_{ij} is the strength of ties between document i and document j where $i \neq j$, and $c_{jj} = \sum_{i=1}^{t} a_{ij}$ is the strength of ties to all terms in j's social network.

If looked at closely, one can see a significant resemblance between the conversion of two-mode binary networks to one-mode networks of social network analysis and latent semantic indexing's application of Singular Value Decomposition. In LSI an SVD is applied onto matrix A, $A = TSD^T$. If one computes $A^TA = (TSD^T)^T\ TSD^T = DS^TT^TTSD^T$ and since $T^TT = I_m$ (as stated in the background information on LSI), it simplifies to $A^TA = DS^2D^T$ showing that A^TA depends only on the documents and not

the terms. Equivalently $AA^T = TS^2T^T$ and AA^T depends only on the terms and not the documents. Looking back at the two-mode to one-mode conversion we see that matrix B representing the network of term actor is computed as $B = AA^T$, which has the same expression seen in $AA^T = TS^2T^T$. And matrix C representing the document actor $C = A^TA$ contains the same expression seen in $A^TA = DS^2D^T$. There is a similar isolation of terms and documents in LSI and an isolation of actors in social network analysis [52].

Unfortunately the technique of converting a two-mode network into a one-mode network based on the duality between graphs and their adjacency matrices only applies to binary networks, and the networks I will be dealing with in this research are weighted two-mode networks. To that extent I looked into similarity measurements that accepted weighted two-mode networks and output a one-mode network.

3.2 Similarity Measurement

There are quite a few similarity measurements that deal with weighted networks; choosing which measurement is a critical step that is often neglected in social network analysis. The reason is that this critical conversion essentially compresses or reduces the amount of information, and that always results in a loss of information, which can affect the accuracy of the clustering phase. To this extent I have tested various similarity measurements to see which measurement would be the best fit for the data that are used in this dissertation. Since the social network itself will be composed of malware and their respective n-gram instructions, the similarity measurement must fulfill several criteria for

malware n-gram instructions. To better visualize creating the similarity measurement, we'll look at Figure 6 as an example of two malware being compared on the basis of their instructions. The numbers represent the number of instances that the instruction was found in each particular malware. In this instance, there are two malware being compared using five different instructions, and the output should be a single similarity measure.

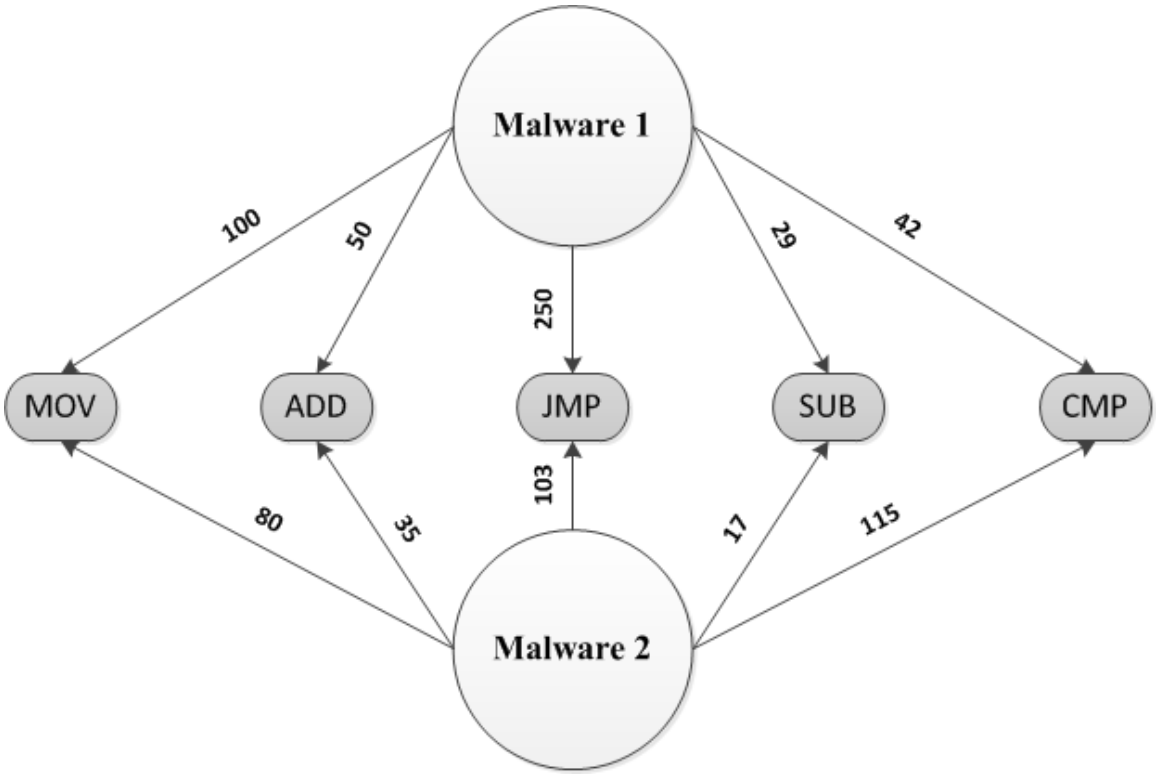

Figure 6: Malware Comparison

What are the criteria that a similarity measurement must fulfill to obtain a more precise measurement for the malware? There are three criteria:

1. It must take into account the sum of the two counts being compared; for example, the result of 100-99 = 1 will not be equal to 2-1 = 1. Instead a more accurate answer of (100-99) / 199 = .005 and (2-1)/3 = .333. The closer the measurement is to zero, the more similar the malware are in respect to that instruction. Intuitively having 100 and 99 counts are very similar but having 2 and 1 counts should not be as similar.
2. The measurement must not result in negative valued answer. For example 99 – 100 = -1 must have its absolute value taken.
3. If both malware have a count of zero for that instruction, then it should not be included in the summation. The reasoning behind this is that there are over 300+ instructions in the IA-32 instruction set, and there is high chance that the programs that are being compared will have many instructions that they both have a zero count for; this in no way shows that they are similar.

The similarity measurement that best satisfies the previous criteria is the Canberra distance measurement. The following is an explanation of how I will use the Canberra distance measurement for malware analysis.

Weighted two-mode network conversion to a weighted one-mode network:

Where, $A =$ set of malware actors

$B =$ Set of n-gram actors

$$AB_{n\ x\ m} \Rightarrow AA_{n\ x\ n}$$

The equation describes the conversion of a weighted two-mode *n x m* matrix into a weighted one-mode *n x n* matrix.

Specifically,

$$aa_{ij} = \sum_{k=1}^{m} \left|((ab_{ik} - ab_{jk})/(ab_{ik} + ab_{jk}))\right|$$

Note that, $aa_{ij} = aa_{ji}$

Similarly,

$$AB_{n\ x\ m} \Rightarrow BB_{m\ x\ m}$$

The conversion of a weighted two-mode *n x m* matrix into a weighted one-mode *m x m* matrix.

Specifically.

$$aa_{ij} = \sum_{k=1}^{n} \left|((ab_{ki} - ab_{kj})/(ab_{ki} + ab_{kj}))\right|$$

Note that: $aa_{ij} = aa_{ji}$

The reasoning behind converting a two-mode social network of opcodes and malicious programs into a one-mode social network is to determine the strength of ties between the set of malware relative to the n-grams extracted from them, and, optionally to calculate the strength of ties between n-grams relative to the set of malware used. From there I can then cluster the malware by using the computed one-mode network as input into a clustering algorithm. After performing clustering, I can create block models from the clusters generated. The block models give the ability to discover similar sets of malware with respect to the set of extracted n-grams.

3.3 Creating the Malware Families

Using the data that was extracted from the malware using the n-gram extractor, a two-mode relational social network is created in the form of an *n x m* matrix. Thus creating two sets of actors in the social network, where the first type of actor is the set of malware viruses and the second is their corresponding set of n-grams (that were generated from the malware virus set). Table 1 shows an example.

Table 1: Two-Mode Social Network of Malware and Their N-grams

	n-gram1	n-gram2	n-gram3	n-gram4	n-gram5	n-gram6
Virus 1	10	1	0	18	0	5
Virus 2	4	9	2	0	9	1
Virus 3	15	1	0	0	2	10
Virus 4	1	8	15	7	0	1
Virus 5	0	5	2	10	13	0
Virus 6	1	0	3	6	4	2
Virus 7	2	2	0	7	4	14

To be able to cluster the malware into families, the two-mode social network must be converted into a one-mode network through the use of the Canberra similarity measurement. Table 2 shows a one-mode social network that was created by applying the Canberra measurement. Blockmodeling also relies on the network being a one-mode social network.

Table 2: One-Mode Social Network of Malware

	Virus 1	Virus2	Virus3	Virus4	Virus5	Virus6	Virus7
Virus 1		49.2753	32.68428	58.52357	49.11164	112.2251	24.0112
Virus 2	49.2753		118.8641	60.33276	43.27488	84.10738	38.10586
Virus 3	32.68428	118.8641		27.95267	107.7099	95.59009	28.27188
Virus 4	58.52357	60.33276	27.95267		37.6921	51.62867	96.79418
Virus 5	49.11164	43.27488	107.7099	37.6921		97.71925	29.43769
Virus 6	112.2251	84.10738	95.59009	51.62867	97.71925		67.77746
Virus 7	24.0112	38.10586	28.27188	96.79418	29.43769	67.77746	

The resulting one-mode social network will be composed of only 1 set of actors, which are the malware actors. The one-mode social network represented as a matrix will now contain the strength of ties or similarities between the set of malware relative to the n-grams extracted from them. Now enters blockmodeling's first phase, where the malware actors will be clustered into classes through the use of a clustering algorithm.

3.3.1 DBSCAN and Parameter Selection

Choosing a particular clustering algorithm is an important decision that requires a thorough understanding of the clustering task at hand, which in this case is malware family clustering. Every clustering algorithm has a set of parameters that it requires to successfully cluster the data set into classes. A particular parameter that should be

avoided for clustering malware into families is the usually labeled ‘k’ parameter, which is the predetermined number of classes to cluster into so that a clustering algorithm can run. A couple clustering algorithms that rely on the parameter ‘k’ are k-means [53] and k-medoids [54]. The reason why these algorithms should be avoided is that the main goal of this research is to create an automated system that does not rely on manual intervention by the user. There are papers that discuss estimating the ‘k’ number of classes [55], but still rely on the user to provide a range where ‘k’ may fall between. To this extent I have chosen the DBSCAN [56] algorithm to perform the clustering.

As discussed previously, DBSCAN has two parameters that it takes in along with the one-mode similarity matrix. The first is MinPts, which is the minimum number of points that a class must have. The second is the radius Eps, which is used to measure if a point lies within the neighborhood of other points and vice versa. For the MinPts parameter, I choose a value of 2 because the whole purpose of the clustering phase is to find families of malware, and a family must have at least two members to constitute a family. Hence, all malware that have less than two members (including themselves) in a class will be considered noise by DBSCAN, but in this dissertation they will be treated as the only member of their family. The Eps parameter, on the other hand, requires a more in-depth look.

In [56] they created a heuristic method to estimate the parameter of Eps by means of graphical analysis. The method is inspired by the idea that points in a cluster have almost the same value for their k^{th} nearest neighbor, and that points that have their k^{th} nearest neighbor at a farther distance are considered noise. The method itself starts with

calculating the k^{th} nearest neighbor for each point. The points are then sorted by descending order of their k^{th} distance values. The resulting graph is called the sorted k^{th} distance graph. From the graph a threshold point is chosen that will maximize the k^{th} distance in the leanest cluster. To estimate this threshold point, a scan of the k^{th} distance graph reveals a "valley"; that point will be the beginning of this valley. When the threshold point is chosen, the Eps parameter is set to the point's k^{th} distance, and the parameter MinPts is set to k. This makes every point that has a k^{th} distance that is smaller than or equal to Eps, a core point.

Figure 7 shows a k^{th} distance graph for 77 malware that are to be clustered into families. There are two plots on the graph with k values of 2 and 4. The threshold point is visually calculated by a steep change in the value of k in the graph; the point where that happens will be designated as the threshold point. The valley shape can be clearly seen for both k=2 and k=4 on the left side of the graph. The arrow points to the threshold point with all the points after the threshold point being noise, and the rest of the points being a part of some cluster. In [56] they claim that having values of $k > 4$ does not show much difference than the $k = 4$ plot, and because larger values of k require much more computation, they let $k = 4$ for all 2-dimensional data. In this dissertation the definition of a family has been set to having at least two or more malware; therefore, the value of k will always be 2. In this graph the threshold point is about 145, and it can be seen that both plots of k=2 and k=4 start taking a steep rise.

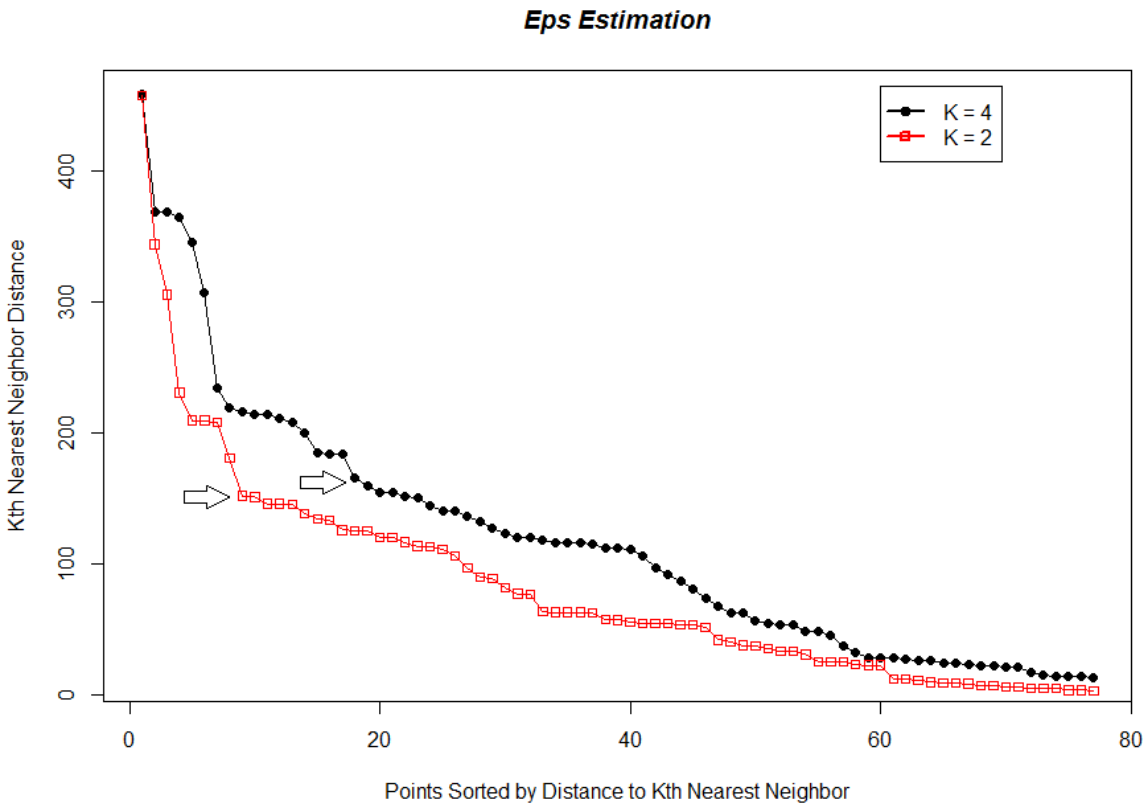

Figure 7: Eps Estimation

The previous heuristic method is one way of estimating the parameters manually, but in this dissertation I am focusing on a fully automated system that is mostly free of any manual intervention. In this light, I sought out another method based on the sorted k^{th} distance graph. Instead of visualizing the threshold point, I take the mean of the k^{th} nearest neighbor of all the points in the dataset. Another variation is taking the mean of the average of all values till the k^{th} nearest neighbor. The second variation usually comes out with a value that is smaller than the first. Both methods output values that are more accurate than the visual sorted k^{th} distance graph method. The second method is better for

smaller values of k, and because the value of k has been set to 2, for this dissertation the second variation would be a better choice in malware clustering.

