AN INVESTIGATION OF MACHINE LEARNING TECHNIQUES FOR
THE DETECTION OF UNKNOWN MALICIOUS CODE

A Project

Presented

to the Faculty of

California State University, Chico

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

© Mark Kellogg 2010

Fall 2010
AN INVESTIGATION OF MACHINE LEARNING TECHNIQUES FOR
THE DETECTION OF UNKNOWN MALICIOUS CODE

A Project

by

Mark Kellogg

Fall 2010

APPROVED BY THE DEAN OF GRADUATE STUDIES
AND VICE PROVOST FOR RESEARCH:

Katie Milo, Ed.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

Abdel-Moaty M. Fayek
Graduate Coordinator

Anne Keuneke, Ph.D., Chair

Benjoe Juliano, Ph.D.
PUBLICATION RIGHTS

No portion of this project may be reprinted or reproduced in any manner unacceptable to the usual copyright restrictions without the written permission of the author.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Publication Rights</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Problem Background</td>
<td>1</td>
</tr>
<tr>
<td>Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>Project Purpose</td>
<td>3</td>
</tr>
<tr>
<td>Project Goals</td>
<td>4</td>
</tr>
<tr>
<td>Preliminary Background</td>
<td>4</td>
</tr>
<tr>
<td>Outline of Remainder of Paper</td>
<td>6</td>
</tr>
<tr>
<td>II. Background</td>
<td>7</td>
</tr>
<tr>
<td>Decision Trees</td>
<td>7</td>
</tr>
<tr>
<td>Bayes’ Classification</td>
<td>15</td>
</tr>
<tr>
<td>Neural Networks</td>
<td>22</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>31</td>
</tr>
<tr>
<td>Self Organizing Maps</td>
<td>40</td>
</tr>
<tr>
<td>N-gram Analysis</td>
<td>44</td>
</tr>
<tr>
<td>Past/Current Application of Techniques to This</td>
<td>47</td>
</tr>
<tr>
<td>Problem Domain</td>
<td></td>
</tr>
<tr>
<td>III. Solution Implementation</td>
<td>53</td>
</tr>
<tr>
<td>Overview</td>
<td>53</td>
</tr>
<tr>
<td>Solution Design</td>
<td>54</td>
</tr>
<tr>
<td>Input Features</td>
<td>57</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Analysis Data</td>
<td>58</td>
</tr>
<tr>
<td>Application and Results</td>
<td>61</td>
</tr>
<tr>
<td>IV. Project Results</td>
<td>94</td>
</tr>
<tr>
<td>V. Conclusion</td>
<td>97</td>
</tr>
<tr>
<td>VI. Future Research</td>
<td>99</td>
</tr>
<tr>
<td>Preexisting Knowledge</td>
<td>99</td>
</tr>
<tr>
<td>Malware Pre-classification</td>
<td>100</td>
</tr>
<tr>
<td>Multi-technique Approaches</td>
<td>101</td>
</tr>
<tr>
<td>Deeper Testing</td>
<td>101</td>
</tr>
<tr>
<td>Feature Pruning with Decision Trees</td>
<td>101</td>
</tr>
<tr>
<td>Input Feature Format</td>
<td>102</td>
</tr>
<tr>
<td>References</td>
<td>103</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sample Data Set</td>
<td>8</td>
</tr>
<tr>
<td>2. Joint Bayesian Probabilities</td>
<td>20</td>
</tr>
<tr>
<td>3. Decision Tree Results, Set A</td>
<td>63</td>
</tr>
<tr>
<td>4. Decision Tree Results, Set B</td>
<td>63</td>
</tr>
<tr>
<td>5. Decision Tree Results, Set C</td>
<td>64</td>
</tr>
<tr>
<td>6. Bayes’ Classification Results, Data Set C</td>
<td>68</td>
</tr>
<tr>
<td>7. Neural Network Results, Set A</td>
<td>73</td>
</tr>
<tr>
<td>8. Neural Network Results, Set B</td>
<td>73</td>
</tr>
<tr>
<td>9. Neural Network Results, Set C</td>
<td>74</td>
</tr>
<tr>
<td>10. Self-Organizing Maps Results, Set A</td>
<td>85</td>
</tr>
<tr>
<td>11. Self-Organizing Maps Results, Set B</td>
<td>86</td>
</tr>
<tr>
<td>12. Self-Organizing Maps Results, Set C</td>
<td>87</td>
</tr>
<tr>
<td>13. N-gram Analysis Results After Modification</td>
<td>91</td>
</tr>
<tr>
<td>14. Exclusively Benign N-grams</td>
<td>92</td>
</tr>
<tr>
<td>15. Exclusively Malicious N-grams</td>
<td>92</td>
</tr>
<tr>
<td>16. N-grams in Both Benign and Malicious Executables</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>1. Descision Tree A</td>
<td>9</td>
</tr>
<tr>
<td>2. Decision Tree B</td>
<td>9</td>
</tr>
<tr>
<td>3. Data Set Partitioning</td>
<td>10</td>
</tr>
<tr>
<td>4. Split Along Outcome Boundaries</td>
<td>11</td>
</tr>
<tr>
<td>5. Grouping of Outcomes</td>
<td>12</td>
</tr>
<tr>
<td>6. Example Bayesian Network</td>
<td>20</td>
</tr>
<tr>
<td>7. Naïve Bayes’ Classifier</td>
<td>21</td>
</tr>
<tr>
<td>8. Multiple Discriminant Planes</td>
<td>33</td>
</tr>
<tr>
<td>9. Convex Hulls</td>
<td>34</td>
</tr>
<tr>
<td>10. Maximizing Margin</td>
<td>35</td>
</tr>
<tr>
<td>11. Linearly Inseparable Data</td>
<td>37</td>
</tr>
<tr>
<td>12. Linearly Inseparable Data Requiring Quadratic</td>
<td>39</td>
</tr>
<tr>
<td>13. Mapping the Original Two Dimensional Input Space to the Five Dimensional Input Space</td>
<td>40</td>
</tr>
<tr>
<td>14. A Rectangular SOM</td>
<td>42</td>
</tr>
<tr>
<td>15. A Hexagonal SOM</td>
<td>43</td>
</tr>
<tr>
<td>16. Dynamic API Call Interception</td>
<td>56</td>
</tr>
<tr>
<td>17. Executable Code Transformation</td>
<td>56</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>18. Tracked Win32 Libraries</td>
<td>57</td>
</tr>
<tr>
<td>19. Partial Execution Log from Win32.HLLP.Semisoft.a</td>
<td>59</td>
</tr>
<tr>
<td>20. Number of Win32 Functions Encountered During Run-time Analysis</td>
<td>60</td>
</tr>
<tr>
<td>21. Support Vector Machine Results, Set A</td>
<td>79</td>
</tr>
<tr>
<td>22. Support Vector Machine Results, Set B</td>
<td>80</td>
</tr>
<tr>
<td>23. Support Vector Machine Results, Set C</td>
<td>80</td>
</tr>
</tbody>
</table>
ABSTRACT

AN INVESTIGATION OF MACHINE LEARNING TECHNIQUES FOR THE DETECTION OF UNKNOWN MALICIOUS CODE

by

© Mark Kellogg 2010

Master of Science in Computer Science

California State University, Chico