Mean of the k^{th} nearest neighbor,

$$Eps = \overline{kth\_near(dataset)}$$

Average of all values till the k^{th} nearest neighbor,

$$Eps = \overline{avg\_kth\_near(dataset)}$$

Having figured out the appropriate values for the DBSCAN clustering algorithm, I fed the Eps, MinPts, and the similarity matrix based on the Canberra similarity measurement into the DBSCAN implementation for R [57]. The output is a set of malware clusters where each cluster represents a malware family. Unfortunately, clustering algorithms, which includes the DBSCAN algorithm, are estimation based and are not perfect algorithms. Therefore, the current families are considered to be in their semi-completed form. To finalize the families, a pruning algorithm has been developed for such a task.

3.3.2 Family Pruner

Algorithm 1 FamilyPruner(*OrgMatrix, FamiliesList*)

Input: The [malware X features] *OrgMatrix,*
A list of malware families *FamiliesList*
Output: A list of pruned malware families *NewFamiliesList*

```
1:  begin
2:      while FamiliesList not finished do
3:          CurrentFamily = FamiliesList[i]
4:          PrunedFamily = FamiliesList[i]
5:          MatrixA ← ReduceMatrix(OrgMatrix, CurrentFamily)
6:          MatrixB ← CommonFeatures(MatrixA)
7:          while CurrentFamily still needs pruning do
8:              PreviousFamilyState = CurrentFamily
9:              CurrentFamily = PrunedFamily
10:             if (MatrixB >= 20 common features) then
11:                 CurrentMatrix = MatrixB
12:             else
13:                 CurrentMatrix = MatrixA
14:             end if
15:             NormMatrix ← NormalizeMalwareVectors(CurrentMatrix)
16:             AvgVector ← AverageVector(NormMatrix)
17:             CosineMatrix ← CosineSimilarity(AvgVector, NormMatrix)
18:             SimilarityVector ← SimilarityVector(CosineMatrix)
19:             if ((Mean(SimilarityVector) < 90%)
20:                 || (any element in SimilarityVector < 90% of Mean(SimilarityVector))) then
21:                 PrunedFamily ← Prune(CurrentFamily, SimilarityVector)
22:                 MatrixA ← ReduceMatrix(OrgMatrix, PrunedFamily)
23:                 MatrixB ← CommonFeatures(MatrixA)
24:             else if (((CommonFeatures(CurrentFamily) < 30% increase over
25:                 CommonFeatures(PreviousFamilyState))
26:                 || (ExactFeatures(CurrentFamily) < 30% increase over
27:                 ExactFeatures(PreviousFamilyState))
28:                 || (ExactFeatures(PreviousFamilyState) > 10)))
29:                     CurrentFamily = PreviousFamilyState
30:                     Finished pruning CurrentFamily
31:             else
32:                     Finished pruning CurrentFamily
33:             end if
34:         end while
35:     NewFamiliesList[i] = CurrentFamily
36:     i = i + 1
37:     end while
38:     return NewFamiliesList
39: end
```

Algorithm 1 has been created to prune the families created in the clustering phase. There are two parameters inputed, *OrgMatrix* and *FamiliesList*. The first parameter *OrgMatrix* is the original malware x features matrix containing the feature counts for each malware. In this dissertation, the features may be either n-gram instructions or single API calls. The second parameter is *FamiliesList*, which is a list of families. The output is a list of pruned malware families named *NewFamiliesList*.

Algorithm 1 loops through the list of families in *FamiliesList* until it has gone through the entire set of families. In each loop, it calls the function *ReduceMatrix()* and reduces the original matrix *OrgMatrix* to only the set of malware found in *CurrentFamily*, which holds the value of the current family in *FamiliesList*. That same function then reduces the matrix further by removing all features that have count values of 0 for all the malware. It then calls the function *CommonFeatures()* on the resulting reduced matrix and also further reduces the matrix by keeping only the common features shared by all the malware (every malware must have at least a count of 1 to keep the feature). After reducing the matrix, the pruning loop is reached, and it will continue to loop until the pruning has been completed for the current family. First, a check is made to see if the reduced matrix *MatrixB* has at least 20 features, which represent the common features among the current family of malware. If it is above 20 features, then the reduced matrix *MatrixB* is kept else the matrix *MatrixA* with a less reduced form is taken. The reasoning behind this check is to make sure that there are enough features in the matrix, and even though the matrix outputted from *CommonFeatures()* is the optimal choice,

sometimes malware families may not have enough common features at their pre-pruned state, making the subsequent analysis weak in its results. The chosen matrix is then normalized in *NormalizeMalwareVectors()*, where each malware is represented as a unit vector that sums to 1. An average vector is computed on the resulting normalized matrix through the function *AverageVector()*. Then a cosine similarity matrix is created by the function *CosineSimilarity()*, thus comparing the malwares to each other and to the average malware vector. The *SimilarityVector* is taken from the resulting cosine similarity matrix, which shows the similarity between the malware and their average vector.

Next a couple of checks are made to see how strong the current family is. The first is by checking if the mean of the vector of similarities (*SimilarityVector*) between the current malware family members and their average vector is over 90%, and if any of the similarities (in *SimilarityVector*) are smaller than 90% of the mean of *SimilarityVector*. If any of the previous checks are true, then that means this current family still needs more pruning. To prune the current family, the following two variables *CurrentFamily* and *SimilarityVector* are sent to the *Prune()* function, which will be explained after the completion of Algorithm 1. After the pruning function has been completed the newly created family is saved and its matrix is also updated showing the recent family member change.

If both checks have been passed, then this assures that the current malware family members have a similarity mean above 90% to their average vector. And all the similarities are at least 90% of that mean. This gives an indication of the strength that the

members currently have, but sometimes pruning can keep going into an everlasting cycle to try and a stronger and stronger member base. Therefore, three checks have been created to try and bound the pruning process. The first check is to compare the current family's common features with the previous prune (if there was a prune). It checks if the current family's number of common features are smaller than 30% of the pre-prune number of common features, which means that the current prune has not significantly gotten stronger. The second check sees if the current family's number of exact features is smaller than 30% of the pre-prune number of exact features, which means that the current prune has not significantly gotten the family any stronger. What is meant by exact features is that the malware family members all have the exact same counts for a particular feature. The final check sees if the previous prune had at least 10 exact features, which means it is fairly strong. The reason this final check is applied is to prevent the algorithm from creating smaller than needed families. It is important to note that larger families are preferred over smaller families, so as to create a more efficient signature that encompasses a larger number of malware. If one of the previous three checks evaluates to true, then that means the last prune to this malware family set was not a worthwhile prune and did not add any significant strength to the family. Therefore, the previous pre-prune state of the family is taken and the pruning loop is exited. If all of the three checks evaluate to false, then the currently pruned malware family set is kept, because that means the last prune has increased the common and exact features by at least 30%, and the pre-prune state had less than 10 exact features. The pruning loop is then exited.

Once the pruning loop is exited, the final state of the current malware family is stored in *NewFamiliesList* and the loop continues until all the malware families have been iterated through. *NewFamiliesList* is then returned to the calling function. The newly pruned set of families is passed on to Algorithm 3 for subfamily creation, but before introducing Algorithm 3 an explanation of the pruning function given in Algorithm 2 will be given.

Algorithm 2 Prune(*CurrentFamily*, *SimilarityVector*)

Input: The malware family members *CurrentFamily*
The malware family vector of similarities to their average vector *SimilarityVector*
Output: The pruned malware family vector *CurrentFamily*

```
1:  begin
2:      RemoveZero(SimilartyVector)
3:      Max ← Maximum(SimilarityVector)
4:      for all n in vector SimilarityVector do
5:          Y[n] = Max - SimilarityVector[n]
6:      end for
7:      Avg ← Mean(Y)
8:      for all n in vector Y do
9:          if (Y[n] > 1.5(Avg)) then
10:             remove Y[n]
11:             remove CurrentFamily[n]
12:         end if
13:     end for
14:     return CurrentFamily
15: end
```

Algorithm 2 is a pruning function that prunes the current malware family. It accepts two parameters—the current malware family *CurrentFamily* and the malware

family vector of similarities to their average vector *SimilarityVector*. The output is a pruned malware family that will be returned as *CurrentFamily*. The start of the algorithm removes any values of 0, because the similarity score of the average vector to itself has a value of 0. The maximum similarity score *Max* is then found and each similarity score is subtracted from the maximum score. This creates a new vector *Y* composed of max - similarity scores. The goal of the previous calculation is to find out which members of the malware family should be pruned. To visualize the previous calculations, a graph is displayed in Figure 8 with all the values in *Y* plotted, and it can be seen that there are two points that are significantly higher than the others. These points represent the malware family members that should be pruned. But the system designed in this dissertation does not rely on any manual interventions; therefore, an automatic method must be created. To this extent vector *Y* is taken, and its mean is found and for any values in the vector of *Y* that are larger than 1.5 times the mean are removed. This effectively takes out any family members that show a weak similarity to the malware family's average vector compared with the other family members. The final pruned family is returned as *CurrentFamily*.

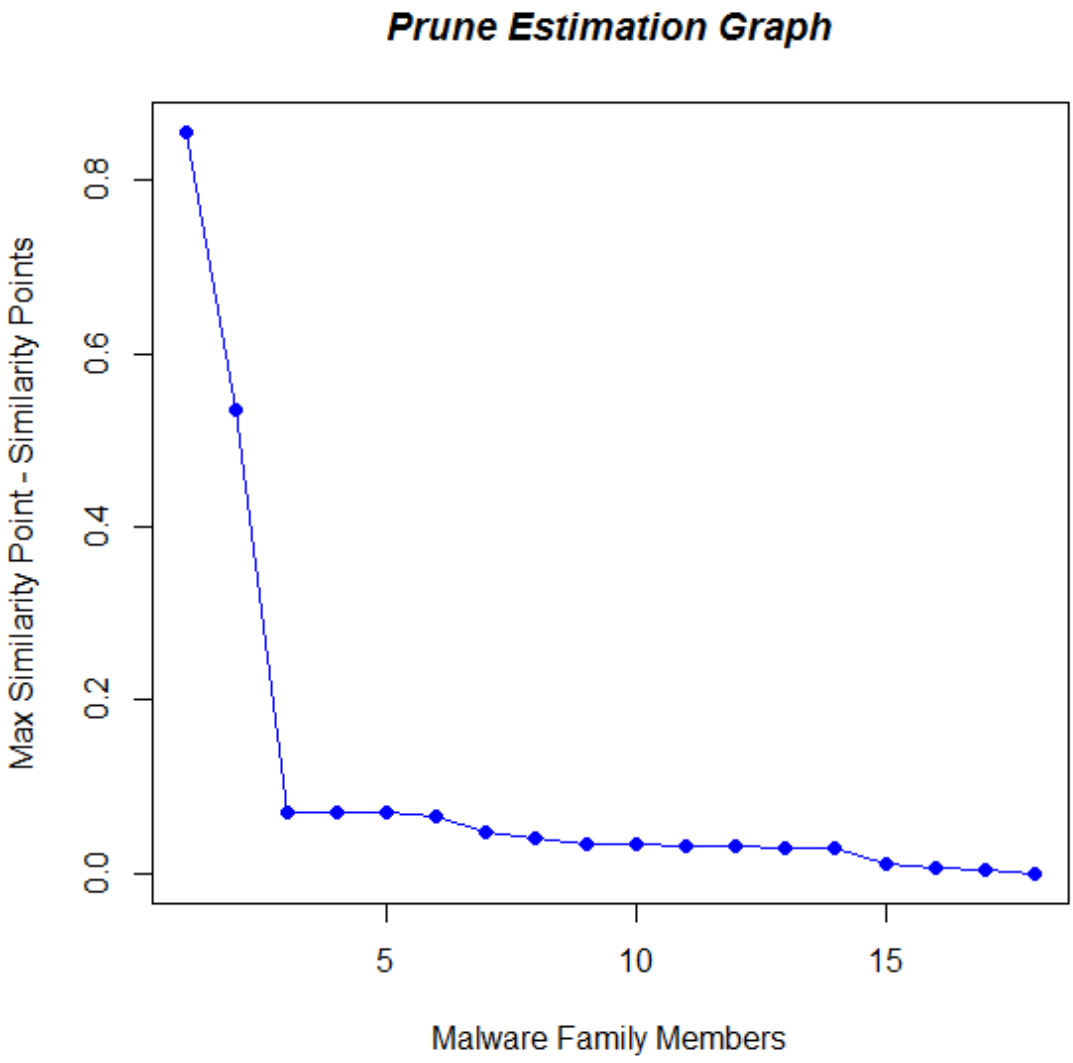

Figure 8: Prune Estimation Graph

Algorithm 3 describes the subfamily creation that takes place after the pruning. Families are only broken up into subfamilies when they have less than 15 exact API features. The threshold of 15 can be adjusted as needed. The function Subfamily() takes in two parameters, *OrgMatrix* and *FamiliesList. OrgMatrix* has been described previously and *FamiliesList* is the list of pruned malware families. The Subfamily()

function outputs a new set of pruned families that have been split into subfamilies and returned in *NewFamilies*. The algorithm starts with a while loop that continues until all the families have been seen. Within the loop a check is made to ensure that the current family is larger than two family members, since having two or less family members would mean it is fruitless to split the family into individual single member families. Then the *OrgMatrix* is reduced through the function *ReduceMatrix()*, which was previously described. Then the previously outputted matrix is further reduced by applying *CommonFeatures()*, which was also described previously. After the previously described functions are applied, *MatrixB* represents the current malware family and the common features shared by all the family members (every malware must have at least a count of 1 for each feature). MatrixB is then given to a function named *DissimilarityCounts()* that outputs the dissimilarity scores of the malware family members based on the exact feature counts. Exact feature counts require that they have the same count for each feature. Two malware that have count values of 0 are not considered an exact count, since zero does not show that they are truly similar. Once the dissimilarity scores have been computed they are fed into the DBSCAN function, and the resulting clusters will represent the subfamilies. The new subfamilies are saved, and after the loop has gone through all the families the new set of families are returned in the variable *NewFamilies*.

Algorithm 3 Subfamily (*OrgMatrix*, *FamiliesList*)

Input: : The [malware X features] *OrgMatrix*,
A list of pruned malware families *FamiliesList*
Output: A new set of pruned families that have been split
into subfamilies *NewFamilies*

```
1:  begin
2:      i = 1
3:      while FamiliesList not finished do
4:          if ((FamiliesList[i] > 2)  && (API_features < 15)) then
5:              CurrentFamily = FamiliesList[i]
6:              MatrixA ← ReduceMatrix(OrgMatrix, CurrentFamily)
7:              MatrixB ← CommonFeatures(MatrixA)
8:              MatrixC ← DissimilarityCounts(MatrixB)
9:             NewFamilies[i] ← DBSCAN(MatrixC)
10:         end if
11:         i = i + 1
12:     end while
13:     return NewFamilies
14: end
```

After the completion of the *FamilyPruner()* and *Subfamily()* functions the malware families have been pruned and, if appropriate, split into subfamilies. Creating the malware families is a critical step in creating the signatures that will be discussed in the next chapter.

3.3.3 Blockmodeling

At this step all malware families have been created, which leads to the creation of a blockmodel. What the blockmodel offers is the ability to visually see the relationship strength between the malware families and the relationship strength within each malware

family. To create the blockmodel, the malware have to be organized by family on the matrix plot. Each family represents a block on the matrix. Figure 9 shows a sample of a blockmodel displaying malware listed by family.

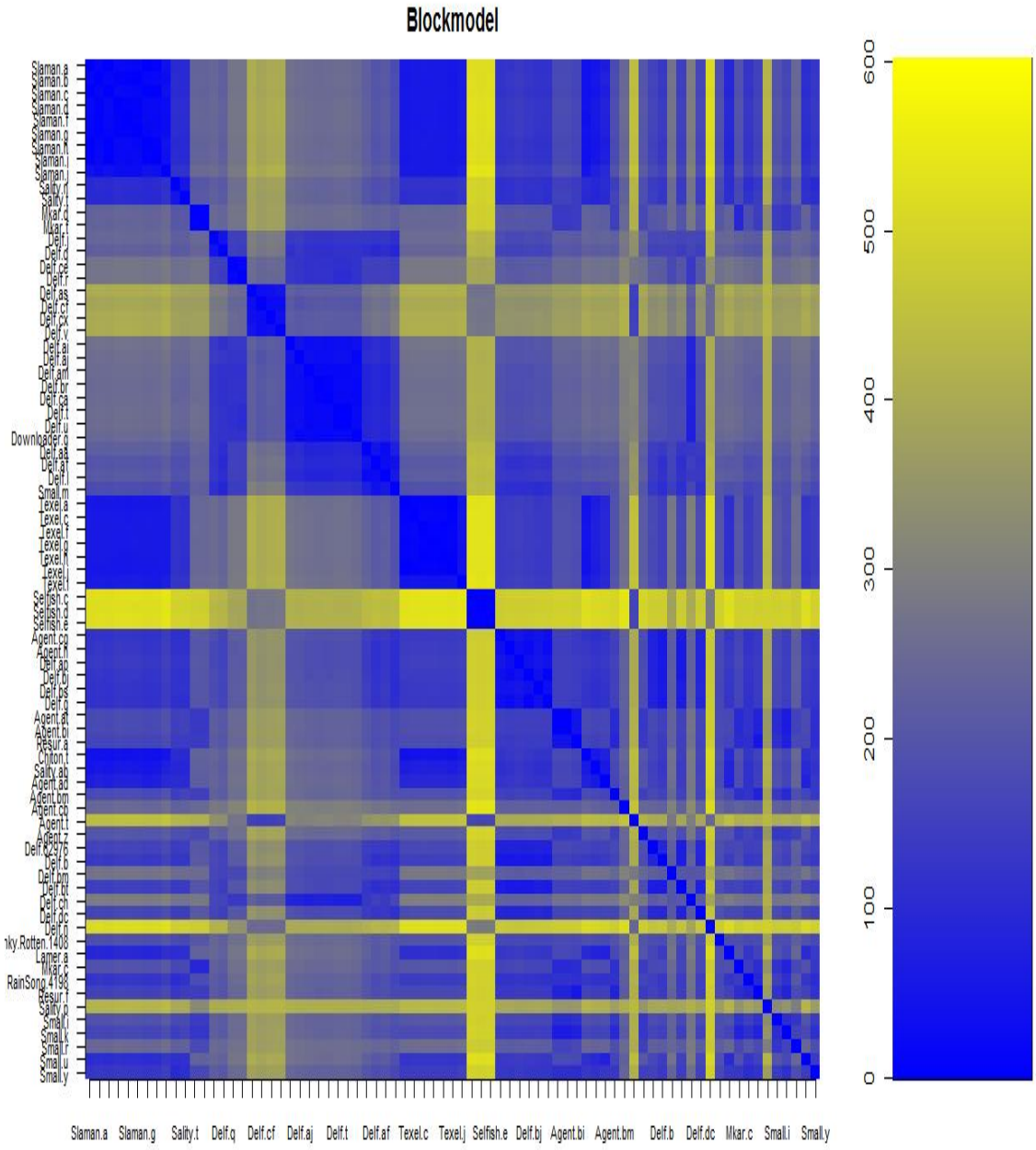

Figure 9: Blockmodel

The blockmodel itself is a visual representation of a similarity matrix, and in this case the dissimilarity scores are between different malware grouped by their families. The color index on the right shows the strength of similarity as a color. Similarity scores

that are closer to 0 mean that the similarity is high, and scores closer to 600 mean that they are very low. For similarity scores that get closer to 0 the color is blue, and for scores towards 600 the color is more yellow. To see the family blocks, first locate the family members on the y-axis and find their positions on the x-axis. If the family is a strong family, it should show a blue colored block covering the whole family. To see if one family is related to or is similar to another family, the first family is located on the y-axis and the second family is located on the x-axis. If there is a blue colored block, then the families are very similar.

Looking at the Figure 9, it is apparent that the blockmodel is a little difficult to read because the blocks tend not to be distinctly visible and the colors contain different shades that reduce visibility. To address this problem a new algorithm, Algorithm 4, has been created to make the graph more readable as well as to make this step completely automated without the need for manual intervention.

Algorithm 4 Simplify (*DissimilarityMatrix, FamiliesList*)

Input: : The [malware X malware] *DissimilarityMatrix* (organized by families),
A list of pruned malware families *FamiliesList*
Output: A new simplified binary matrix in *DissimilarityMatrix*

```
1:  begin
2:      highest = 0
3:      i = 1
4:      while FamiliesList not finished do
5:          if (highest < MaxDissimlarity(FamiliesList[i])) then
6:              highest ← MaxDissimlarity(FamiliesList[i])
7:          end if
8:          i = i + 1
9:      end while
10:     while DissimilarityMatrix not finished do
11:         if (DissimilarityMatrix[i,j] <= highest) then
12:             DissimilarityMatrix[i,j] = 0
13:         else
14:             DissimilarityMatrix[i,j] = 1
15:         end if
16:     end while
17:     return DissimilarityMatrix
18: end
```

Algorithm 4 takes in as input the dissimilarity matrix *DissimilarityMatrix* ordered by malware families and a list of families *FamiliesList*. The output is a new binary matrix returned in *DissimilarityMatrix*. The algorithm first tries to find the highest dissimilarity score of each family, and then finds the highest dissimilarity score among the highest of all the families. The emphasis here is to only find the highest dissimilarity only within the families and not the malware that have not been clustered into families. The highest dissimilarity value score *highest* will be used as a baseline to determine which scores will have a value of 0, indicating similarity or a value of 1 indicating a lack of similarity. If

any score is smaller than or equal to *highest*, then that means it has a value of 0 and anything higher than *highest* will have a value of 1.

If the dissimilarity matrix used to create Figure 9 was run through Algorithm 4, and the resulting matrix used to create a blockmodel, it would look like Figure 10. Just like the previous blockmodel, the yellow color represents dissimilarity and the blue color represents similarity. The difference between the previous blockmodel and this newly created blockmodel is that it is uncluttered and very clear in showing the similarities between families as evident from the clearly defined blue blocks.

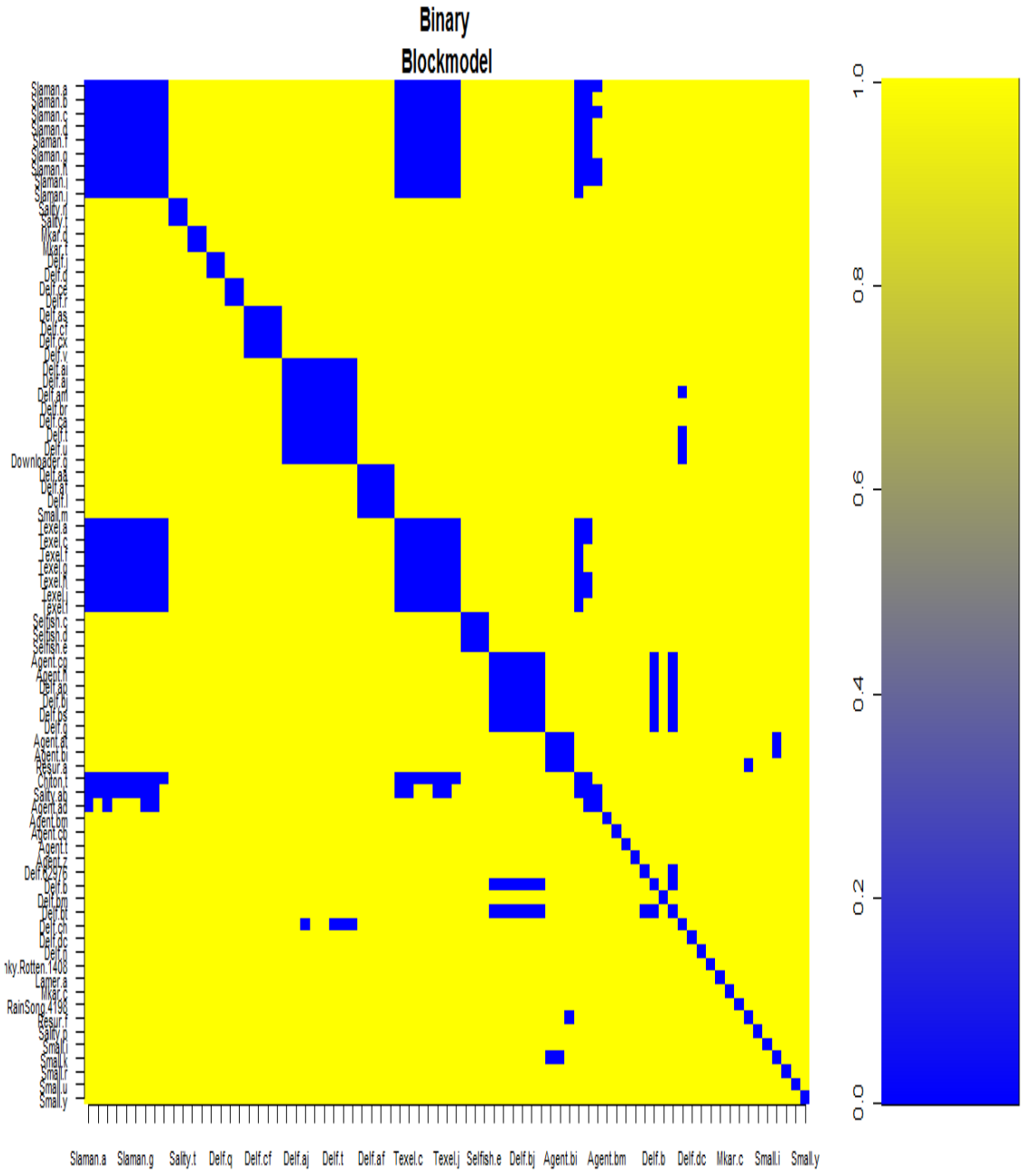

Figure 10: Clarified Blockmodel

The next phase after blockmodeling is to create an image matrix that will determine the relational ties between the newly created positions (families) of malware.

To determine the relational ties Algorithm 5 was created. It accepts a binary matrix that has been outputted by the function *Simplify()* and a list of pruned families *FamiliesList*. Algorithm 5 goes through all the families and compares each family to the other. In particular it computes how many similar members from one family are to the other family being compared. It saves this number under *SimilarCount* through the use of the function *SimilarMembers()*. The total number of family member comparisons is calculated by multiplying the number of members in both families together and saved in *Total*. Then a percentage value is computed by dividing the *SimilarCount* by the *Total*. If the percentage is larger than or equal to 80%, then the two families are considered similar to each other and a value of 1 is given the cell in *IMatrix* indicating a link is present. Often the values in the image matrix are called oneblocks or zeroblocks because they are shown as blocks in the blockmodel. It then returns the image matrix *IMatrix*.

Algorithm 5 ImageMatrix (*BinaryMatrix, FamiliesList*)

Input: : A binary matrix *BinaryMatrix*,
A list of pruned malware families *FamiliesList*
Output: An image matrix returned in *IMatrix*

```
1:  begin
2:      i = 1
3:      while (i <= length(FamiliesList[i])) do
4:          j = i + 1
5:          while (j <= length(FamiliesList[j])) do
6:              SimilarCount ← SimilarMembers(FamiliesList[i], FamiliesList[j], BinaryMatrix)
7:              Total = (length(FamiliesList[i]) * length(FamiliesList[j]))
8:              if ((SimilarCount/Total) >= .8) then
9:                  IMatrix[i,j] = 1
10:                 IMatrix[j,i] = 1
11:             else
12:                 IMatrix[i,j] = 0
13:             end if
14:             j = j + 1
15:         end while
16:         i = i + 1
17:     end while
18:     return IMatrix
19: end
```

Table 3 shows a sample image matrix of the malware families. The image matrix serves the purpose of showing how each family is related to other families.

Table 3: Image Matrix of Malware Families

	Family1	Family2	Family3	Family4	Family5	Family6	Family7	Family8	Family9	Family10	Family11	Family12
Family1	0	0	0	0	0	0	0	0	0	0	0	0
Family2	0	0	0	0	0	0	0	0	0	0	0	0
Family3	0	0	0	0	0	0	0	0	0	0	0	0
Family4	0	0	0	0	0	0	0	0	0	0	0	0
Family5	0	0	0	0	0	0	0	0	0	0	0	0
Family6	0	0	0	0	0	0	0	0	0	0	0	0
Family7	0	0	0	0	0	0	0	0	0	0	0	0
Family8	0	0	0	0	0	0	0	0	0	0	0	0
Family9	0	0	0	0	0	0	0	0	0	**1**	0	0
Family10	0	0	0	0	0	0	0	0	**1**	0	0	0
Family11	0	0	0	0	0	0	0	0	0	0	0	0
Family12	0	0	0	0	0	0	0	0	0	0	0	0

This provides a great wealth of information to the malware analyst using a fairly simple and fully automated procedure a malware analyst can see and further analyze relationships between malware that were never apparent. Also from the image matrix one can create the equivalent graph of it and better visualize the analysis of the malware families as shown in Figure 11. In this example, the only relationship that existed was between family 9 and family 10.

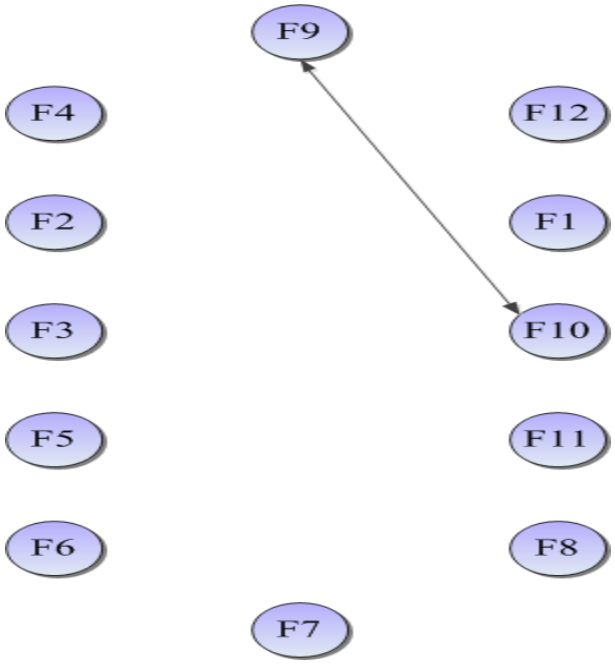

Figure 11: Graph of Image Matrix

4 SIGNATURE CREATION AND CLASSIFICATION OF MALWARE

The previous chapter dealt with creating families of malware, which leads to the final step in this automated system, creating signatures that will distinctly identify each family. Each signature should encompass the most defining traits that make up that particular family.

4.1 Creating the Signatures

There are two different attributes that have been extracted out of the set of malware for the purpose of creating the signatures; they are the opcode instruction mnemonics and the API calls that are called within each executable. The reason behind choosing them as the primary features is that when focusing on what truly makes a program distinct from other programs it would have to be the opcode mnemonic instructions because it is essentially the building blocks of a program. The opcode instructions provide the logical details and flow of a program. It can be thought of as a feature that describes the lower abstractions of the program. The API calls are the second chosen type of features, and they are basically a set of functions provided by the Windows operating system to let user mode processes request various services from the kernel [58]. Specifically the API calls provided by the Windows operating system covers

many different areas like memory management, network management, multimedia, and file management. Figure 12 shows an example of opcode instructions and API calls.

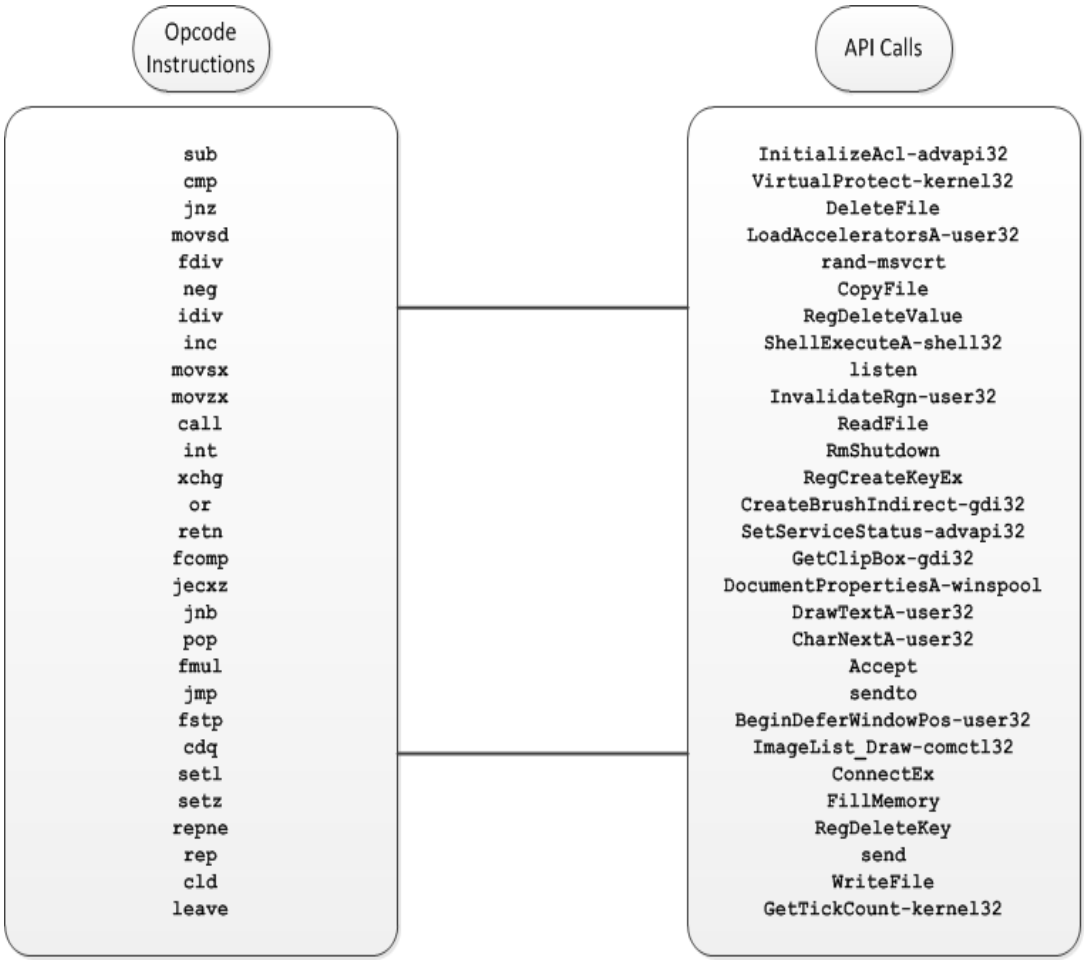

Figure 12: Opcode Instructions and API calls

As can be seen from Figure 12, the opcode instructions like “jnz” (which means jump if not zero) is a control flow instruction. Instructions like “movsd” show movement of data around. These instructions show the low level actions of a program, and so as an example many “mov” instruction may indicate that the program is moving a lot of data

around, which might indicate a problem. On the other hand API calls are at a much higher level of abstraction. The API function calls show the program's behavior. Often the function names themselves can tell a lot about what the program is trying to accomplish. For example the function CopyFile is clearly an indication that the intent to copy a file is there. Another example is the Accept function, which provides the ability to accept incoming network connections on a socket. These can be signs that a program's behavior is suspicious. Together these two feature groups provide a kind of summary to the program they were extracted from. For that reason, I rely on these features as the components of a general signature for the malware families.

The amount of information that I can extract from the feature counts are two types. The first is the number of common features that a family shares, which means that each family member must have at least 1 count for a specific feature to be considered as a common feature. The second piece of information is the number of exact feature counts, which means that each family member must have the exact number of counts for that specific feature to be considered an exact feature.

Table 4: Example of Exact and Common Features

	Feature1	Feature2	Feature3	Feature4	Feature5
Malware1	1	0	3	3	4
Malware2	1	3	2	10	4
Malware3	1	19	4	20	4
Malware4	1	7	1	0	4
Malware5	1	2	5	1	4

Table 4 shows an example of a malware family having two exact-features circled in blue and one common-feature circled in red. Feature2 and feature4 are not considered common features because they both have at least 1 malware member with a count of 0.

4.2 Malware Classification

To complete the automated system, I must create an appropriate classifier. The classifier module's main task is to accept new samples in the form of program executables and decide whether each executable is a part of any of the families that have been formed from previous encounters. Each family will have a distinguishable signature that sets it apart from other families. I will present three different types of classifiers starting with the odds ratio technique, exact feature technique, and the blockmodel comparison technique. I will be using only the exact feature technique in my experimentation section.

4.2.1 Ratio of Averages

In this technique, I gather the malware families that I had clustered together using the methods discussed in chapter 3 of this dissertation. I then represent each malware sample from the malware families as a set of feature counts over the total number of features in that sample. An example is shown in Figure 13, where a single malware family is displayed along with the four members that make up this family. All members will have their features represented as a ratio of the feature count over the total number of features in that malware member. And in this figure the features happen to be bigrams.

Although there are only three bigrams in the figure “MOV JMP”, “RET SUB”, and “ADD JMP” it should extend to all the features for that malware member. The figure displays the concept in a visually clear presentation, but when implementing this method it will be in matrix form.

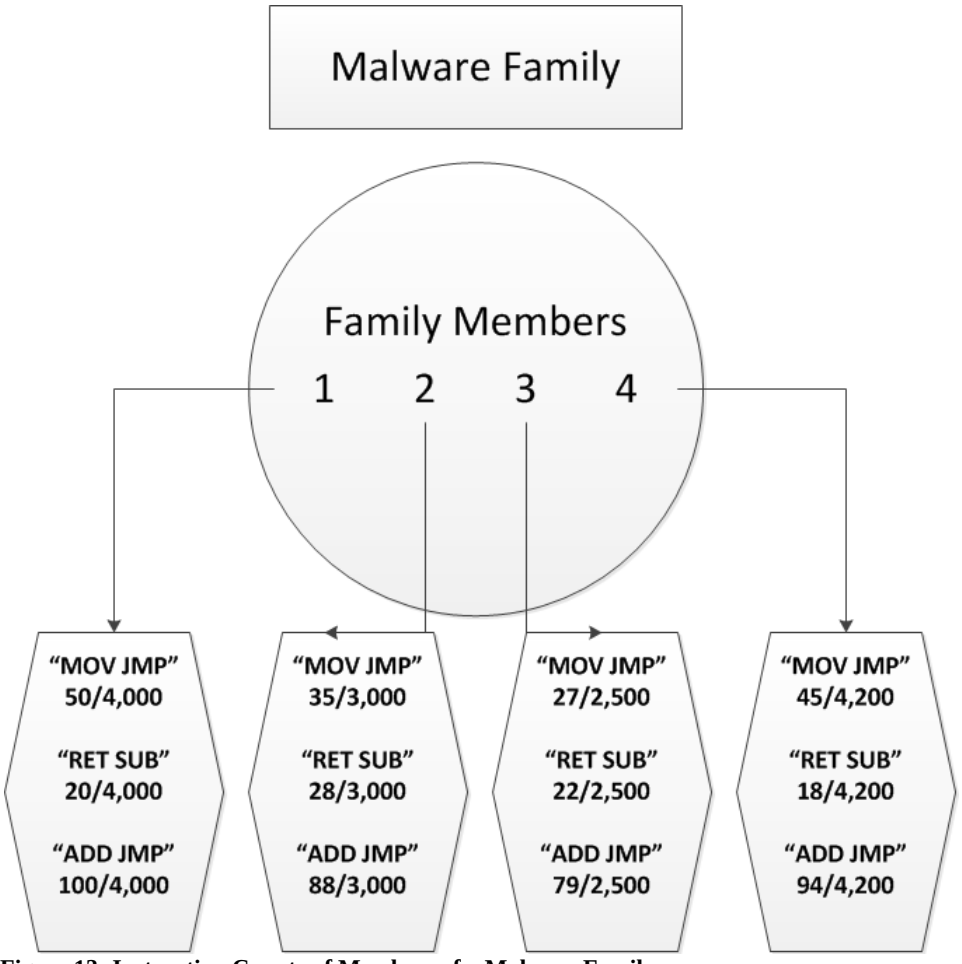

Figure 13: Instruction Counts of Members of a Malware Family

The next step in the process is to unify all of the malware members in the family into a single entity that will represent the family. To do this I will create an average

representative for each family, and it will be the representative malware of that family. This representative takes the average of all the features of the malware members in the family and averages their total feature counts. Therefore, the previous malware family will have a representative as shown in Figure 14.

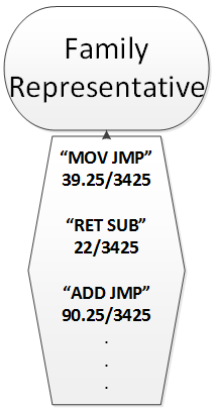

Figure 14: Family Representative

Now that each family has its own representative, a classifier is created in the form of an equation. The following equation is the basis of a classifier, which uses the representative's feature counts as ratios of each other in different arrangements divided by the representative's number of feature counts.

$$\frac{[\frac{feature1}{feature2}+\frac{feature3}{feature8}+\frac{feature1}{feature20}+\cdots]/\text{average \# of features in malware Family \#1}}{[\frac{feature1}{feature2}+\frac{feature3}{feature8}+\frac{feature1}{feature20}+\cdots]/\text{\# of features in the test binary}}$$

Every time a new executable is to be tested it is setup in the same arrangement as listed in the classifier, and each representative is divided by the new executable. The closer the value is to 1 the more similar the executable is to that family. Therefore, the family that has the smallest value outputted by the classifier will be the closest family to that executable. Of course, that does not mean that this unknown executable will be added to the family; it merely says that out of all the families in the database this is the closest family to the executable. For the executable to be put within a family, it must be able to survive a pruning. If it is not pruned out with the prune function, then it will be added to the family and a new representative will be created. In essence the representative has become the signature for each family. If the analyst needs to see how close families are to each other, all that is needed is to compare their representatives and see how similar they are indicating that the families may be related to each other.

Some of the problems associated with this method are that if n-grams are used as features there will be a large amount of possible combinations. For example, if bigrams are chosen as features, then there will be $\binom{300}{2}$ bigrams, and since order matters in this instance that makes the possible combinations at $[\binom{300}{2} * 2]$ bigrams. The possible combination of bigram ratios becomes $\binom{[\binom{300}{2}*2]}{2}$. This number is extremely large, but in reality the bigram sequence combination should shrink considerably due to the fact that some instructions are never found in specific sequences because of the way the instruction architecture set was designed. Another point is that most families use no more than 100 instructions, which is a considerable reduction from 300. Although these two

points greatly reduce the combinations of possible bigrams, it is not enough to make this method practical enough for my purpose.

4.2.2 Exact Feature Counts

This classifier is the one actually implemented in the automated system and was used in the experiment phase. The idea behind this classifier is that it is very simple and direct. The way the classifier works is that after the clustering phase has been completed each family is grouped together and all the features that have the same count number are kept as the signature. The family ends up being represented with a set of exact feature counts that all members of the family share. For example Table 5 and Table 6 shows an example of a feature set for a family. Table 5 starts first with the API function calls and their counts, and then Table 6 lists the bigram instructions and their counts.

Table 5: API Feature Counts

API Function Name	API Count
GetTickCount	3
MultiByteToWideChar	1
FindFirstFileA	1
CharLowerA	2
FindNextFileA	1
FindClose	1
ExitThread	3
CreateFileMappingA	2
RegEnumValueA	1
GetFileSize	2
SetEndOfFile	1
WritePrivateProfileStringA	3
GetModuleFileNameA	1
MapViewOfFile	2
RtlUnwind	1

OpenFileMappingA	1
ReadFile	1
WaitForSingleObject	1
SetTimer	1
GetFileTime	1
SetErrorMode	1
GetUserNameA	1
SetFileAttributesA	3
SetFileTime	1
GetPrivateProfileIntA	2
lstrcpynA	1
KillTimer	1
GetComputerNameA	1
UnmapViewOfFile	2
GetDriveTypeA	1

Table 6: Bigram Feature Counts

Bigram Name	Bigram Count
movsx mov	1
pop inc	1
sub neg	1
and sub	2
lea cmp	2
or jz	1
pop add	1
add and	2
inc mov	1
cmp jle	3
cmp jge	4
setz add	1
push cld	1
push inc	1
div test	1
xor jmp	9
add sub	1
cld mov	1
add pop	3
sub push	6
add imul	3
inc xor	2

sbb inc	1
lea call	1
mov jz	1
movsx sub	1
movsx xor	1
xor push	1

As stated previously, I chose to keep the feature set limited to only two types of information, the API function calls and the instruction bigrams. This does not mean that this system cannot take on more features. It can also take on totally different features from the ones that have been discussed.

4.2.3 Blockmodel Comparison

A third and potentially useful classifier is the blockmodel comparison classifier. After having the malware clustered into families, I take each family and represent them as a two-mode social network in matrix form. To reduce the matrix into a more compact form, I remove all the features that are not common between all of the members of the family. Therefore, each member must have at least 1 count of the feature or else that feature is removed. This step not only makes the matrix more compact but also increases the accuracy of the signature. Then for each remaining feature I take the minimum and maximum value for it from the family members and create a range {Min:Max} for that feature. I continue doing this for all the features until I have a vector of ranges, one for each feature of that family. This vector becomes the family's signature. An example is shown in Table 7:

Table 7: Initial Malware Family

	feature1	feature2	feature3	feature4	feature5	feature6
mawalre 1	9	1	0	18	12	5
malware 2	4	3	2	20	9	1
malware 3	7	1	0	7	2	10
malware 4	3	2	15	7	3	1
malware 5	4	5	2	10	13	0

Then the matrix reduced to only common features, resulting in Table 8:

Table 8: Reduced Blockmodel Matrix

	feature1	feature2	feature4	feature5
mawalre 1	9	1	18	12
malware 2	4	3	20	9
malware 3	7	1	7	2
malware 4	3	2	7	3
malware 5	4	5	10	13

Then the vector is created which will be the final signature for the family:

Family signature = [feature 1{3:9}, feature 2{1:5}, feature 4{7:20}, feature 5{2:13}]

If an unknown executable is fed into the classifier, its features will be converted into a single vector. This vector will be compared to the vectors in the database and if the vector falls within the ranges of a family's vector then it will be considered a match. In the case that a vector has more or fewer features than the families in the database, the

family that has the greatest number of common features with the vector as well as having the unknown vector fall within the range of its common features, will be considered the closest family. In the case that there is more than one family found as a match, then the family that has an average range nearest to the unknown vector will be chosen.

5 SCALABILITY AND SIMULATION

In this chapter I discuss the problem of scalability since most malware analysis will be conducted with thousands of malware at a time; therefore, I must test how well the system is able to handle a larger number of malware. There are two different experiments that will be carried out in this dissertation, the first being a small scale analysis of a set of malware, and the second a substantially larger set of malware. I will be discussing the procedure and implementation of the second and larger experiment.

5.1 Simulation

When first starting this dissertation, I obtained a large portion of my malware samples from the VX Heavens website, which was an extremely helpful website that provided a vast variety of different malware, helpful articles, and tools for educational purposes. Unfortunately, as I was conducting my research the site was taken down due to legal problems in the country where the server resided. This left me in a tough position of trying to obtain new malware samples for the second experiment, which would require thousands of samples. There is almost no other site that offers labeled malware as the VX Heavens website. Most of the anti-virus companies are not very willing to give up the malware samples that they have collected because they are in competition with other commercial companies and because distributing malware samples might cause legal

problems for them. This dilemma created a problem for my research, and so I had to think of a way to circumvent not having a lot of malware samples by simulating malware family members.

To simulate the malware family members, I first brought a set of 44 malware samples that were from relatively different families as labeled by Kaspersky. Each of these 44 malware will now represent a separate family. Taking each sample, I created 99 other simulated samples per family, based on the initial malware sample. This comes out to be 4,400 samples created from the initial 44 samples. The 44 samples are listed in Table 9:

Table 9: The 44 Malware Samples

Initial 44 Malware
Agent.ad, Agent.cb, Agent.cg, Agent.z, Backdoor.Win32.Yoddos.an, BackdoorWin32Jukbot.B, Delf.aa, Delf.ai, Delf.as, Delf.ce, Delf.j, Henky.Rotten.1408, HEURTrojan.Win32.Generic, HEURTrojan.Win32.Generic10, HEURTrojan.Win32.Generic13, HEURTrojan.Win32.Generic16, HEURTrojan.Win32.Generic18, HEURTrojan.Win32.Generic24, HEURTrojan.Win32.Generic3, HEURTrojan.Win32.Generic4, HEURTrojan.Win32.Generic6, Lamer.a Mkar.d, not-avirusAdWare.Win32.Zwangi.jgn, RainSong.4198, Sality.n, Sality.p, Selfish.c, Slaman.a, Small.r, Small.u, Texel.a, TrojanDropper.Win32.Agent.ayfc, Trojan-GameThief.Win32.Magania.fyax, Trojan.Generic.7067332, Trojan.Win32.Agent.rvpr, Trojan.Win32.Cosmu.awlb, Trojan.Win32.Diple.ddsg, Trojan.Win32.Tgk.afx, Uknown9, unknown12, Viking, Worm.Win32.AutoRun.dgxu, WrongInf-A [Susp]

The simulated samples are created by setting the mutation rate to .05, which is 5% of the original malware sample. Previous research in simulation also set the mutation rate at 5% as in [59]. This means that for each of the 99 derived malware samples, they will have feature count vectors that differ randomly by 5% from the original malware sample. This is achieved by going through the original malware vector of features and randomly choosing 5% of the features and changing the feature counts to a randomly selected number that is within the range of possible values. The 5% might not seem significant at first thought, but considering that 5% of the bigram feature counts will be randomly changed and 5% of the API feature counts will also be randomly changed, it quickly creates a distinct variant of the original base malware. The creation of the 99 family members is not to make them substantially different, but to make them different enough to be distinct as a different malware family member. Also, if the count is changed to a value of 0, it means that the feature does not occur in the newly created sample, which is sufficient to represent deletions of features.

5.2 Cloud Computing

Since the malware samples are in the thousands and may exceed that in future analyses, I had to find a way to decrease the computation time. The best practical way was to make use of cloud computing services offered online. The first solution was from a company called Cloudnumbers.com that provides a high performance computing platform tailored for the scientific and research community that need to perform processor intensive calculations on various mathematical and statistical software like R,

Python, Fortran, and Matlab. The advantage they had was that the system did not need any configuration. It was created virtually without the need for the user to have any interaction with the operating system; the user simply used the software as though it was an online application. Unfortunately for me Cloudnumbers's services are no longer offered and it appears to have gone out of business. The next most convenient method was using Amazon's Elastic Compute Cloud (Amazon EC2) [60]. This was the solution adopted for the purpose of this research. Amazon EC2 is a web service that gives users the ability to configure their own virtual systems and obtain a high level of computational speed. Through the use of combining many different computer clusters, which they call Compute Units, one can substantially speed up computation. Each Compute Unit has the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor [60]. The interface to create the virtual systems was very straight forward and relatively simple to use. The system I created was a High-CPU Extra Large Instance, which has 7 GiB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of local instance storage, and a 64-bit platform. The pricing was at $0.660 per hour, which for the computing power I was receiving was very fairly priced.

To utilize the 20 clusters provided by Amazon's EC2 web service, I had to make my algorithms parallel compatible, and, therefore, re-wrote portions of them. The statistical software R has a package named multicore [61] that contains a collection of various functions, which gives a way to run parallel computations with the computing clusters on the current system. Specifically, I used the mclapply() function out of the multicore package, and fitted my existing algorithms with it. For example using the

mclapply() function with my Canberra similarity measurement I would invoke the following command:

output <- mclapply(pair, canberra, matrix_api)

Where “pair” is the list of all possible combinations of malware pairs that will be compared and “canberra” is the similarity function that will provide a similarity score for each pair. And finally, “matrix_api” is the two-mode matrix of malware and their API function call counts.

5.3 Performance Analysis

In this section I will discuss the decrease in computational time through parallelizing the most time consuming function which in this case is the Canberra similarity measurement, and I will list the performance of each algorithm used in my malware analysis system.

The Canberra similarity measurement has a complexity of O($N^2 * k$), where k is the number of columns in an *n x m* matrix, and can be optimized to O($\frac{N^2 * k}{2}$). I had a difficult time running the Canberra similarity measurement on my laptop (which uses a 2.0 GHz Intel Core processor) when the n-gram features began increasing in number. For the 2-grams I had 1569 features in the column of the *n x m* matrix, but as I reached 5-grams, it was around 20020 and it became very tedious to execute. I, therefore, decided to run the Canberra similarity measurement in parallel by rewriting the code. By using the 20 clusters provided by the Amazon EC2 platform, the computational time was decreased

by 6.1 times as compared to the laptop's computational time. The main advantage of having the code parallel compatible is that increasing the number of clusters will also decrease the computational time without any needed addition to the code, and increasing the number of clusters is very convenient on the Amazon EC2 platform. Therefore, the computational time of the malware analysis system can be decreased by increasing the number of clusters without any additional coding.

Table 10 shows each algorithm that has been created for my malware analysis system and its corresponding complexity. The algorithms have not been fully optimized in this work and optimizing the algorithms is listed in the future work section.

Table 10: Algorithm Complexity

Algorithm	Complexity
FamilyPruner()	O(N^2)
Prune()	O(N)
Subfamily()	O(N)
Simplify()	O(N)
ImageMatrix()	O(N^2)

6 EXPERIMENTS AND PERFORMANCE EVALUATION

The discussion in this chapter will be about two different experiments carried out separately, with each experiment being independent of the other. The first experiment tests the how well the system does on a small scale and generally makes sure that the system functions correctly. The second experiment tests how well the system functions at a much larger scale. The final results for each will be presented along with applying the performance evaluation equations to see how well the system functioned in both experiments.

6.1 Small Scale Analysis of Malware

As discussed in section 2.3 Collecting Malware, after running a couple hundred of the collected malware through the system in Figure 3, I was left with 77 malware samples that successfully passed through all the filters and met the criteria. The 77 malware were all that I could successfully obtain from the VX Heavens website, and are enough to establish the effectiveness of the malware analysis system prototype that I developed by implementing the methodologies presented in this dissertation. Establishing the effectiveness of my system includes the ability to create clearly defined malware families and creating signatures that are capable of detecting zero-day malware for each family. The malware were already named by the VX Heavens website, and they used Kaspersky

antivirus to label their malware set. I used that naming mechanism to get an initial assessment of how many distinct malware and distinct families I had to experiment with. There were 77 unique malware and 14 families in total (as recognized by Kaspersky). Table 11 lists the 14 families.

Table 11: The 14 Malware Families

Malware Family	Number of members
Agent	9
Chiton	1
Delf	29
Downloader	1
Henky.Rotten	1
Lamer	1
Mkar	3
RainSong	1
Resur	2
Sality	4
Selfish	3
Slaman	9
Small	6
Texel	7

After gathering the malware members of all the previously listed families, I created a two-mode social network consisting of the malware members and all of their associated API calls. I also created another two-mode social network consisting of the malware members and all of their associated bigram instructions. This took on the form of a matrix with the elements representing counts of both features the API calls and the bigram instructions. For the API feature matrix it had the dimensions of [77 x 1241], and the bigram instructions had dimensions of [77 x 1569]. Then both matrices were reduced

to a one-mode social network consisting of malware as the one set of actors. This reduction was implemented by applying the Canberra similarity measurement on the two-mode network and outputting the resulting similarities between the malware. Now there are two one-mode networks with dimensions of [77x77], one for the API calls features and one for the bigram instructions. Next is to apply the clustering algorithm to obtain the clusters of families. But before that step is reached, a single one-mode matrix must be given to the clustering algorithm. I took the weighted average of both one-mode matrices and created a single matrix representing the similarities of the malware members. In this experiment an added weight of 5% was given to the API call features, giving a slightly more accurate representation in the clustering phase. Therefore, the two features of API calls and bigram instructions have a more accurate influence on the final one-mode matrix than a non-weighted average. This shows a stronger representation of the true similarities between the malware members.

6.1.1 N-gram Selection

Before discussing the clustering portion, I'd like to discuss the sets of n-gram instructions that were collected from the malware. In this dissertation, I chose the bigram (2-grams) instructions to represent the malware even though I had 3-grams to 10-grams of instructions collected and created. The reason is that during the clustering phase the n-grams that were substantially larger than the bigrams had a very poor clustering result that grouped them into very large cluster sizes. I achieved the most accurate clusters using the small n-grams from 1 to 3. What is meant by accurate clusters is that they

somewhat resemble Kaspersky's family clustering as well as not having crowded the malware all in a small amount of clusters. I also obtained a sufficient amount of signatures from the bigrams as opposed to the other n-grams. The 1-grams are of single instructions that had too much overlap between the families and, therefore, could not be used as signatures. The 3-grams had too little shared between malware family members and were not enough to establish a criterion of at least 20 common features let alone have enough exact feature counts. As Table 12 shows the counts and sparsity for each n-gram, it is evident that that the number of n-gram counts rises quite dramatically starting from the single instructions of 1-grams till it peaks at the 8-grams, and showing that as 'N' rises so does the sparsity.

Table 12: N-grams Counts and Sparsity

N-gram	N-gram Counts	Sparsity
1-gram	168	61.14719%
2-gram	1569	77.40475%
3-gram	6158	86.56821%
4-gram	13604	91.36417%
5-gram	20020	93.32518%
6-gram	23819	94.20876%
7-gram	25634	94.76081%
8-gram	25829	95.18461%
9-gram	24795	95.51093%
10-gram	22997	95.78223%

6.1.2 Clustering

Next the averaged matrix is fed into the clustering algorithm DBSCAN. As described in section 3.3.1, it accepts three parameters including the similarity matrix. The first parameter MinPts will be set to the value of 2, because a family must have at least two members to be considered a family. The Eps parameter is computed by taking the mean of the k^{th} nearest neighbor of all the points in the dataset, and that value turned out to be 73.01168. It is also important to note that the matrix was not normalized, thereby explaining the large number for Eps. The resulting output from DBSCAN is shown in Table 13.

Table 13: DBSCAN Clustering Results

Family Name	Family Members
Family1	Selfish.c, Selfish.d, Selfish.e
Family2	Sality.n, Sality.t
Family3	Mkar.d, Mkar.f
Family4	Delf.j, Delf.q
Family5	Delf.ce, Delf.r
Family6	Delf.as, Delf.cf, Delf.cx, Delf.v
Family7	Delf.ai, Delf.aj, Delf.am, Delf.br, Delf.ca, Delf.t, Delf.u, Downloader.g
Family8	Delf.aa, Delf.af, Delf.l, Small.m

Family9	Chiton.t, Sality.ab, Slaman.a, Slaman.b, Slaman.c, Slaman.d, Slaman.f, Slaman.g, Slaman.h, Slaman.i, Slaman.j, Texel.a, Texel.c, Texel.f, Texel.g, Texel.h, Texel.j, Texel.l
Family10	Agent.cg, Agent.h, Delf.ap, Delf.bj, Delf.bs, Delf.g
Family11	Agent.at, Agent.bi, Resur.a

The table shows that from the 77 malware composing 14 families, the DBSCAN algorithm clustered only 54 malware into 11 families. This leaves out 23 malware that could not be clustered into any families, as shown in Table 14. One important note is that the 23 unclustered malware will not be ignored by the system; they will each represent an independent family after they are found to not be affiliated with any malware family as explained in test 1 of section 6.1.4. The clustering resulted in about 70% of the original malware clustered into some type of family. As can be seen from the results, a majority of the malware samples seem to be grouped with similarly named malware, which is a good early indicator that my results generally agree with Kaspersky's classification method. The resulting comparison is not necessarily an accurate assessment because the malware samples were named by Kaspersky antivirus, which is by all means not a perfect antivirus system.

Table 14: Malware not Clustered by DBSCAN

Unclustered Malware

Agent.ad, Agent.bm, Agent.cb, Agent.t, Agent.z, Delf.62976, Delf.b, Delf.bm, Delf.bt, Delf.ch, Delf.dc, Delf.n, Henky.Rotten.1408, Lamer.a, Mkar.c, RainSong.4198, Resur.f, Sality.p, Small.i, Small.k, Small.r, Small.u, Small.y

Next the families were fed into the FamilyPruner function, which will prune the families into a tighter and accurate f it. The resulting output from the FamilyPruner function is shown in Table 15. As can be seen almost all of the families stayed the same, with only Family 9 having had two malware pruned from it Chiton.t and Sality.ab.

Table 15: The Resulting Families after Pruning

Family Name	Family Members
Family1	Selfish.c, Selfish.d, Selfish.e
Family2	Sality.n, Sality.t
Family3	Mkar.d, Mkar.f
Family4	Delf.j, Delf.q
Family5	Delf.ce, Delf.r
Family6	Delf.as, Delf.cf, Delf.cx, Delf.v
Family7	Delf.ai, Delf.aj, Delf.am, Delf.br, Delf.ca, Delf.t, Delf.u, Downloader.g
Family8	Delf.aa, Delf.af, Delf.l, Small.m

Family9	Slaman.a, Slaman.b, Slaman.c, Slaman.d, Slaman.f, Slaman.g, Slaman.h, Slaman.i, Slaman.j, Texel.a, Texel.c, Texel.f, Texel.g, Texel.h, Texel.j, Texel.l
Family10	Agent.cg, Agent.h, Delf.ap, Delf.bj, Delf.bs, Delf.g
Family11	Agent.at, Agent.bi, Resur.a

To show why Chiton.t and Sality.ab were removed, I'll go through some the steps in the FamilyPruner function. After creating a matrix that encompasses Family 9, an average vector is computed then a cosine similarity matrix is created. The SimilarityVector is taken from the resulting cosine similarity matrix, which shows the similarity between the malware and their average vector. Table 16 shows the SimilarityVector for Family 9:

Table 16: Similarity between Malware Members and Average Vector for Family

SimilarityVector of Family 9					
Chiton.t	Sality.ab	Slaman.a	Slaman.b	Slaman.c	Slaman.d
0.39183113	0.07104826	0.89500671	0.92112075	0.89709911	0.92259195
Slaman.f	Slaman.g	Slaman.h	Slaman.i	Slaman.j	Texel.a
0.92677663	0.91690507	0.88581112	0.89706797	0.86114043	0.85730602
Texel.c	Texel.f	Texel.g	Texel.h	Texel.j	Texel.l
0.85715554	0.89457066	0.89326477	0.89385601	0.85715554	0.87928455

The two malware along with their similarity scores have been enclosed in a red circle, indicating that these scores are very low compared to the rest. Next the SimilarityVector for Family 9 is sent to the Prune function, which chooses the largest similarity score, Slaman.f in this case, and subtracts each score from it to create a new (Maximum – point) vector as shown in Table 17. The average of the new vector is computed at 0.115470 and that is multiplied by 1.5 to obtain 0.173205, which will be the threshold used. Then all the elements in the vector are compared to the threshold and anything that is higher than that value is marked for deletion. Again it can be seen that Chiton.t and Sality.ab have a red circle encompassing them because they are clearly over the threshold. After the algorithm is completed and the two malware are pruned, Family 9 is left with two groups of malware the Slaman and the Texel groups.

Table 17: Application of the Pruning Function on the SimilarityVector

(Maximum – Point) Vector of Family 9					
Chiton.t	Sality.ab	Slaman.a	Slaman.b	Slaman.c	Slaman.d
0.5349455	0.8557284	0.03176992	0.005655881	0.02967752	0.004184675
Slaman.f	Slaman.g	Slaman.h	Slaman.i	Slaman.j	Texel.a
0	0.009871556	0.04096551	0.02970865	0.0656362	0.06947061
Texel.c	Texel.f	Texel.g	Texel.h	Texel.j	Texel.l
0.06962109	0.03220597	0.03351186	0.03292062	0.06962109	0.04749208

The next step is to try and create subfamilies from the pruned family through the Subfamily function. In the case of Family 9, after running it through the function I obtained two subfamilies as shown in Table 18.

Table 18: Family 9 is Split into two Families

Family 9	
Subfamily 9A	Slaman.a Slaman.b Slaman.c Slaman.d Slaman.f Slaman.g Slaman.h Slaman.i Slaman.j
Subfamily 9B	Texel.a Texel.c Texel.f Texel.g Texel.h Texel.j Texel.l

The final set of families are arranged to create a blockmodel, which will visually show relationship strengths between the malware families. The blockmodel is shown in Figure 15. The names of the families are hard to see in the image and they are arranged by the following list of families: Family 9A, Family 2, Family 3, Family 4, Family 5, Family 6, Family 7, Family 8, Family 9B, Family 1, Family 10, and Family 11. From the figure one can clearly see the blocks of blue representing each family. For almost all the blocks there is little if no relation to other families, but Family 9A and Family 9B clearly have a blue block shared between them. This relationship is encircled in red in the figure, and because the blockmodel is symmetric there are two red circles. This relationship can be explained by the fact that they are both a part of a larger family that was split into two separate subfamilies. When looking at the blockmodel, one can see a shift in the size of the blue blocks after Family 11 and starting with Chiton.t and the reason is that after Family 11 all of the unclustered malware were listed and therefore no large blocks were formed. Although there are blotches of blue in various areas in the blockmodel, it is still

not enough to create any similarity between them; it just simply means that one member of a family may be similar to a member of another family but not as similar to the other members and not qualifying to be clustered with that family. But if there is a consecutive blue streak that goes through all members of the family it might indicate that the malware was not properly clustered with that family. An example is Delf.b and Delf.bt that have clear horizontal and vertical blue lines across Family 10. I will go into more detail when I discuss experiment 1.

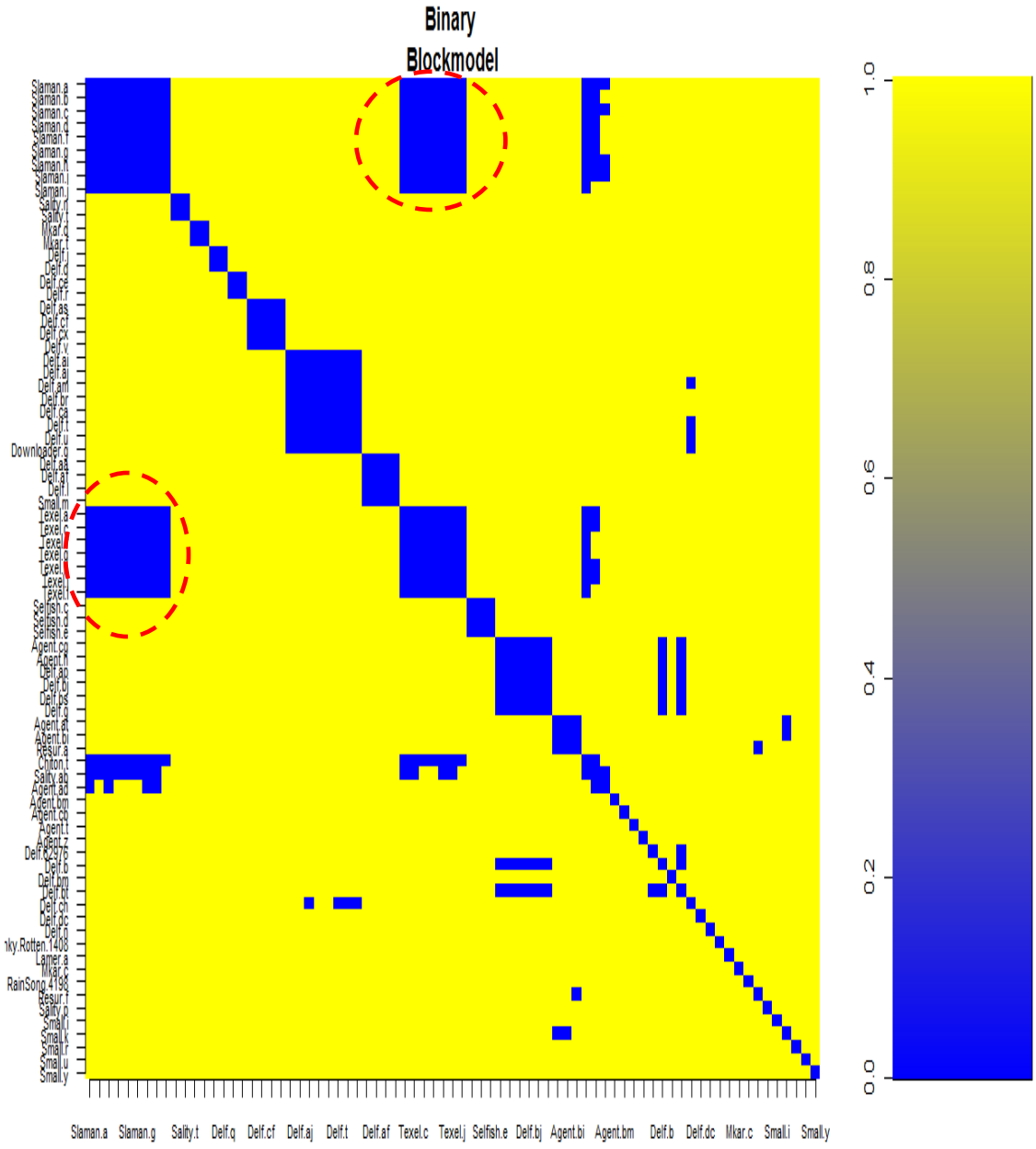

Figure 15: Clarified Blockmodel

The image matrix is then created from the blockmodel, showing the relationships of the families in a compressed form. This image matrix is show in Table 19. The other unclustered malware are not put in the image matrix.

Table 19: Image Matrix of Family Relationships

	Family1	Family2	Family3	Family4	Family5	Family6	Family7	Family8	Family9A	Family9B	Family10	Family11
Family1	0	0	0	0	0	0	0	0	0	0	0	0
Family2	0	0	0	0	0	0	0	0	0	0	0	0
Family3	0	0	0	0	0	0	0	0	0	0	0	0
Family4	0	0	0	0	0	0	0	0	0	0	0	0
Family5	0	0	0	0	0	0	0	0	0	0	0	0
Family6	0	0	0	0	0	0	0	0	0	0	0	0
Family7	0	0	0	0	0	0	0	0	0	0	0	0
Family8	0	0	0	0	0	0	0	0	0	0	0	0
Family9A	0	0	0	0	0	0	0	0	0	**1**	0	0
Family9B	0	0	0	0	0	0	0	0	**1**	0	0	0
Family10	0	0	0	0	0	0	0	0	0	0	0	0
Family11	0	0	0	0	0	0	0	0	0	0	0	0

From the image matrix, an equivalent graph of it is created to better visualize the analysis of the malware families as shown in Figure 16. A line is drawn to show that a relationship between family 9A and family 9B exists.

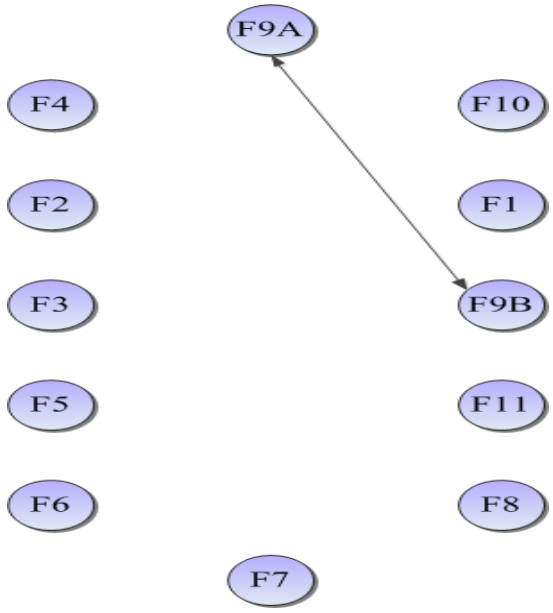

Figure 16: Graph of Image Matrix

6.1.3 Signature Creation

Next I created the signatures for each family, which will constitute two sets of features the API function call counts and the bigram counts. Table 20 shows the number of features that were extracted out of each family like the common features and the exact feature counts. Only the exact feature counts were used as the signatures during classification. From the table, one can see that the common features occur much more than the exact counts because to have an API or Bigram feature listed as common all members must have at least 1 count of the feature, which is less stringent than having an exact count of the feature. The Bigram features also have many more counts than the API calls because the executable itself is composed of single instructions, and they outnumber

the API calls many times over. The signature of each family is composed of a set of both exact API function calls and exact bigrams.

Table 20: Exact and Common Counts of Features for each Family

Family Name	API Calls		Bigrams	
	Exact Count	Common	Exact Count	Common
Family1	414	431	843	951
Family2	31	60	28	102
Family3	96	98	406	493
Family4	62	77	293	442
Family5	75	90	458	574
Family6	228	321	418	672
Family7	50	78	309	503
Family8	36	49	166	347
Family9A	18	64	13	58
Family9B	17	46	8	48
Family10	11	24	55	185
Family11	29	44	140	330

6.1.4 Performance Evaluation

Having finished the signatures for the families, I tested the accuracy of the signatures by performing two tests that compare how well the signatures can classify each malware from the initial 77 malware samples into their correct family. The first test uses only the 25 unclustered malware. The second test uses the 52 clustered malware. A third and final test will show how well the signatures handle 92 samples of non-malware executables called goodware. To ensure that the goodware are 100% clean I collected as many IA-32 executables as I could find from a newly installed and, therefore, clean version of Windows XP. This is common practice when trying to obtain goodware from a trusted source as done in [40], [42].

The rationale behind this first test is to see if any of the 25 unclustered malware may fit into any of the families. The 25 unclustered malware can be considered zero-day malware because the creation of the signatures didn't include them and hence are completely unknown to the system. A sample of the results is displayed as in Figure 17.

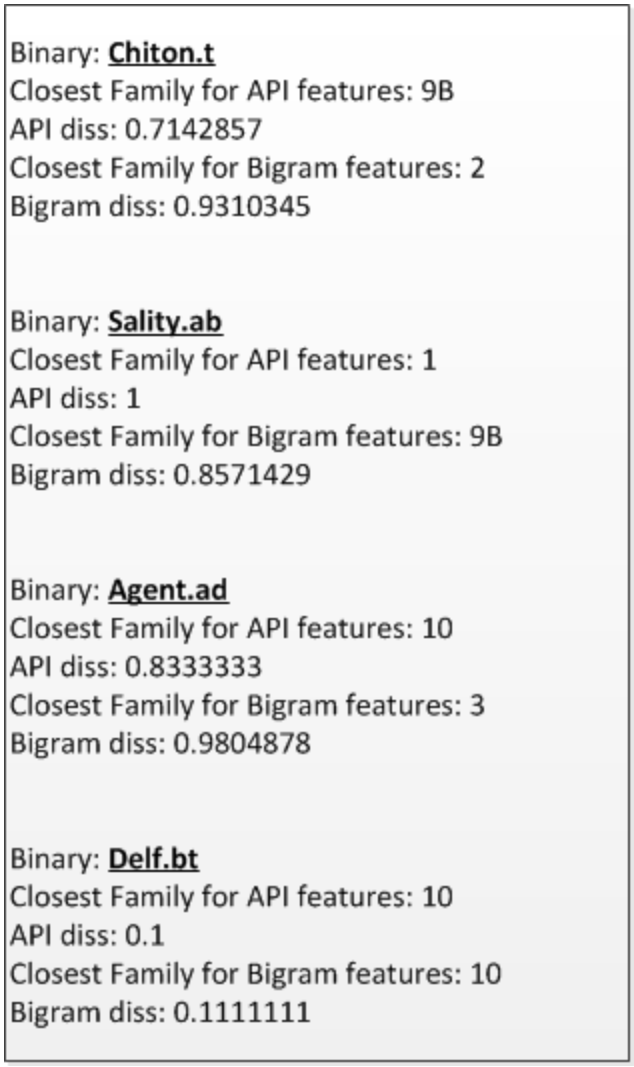

Figure 17: Partial Output of the 25 Unclustered Malware Classification

Each malware sample has the closest family in terms of API features and in terms of Bigram features outputted along with their dissimilarity score. The closer the score is to zero, the closer it is to an exact match of the family. From the figure it can be seen that Chiton.t, Sality.a, and Agent.ad have high dissimilarity scores for their closest family. Another observation is that the closest family for the API features is different than the closest family for the bigram features. In contrast, Delf.bt has very low dissimilarity

scores for both the API and bigram closest family, as well as having the same family be a match for both features. This is a clear indication that Delf.bt is a part of family 10. There are 3 criteria that a malware sample must fulfill to be classified in a particular family:

1. The API feature dissimilarity score to the closest family must be smaller than 0.2.
2. The bigram feature dissimilarity score to the closest family must be smaller than 0.2.
3. The family closest to the API features must match the family closest to the bigram features.

There were three malware, Delf.62976, Delf.b, and Delf.bt, that met all of the 3 criteria listed previously, and all were classified under family 10. An interesting note is that family 10's members are 67% "Delf" as labeled by Kaspersky anti-virus. This does show a similar classification between Kaspersky anti-virus and the classification of this system. Out of the 25 unclustered malware, only 3 were found to be highly associated with a family. The reason why these malware were not clustered originally with the family in the clustering phase is because clustering algorithms are estimation based, and, therefore, are prone to errors. These newly found family members have been classified as part of the family, but deciding to include them in the family and re-generate the signature is another issue. These newly classified malware have to survive the pruning procedure to make sure that they are a good fit. But in either case they will be classified as a part of that family. An interesting observation is that the three malware's association

with Family 10 could clearly be seen in the blockmodel, which shows a solid vertical and horizontal blue line encompassing Family 10 for both Delf.b and Delf.bt.

A problem that might arise from when new binaries are classified into a malware family and survive the pruning stage is that as the malware family grows its number of exact features gets reduced until eventually it becomes very small and will not be able to correctly function as a signature for such a large family. This problem is treated by checking the number of exact features after each binary is added to a malware family and seeing if it goes under a critical threshold point, in which the Subfamily() function will create independent subfamilies with each family having exact features over the threshold.

The second test will show how well the signatures recognize the 52 clustered malware. Even though one might first think this to be a redundant experiment because the signatures were extracted out of the 52 malware, it is important to note that there may be a lot of overlay of exact API and bigram counts between those signatures, which might cause some ambiguity in the classification. The results were a 100% correct classification of all 52 malware in their correct families.

The third test shows how well the signatures do against 92 goodware. The results were very positive with a 100% of the goodware not classified into any of the malware families. And the majority of the dissimilarity scores were very high indicating that there is a clear distinction between malware and goodware. One of the goodware files named "cplexe.exe" had a 0.2340426 for the API features and a 0.4659686 for the bigram features, which were the closest dissimilarity scores of all the goodware but were still too high to be a part of any malware family. When I looked into what the cplexe.exe file's

purpose in the Windows XP operating system, it turned out to be an important part of the operating system that executes dll files and saves the file's libraries into memory [62]. As it also turns out some malware can Impersonate the cplexe.exe file by injecting some malicious lines of code to it and thus replacing the original file with a corrupted one [62]. This may explain why it was closer to being classified into a family than the other goodware. That malware family could have had the cplexe.exe file embedded within their own malware file to replace the victim's original one. Another theory could be that because cplexe.exe manipulates dll files and accesses memory, it, therefore, uses similar API function calls and bigrams as the malware family which displays similar type of behavior but for malicious purposes.

For tests 2 and 3, I performed the performance measurements discussed in section 2.6 to evaluate my proposed techniques. Test 1 is excluded from the evaluation because there is no definite classification label of the malware being tested, because it was originally labeled and assigned a family by Kaspersky anti-virus, which is not 100% accurate in contrast to the malware used in test 2 that were assigned a family by the system discussed in this dissertation. I calculated the following values:

True Positive (TP): 52

True Negative (TN): 92

False Positive (FP): 0

False Negative (FN): 0

Therefore,

The True Positive Rate is (TPR): $\frac{52}{52+0} = 1$

The False positive Rate (FPR): $\frac{0}{92+0} = 0$

Accuracy: $\frac{52+92}{52+92+0+0} = 1$

From the results, one can see that the system has correctly classified all the malware in their correct families with no false positives, and has correctly not classified any of the goodware into the malware families.

6.2 Larger Scale Analysis of Malware

In this experiment, I followed the same procedure as the small scale experiment only with 44 families with each family having 100 malware members. This brings the total to 4400 malware. I took as many malware samples from the previous 77 malware that were used in the small scale experiment, but I made sure that each malware was from a different family to ensure that samples were distinctly different. The reason is that in this experiment each malware will have 99 other malware created from it to simulate a malware family and so the starting 44 malware samples must be distinct. As to why I chose to do only 44 malware is because I wanted to make sure that I simulated at least 4,000 samples of malware, which is considered to be a medium to large scale analysis.

The experiment began by randomly choosing 20 malware out of each family. These 20 malware represented the training data, bringing the total malware to 880 malware. The set of 880 malware was then put into a two-mode social network consisting

of the 880 malware members and all of their associated API calls. I also created another two-mode social network consisting of the malware members and all of their associated bigram instructions. This took on the form of a matrix with the elements representing counts of both features the API calls and the bigram instructions. For the API feature matrix, it had the dimensions of [880 x 1453], and the bigram instructions had dimensions of [880 x 2328]. Then both matrices were reduced to a one-mode social network consisting of malware as the one set of actors. This reduction was implemented by applying the Canberra similarity measurement on the two-mode network and outputting the resulting similarities between the malware. Now there were two one-mode networks with dimensions of [880x880], one for the API calls features and one for the bigram instructions. Next I applied the clustering algorithm to obtain the clusters of families. To feed the clustering algorithm a single one-mode matrix, I took the weighted average of both one-mode matrices and created a single matrix representing the similarities of the malware members. In this experiment an added weight of 5% was given to the API call features, giving a slightly more accurate representation in the clustering phase. Therefore, the two features of API calls and bigram instructions have a more accurate influence on the final one-mode matrix than a non-weighted average, which shows a stronger representation of the true similarities between the malware members.

6.2.1 Clustering

Next the averaged one-mode social network matrix was fed into the clustering algorithm DBSCAN. As described in section 3.3.1 it accepts three parameters including the similarity matrix. The first parameter MinPts was set to the value of 2, because a family must have at least two members to be considered a family. The Eps parameter was computed by taking the mean of the k^{th} nearest neighbor of all the points in the dataset, and that value turned out to be 6.991955. It is also important to note that the matrix was not normalized hence explaining the large number for Eps. The number of families outputted from DBSCAN is shown in Table 21. For the purpose of keeping the table small, only the names of the families are listed.

Table 21: Families Outputted by DBSCAN

31 Created Families
WrongInf-A [Susp], Viking, unknown12, Uknown9, Trojan.Win32.Tgk.afx, Trojan.Win32.Diple.ddsg, Trojan.Win32.Cosmu.awlb, Trojan.Generic.7067332, Trojan-GameThief.Win32.Magania.fyax, Trojan-Dropper.Win32.Agent.ayfc, Texel.a, Small.u, Slaman.a, Sality.n, RainSong.4198, not-a-virusAdWare.Win32.Zwangi.jgn, Lamer.a, HEURTrojan.Win32.Generic6, HEURTrojan.Win32.Generic3, HEURTrojan.Win32.Generic24, HEURTrojan.Win32.Generic16, HEURTrojan.Win32.Generic13, HEURTrojan.Win32.Generic10, HEURTrojan.Win32.Generic, Henky.Rotten.1408, Delf.j, BackdoorWin32Jukbot.B, Backdoor.Win32.Yoddos.an, Agent.cg, Agent.cb, Agent.ad

There were 31 families created from the original 44 families, giving a 70% correct clustering of the families. There were 3 families that were not fully clustered correctly. The “Uknown9” family only had 6 out of 20 from its members clustered, the “Delf.j” family only had 18 out of 20 from its members clustered, and the “Agent.cb” family only had 17 out of 20 from its members clustered.

The families were then pruned, but the pruning did not produce any results and the families were kept the same. The same was true after feeding the families to the subfamily procedure, where no subfamilies were found.

The final set of families were arranged to create a blockmodel, which will visually show relationship strengths between the malware families. The blockmodel is shown in Figure 18. Since the malware set is 880, it would be impossible to list the names on the axes; therefore. they have been ordered by the previous list of 31 families followed by the unclustered malware. From the figure, one can clearly see the blocks of blue representing each family. For all the blocks there is little if no relation to other families except for some slight blotches of blue that may indicate some members of a family are related to some other members in another family. When looking at the blockmodel, one can see a shift after the 31st blue block, and the reason is that after Family 31 all of the unclustered malware were listed and, therefore, no solid blue blocks were formed. There is an interesting finding that the blue blocks stop after the 31st blue block, but after the 31st block there is a mist of blue, taking on a box shape and showed a clear pattern that might indicate there are further families that have not been clustered and were left out. This thought will be expanded in the classification section.

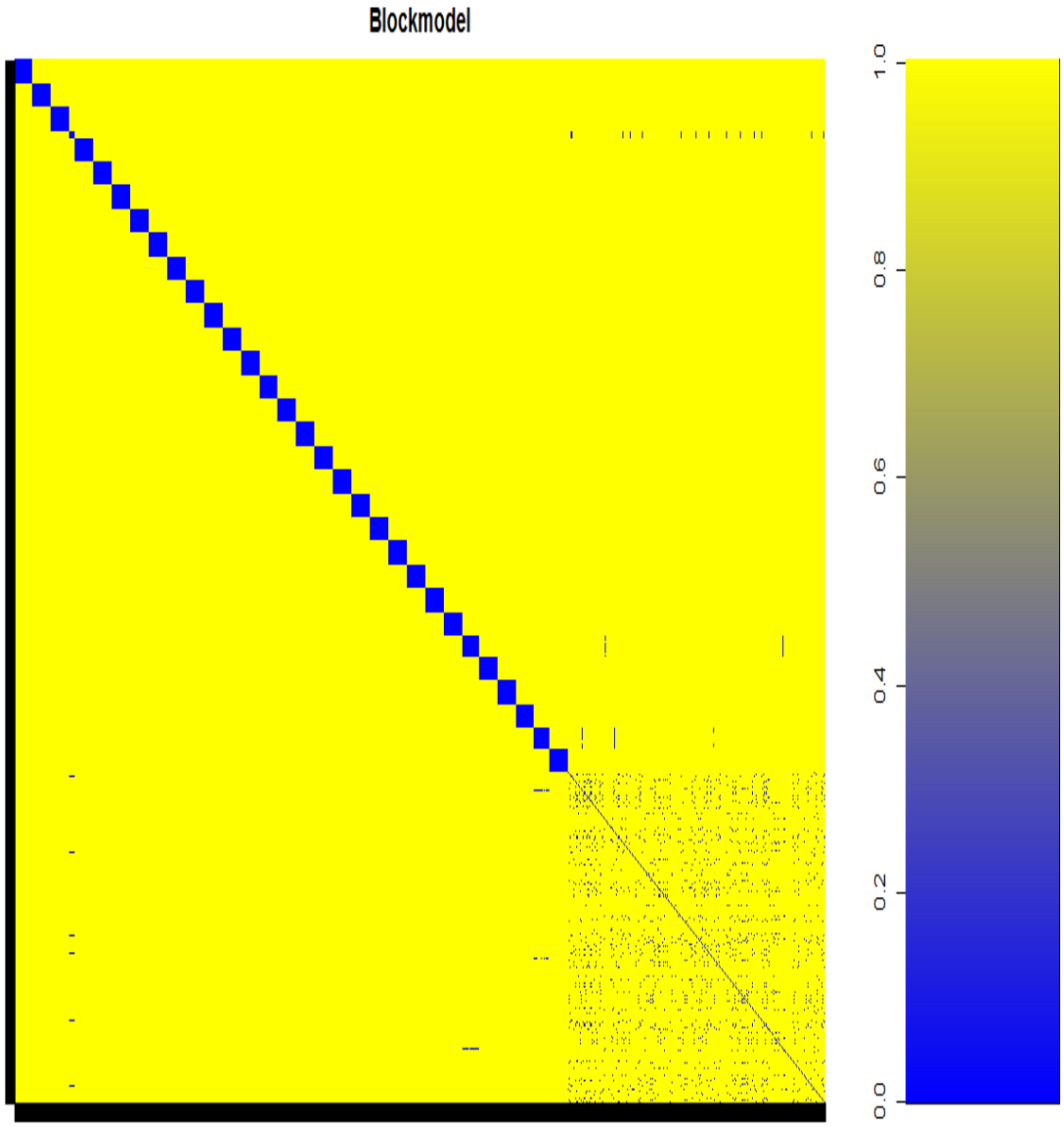

Figure 18: Blockmodel

The image matrix was then created from the blockmodel showing the relationships of the families in a compressed form, but in this instance the image matrix is

all 0's because there are no relationships between the families. The corresponding graph of the image matrix will also have no relationships, so there is no need in displaying both of them. Next I created the signatures for each family, which will constitute two sets of features the API function call counts and the bigram counts.

6.2.2 Performance Evaluation

After generating the signatures for the families, I tested the accuracy of the signatures by comparing how well they can classify the testing malware that was not used to generate the signatures. There were 31 families each generated with only 20% of the original 100 malware using around 620 malware. The testing malware were 80 per family, coming to a total of 2480 test malware. The 2480 test malware will represent the zero-day malware because they haven't been seen by the system. Using the same methods as in the previous smaller experiment I obtained a 100% correct classification for all the 31 families except for Family 18, which is the HEURTrojan.Win32.Generic6 family. For that family there was a 100% match on the bigram signatures but got a complete failure for the API signatures with a 0% correct classification. The reason behind the failure was due to the HEURTrojan.Win32.Generic6 family having no API functions, which was a very odd situation and was most likely due to it being intentionally hidden by the malware author. I would like to stress that the 100% correct classification was only on the 31 malware families that were created and hence had signatures created for them.

The second test shows how well the signatures do against 92 goodware. The results were very positive with a 100% of the goodware not classified into any of the malware families. And the majority of the dissimilarity scores were very high indicating that there is a clear distinction between malware and goodware.

Inspired that there is more to the mist-like blue block at the right corner of the blockmodel, I investigated the mist block further, which showed an indication that there were more families within the mist block that were not clustered by the DBSCAN algorithm. To that extent, from the 4400 malware I removed the 31 families and their members from the data set (since they were correctly clustered) and was left with 13 families comprising 1300 malware. I ran it through the previous methods and obtained 9 new malware families from the 13 families. I created signatures and tested how well they stood by testing them with the test data and got a 100% correct classification. I repeated the goodware test and got the same positive result. I also tested the signatures against the 3100 malware that were not included in this experiment and got a 100% correct classification that shows they do not fall in any of the 9 newly created families. One reason why the 9 newly found malware families were not caught in the previous large scale experiment might be because there were a large amount of malware families and the algorithm could not create a clear cut line between the clusters.

I performed the performance measurements discussed in section 2.6 to evaluate my proposed techniques:

True Positive (TP): 3900

True Negative (TN): 92

False Positive (FP): 0

False Negative (FN): 500

Therefore,

The True Positive Rate is (TPR): $\frac{3900}{3900+500} = 0.8863$

The False positive Rate (FPR): $\frac{0}{92+0} = 0$

Accuracy: $\frac{3900+92}{3900+92+0+500} = 0.8886$

From the results, one can see that the system has a good true positive rate that is around 88.6%, a false positive rate that is at 0%, and the accuracy is at 88.8%.

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

This dissertation tackles the problem of creating effective malware signatures through a fully automated malware analysis system free of any manual input that creates families of malware and creates a single, effective, and stable signature per family that is able to detect zero-day malware. It incorporates a visual component to the system by displaying a blockmodel of the malware families and how they are related to each other. It has the ability to adapt by deriving sub-families from their original families in which these sub-families become stand alone families themselves. The system was developed by fusing many different disciplines together, such as term/document-matrix from information retrieval, blockmodeling from social network analysis, and ratios from statistics. My research generally differs from previous related research in two main aspects:

1. The lack of TF-IDF weighting: TF-IDF is a computed weight that is applied to each of the terms. It increases with the number of times a term occurs in a document and with the rarity of the term in the whole document collection. Almost all of the previous research that is closest to my own research has used the TF-IDF weighting method on the pre-clustered malware binaries. This is an

erroneous step that should not be taken because the IDF measures the general weight of a term by including all the documents, and, therefore, it ends up relating all the malware binaries, which may be from different families, in the computed weight. Furthermore, even after clustering the malware into families, IDF is unsuitable for finding family signatures because the malware in each family will be very similar to each other sharing many terms. Therefore, because IDF adds weight to the rarity of a term, it accomplishes a completely different goal than what is intended when producing signatures.

2. Utilizing term frequencies during detection phase: Previous research most similar to this research only makes use of term frequencies during the clustering and family creation phase, but not during the detection phase. The problem is that these frequencies represent a rich amount of information that is being completely ignored instead of utilizing them to create a stronger signature.

The main contributions of my work are the following:

1. A fully automated system that can classify zero-day malware into its appropriate family or if that malware does not fit into any family then the closest family to it is used to give the analyst insight on possible similar behavior.
2. Previous research did not address the issue of pruning malware families to make sure that the clustering phase was accurate enough. Therefore, I developed a new pruning algorithm that tests cluster strength and ensures a tight malware family.

3. I created a novel application of blockmodeling to the problem of malware analysis whereby I incorporated a blockmodeling visualization component to the system that visually describes malware family relationships and possible hidden families that might have been missed by the clustering phase. I also created a novel algorithm that visually simplifies viewing the blockmodel.
4. I utilized the term frequencies during detection phase to create a stable malware family signature.
5. I created 3 different malware classification methods: the ratio of averages, exact feature counts, and the blockmodel comparison methods.

Through this research, I was able to recognize and steer away from mistakes that previous research had made, and created a different method to automate malware analysis. The first experiment was able to cluster 70% of the malware into families, and the rest became their own single member families. From those created families the overall classification accuracy was 100% with 0% false positives. The second experiment also had an initial 70% clustering ability that was later refined to 91% after applying second level clustering on the remaining unclustered malware. From those created families, the overall classification accuracy was 88.8% with 0% false positives.

7.2 Limitations and Future Work

Malware analysis in and of itself has inherent limitations, and there will never be a malware detector able to perfectly detect all malware with no false positives as proven

in [63]. To this extent I'll discuss some of the limitations of the malware analysis system in this dissertation:

1. *Encrypted or compressed malware*: Currently the system assumes that the malware is uncompressed and unencrypted. If the malware is not in its normal form it would mean that none of the features like API function calls and n-grams of instructions could be extracted. Decrypting and decompressing is an independent problem that is already being addressed to help solve this problem.
2. *Obfuscation*: A malware author may obfuscate the instructions through nop-insertion, code transposition, register reassignment, and instruction substitution. The serious problem is in instruction substitution as the system is immune to the rest of the mentioned methods. The system in this dissertation uses two feature sets, the API and the n-gram instructions; therefore, if the instructions were obfuscated the API function calls would still be able to classify the malware into a family as well as showing that the malware was probably obfuscated since the n-gram instructions did not match. The reason is that API calls are difficult to change without recompiling the code. I propose a future solution where the n-gram instruction feature set is immune to code obfuscation. This solution is carried out by isolating a set of instructions that can be substituted for other instructions or transformed into many instructions. This set of

instructions is then filtered from the set of features and hence the signature focuses on instructions that cannot be substituted or transformed easily.

The previous limitations are all open to improvement and future work. There is more potential future work and research associated with this dissertation like:

1. *Optimization*: The current algorithms have been developed to function correctly, but have not been fully optimized with speed and efficiency in mind. Therefore, there is much more that needs to be done in terms of optimization.
2. *Feature set*: This dissertation dealt with two different types of features the n-gram instructions and the API function calls. Future work could find further feature sets that might be a valuable addition to the current features. For example, features like basic block counts, which group instructions into larger chunks, may show interesting results.
3. *Classification methods*: There are two other classification methods that have not been implemented in the dissertation, but have been proposed and explained. It might be interesting to see how well they do if implemented and tested.

REFERENCES

REFERENCES

[1] F-Secure Corporation, "F-Secure Data Security Wrap-up 2008/2." [Online]. Available: http://www.f-secure.com/2008/2/index.html.

[2] "Computer viruses make it to orbit," *BBC*, 27-Aug-2008.

[3] "Annual Worldwide Economic Damages from Malware Exceed $13 Billion," *Computer Economics*, Jun-2007. [Online]. Available: http://www.computereconomics.com/article.cfm?id=1225.

[4] C. T. Reviews, *e-Study Guide for: Computer Science: Overview by J. Glenn Brookshear, ISBN 9780321524034*. Cram101 Textbook Reviews, 2012.

[5] Deerwester, Dumais, Furnas, Lanouauer, and Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, pp. 391–407, 1990.

[6] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse Processes*, vol. 25, no. 2–3, pp. 259–284, 1998.

[7] S. T. Dumais, "Latent Semantic Indexing (LSI) and TREC-2," presented at the The Second Text REtrieval Conference (TREC2), 1994, pp. 105–116.

[8] L. C. Freeman, *The Development Of Social Network Analysis: A Study In The Sociology Of Science*. Empirical Press, 2004.

[9] I.-H. Ting and H.-J. Wu, *Web Mining Applications in E-Commerce and E-Services*. Springer, 2009.

[10] P. Doreian, V. Batagelj, and A. Ferligoj, *Generalized Blockmodeling*. Cambridge University Press, 2004.

[11] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[12] Y. Dodge, *The Concise Encyclopedia of Statistics*. Springer, 2008.

[13] D. A. Grimes and K. F. Schulz, "Making sense of odds and odds ratios," *Obstet Gynecol*, vol. 111, no. 2 Pt 1, pp. 423–426, Feb. 2008.

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

[15] Atlantic, *Encyclopedia Of Information Technology*. Atlantic Publishers & Dist, 2007.

[16] M. E. Whitman and H. J. Mattord, *Principles of Information Security*. Cengage Learning, 2011.

[17] John Canavan, "The evolution of malicious IRC bots," presented at the Proceedings of Virus Bulletin Conference, 2005, pp. 104–114.

[18] M. C. Corporation, *Gods, Goddesses, And Mythology*. Marshall Cavendish, 2005.

[19] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2006.

[20] Sachin Shetty, “Introduction to Spyware Keyloggers,” 2010. [Online]. Available: http://www.symantec.com/connect/articles/introduction-spyware-keyloggers. [Accessed: 03-Feb-2013].
[21] Jeffrey 0. Kephart and William C. Arnold, “Automatic Extraction of Computer Virus Signatures,” presented at the 4th Virus Bulletin International Conference, 1994, pp. 178–184.
[22] William Arnold and Gerald Tesauro, “Automatically Generated Win32 Heuristic Virus Detection,” presented at the Proceedings of the 2000 International Virus Bulletin Conference, 2000.
[23] G. J. Tesauro, J. O. Kephart, and G. B. Sorkin, “Neural networks for computer virus recognition,” *IEEE Expert*, vol. 11, no. 4, pp. 5 –6, Aug. 1996.
[24] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, “Data mining methods for detection of new malicious executables,” presented at the 2001 IEEE Symposium on Security and Privacy, 2001. S P 2001. Proceedings, 2001, pp. 38 –49.
[25] Peter Miller, *Hexdump*. 2000.
[26] J. Z. Kolter and M. A. Maloof, “Learning to Detect and Classify Malicious Executables in the Wild,” *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, Dec. 2006.
[27] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “N-gram-based detection of new malicious code,” presented at the Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, 2004, vol. 2, pp. 41 – 42 vol.2.
[28] S. M. Tabish, M. Z. Shafiq, and M. Farooq, “Malware detection using statistical analysis of byte-level file content,” New York, NY, USA, 2009, pp. 23–31.
[29] D. M. Cai, J. Theiler, and M. Gokhale, “Detecting a malicious executable without prior knowledge of its patterns,” pp. 1–12, Mar. 2005.
[30] S. J. Stolfo, K. Wang, and W.-J. Li, “Towards Stealthy Malware Detection,” in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Springer US, 2007, pp. 231–249.
[31] M. Z. Shafiq, S. A. Khayam, and M. Farooq, “Embedded Malware Detection Using Markov n-Grams,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Zamboni, Ed. Springer Berlin Heidelberg, 2008, pp. 88–107.
[32] H. Khan, F. Mirza, and S. A. Khayam, “Determining malicious executable distinguishing attributes and low-complexity detection,” *J Comput Virol*, vol. 7, no. 2, pp. 95–105, May 2011.
[33] O. Henchiri and N. Japkowicz, “A Feature Selection and Evaluation Scheme for Computer Virus Detection,” presented at the Sixth International Conference on Data Mining, 2006. ICDM ’06, 2006, pp. 891 –895.
[34] R. Tian, L. M. Batten, and S. C. Versteeg, “Function length as a tool for malware classification,” in *3rd International Conference on Malicious and Unwanted Software, 2008. MALWARE 2008*, 2008, pp. 69 –76.
[35] D. Komashinskiy and I. Kotenko, “Malware Detection by Data Mining Techniques Based on Positionally Dependent Features,” in *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2010, pp. 617 –623.

[36] M. Alazab, R. Layton, S. Venkataraman, and P. Watters, “Malware Detection Based on Structural and Behavioural Features of API Calls,” *International Cyber Resilience conference*, Aug. 2010.
[37] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, “Malware phylogeny generation using permutations of code,” *J Comput Virol*, vol. 1, no. 1–2, pp. 13–23, Nov. 2005.
[38] Karypis G, *CLUTO: A clustering toolkit*. Department of Computer Science, University of Minnesota, 2003.
[39] M. Siddiqui, M. C. Wang, and J. Lee, “Data Mining Methods for Malware Detection using Instruction Sequences,” presented at the Artificial Intelligence and Applications, 2013.
[40] D. Bilar, “Opcodes as predictor for malware,” *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, Jan. 2007.
[41] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, “Idea: Opcode-Sequence-Based Malware Detection,” in *Engineering Secure Software and Systems*, F. Massacci, D. Wallach, and N. Zannone, Eds. Springer Berlin Heidelberg, 2010, pp. 35–43.
[42] Z. Wang, M. A. Nascimento, and M. H. MacGregor, “A Multidisciplinary Approach for Online Detection of X86 Malicious Executables,” presented at the Communication Networks and Services Research Conference (CNSR), 2010 Eighth Annual, 2010, pp. 160 –167.
[43] “74% of work PCs still run XP, and they’re 4.4 years old | ZDNet,” *ZDNet*, 13-Jul-2010. [Online]. Available: http://www.zdnet.com/blog/btl/74-of-work-pcs-still-run-xp-and-theyre-4-4-years-old/36641. [Accessed: 25-Jan-2013].
[44] “VX Heavens Virus Collection,” *VX Heavens website*. [Online]. Available: http://vx.netlux.org.
[45] Jibz, Qwerton, Snaker, and XineohP, *PEiD*. 2008.
[46] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools*. Addison-Wesley Pub. Co., 1986.
[47] Hexrays, *Ida Pro Disassembler and Debugger*. 2010.
[48] Oleh Yuschuk, *OllyDbg*. 2012.
[49] Vincent Bénony, *Hopper*. .
[50] *PE Explorer Disassembler*. Heaventools Software.
[51] C. Anley, J. Heasman, F. Lindner, and G. Richarte, *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2011.
[52] Edward J. Wegman and Yasmin H. Said, “Text Mining and Social Networks: Some Unexpected Connections,” presented at the HDM, 2008.
[53] K. Alsabti, S. Ranka, and V. Singh, “An efficient k-means clustering algorithm,” *Electrical Engineering and Computer Science*, Jan. 1997.
[54] H.-S. Park and C.-H. Jun, “A simple and fast algorithm for K-medoids clustering,” *Expert Systems with Applications*, vol. 36, no. 2, Part 2, pp. 3336–3341, Mar. 2009.
[55] Dan Pelleg and Andrew Moore, “X-means: Extending K-means with Efficient Estimation of the Number of Clusters,” presented at the ICML-2000, 2000.

[56] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *2nd International Conference on Knowledge Discovery and Data mining*, 1996, pp. 226–231.
[57] R Development Core Team, *R: a language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing, 2008.
[58] P. Orwick and G. Smith, *Developing Drivers with the Windows® Driver Foundation*. O’Reilly Media, Inc., 2010.
[59] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, “A.: Exploiting similarity between variants to defeat malware: ‘Vilo’ method for comparing and searching binary programs,” in *In: Proceedings of BlackHat DC 2007. (2007) https: //blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf*.
[60] Amazon, “Amazon Elastic Compute Cloud (Amazon EC2).” [Online]. Available: http://aws.amazon.com/ec2/.
[61] Simon Urbanek, *multicore*. .
[62] IObit, “cplexe.exe - Definition and Free Error Solution.” [Online]. Available: http://blog.iobit.com/cplexe-exe_3828.html.
[63] F. Cohen, “Computer viruses: theory and experiments,” *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, Feb. 1987.

CURRICULUM VITAE

Muhammad Aljammaz graduated from Robinson High School, Fairfax, Virginia, in 1998. He received his Bachelor of Science in Computer Science from George Mason University in 2002. He then went on to receive his Master of Science in Computer Science from George Mason University in 2004. He continued on with his doctoral degree in Information Technology at George Mason University giving many talks on malware analysis and filed for a patent based on his dissertation in 2012.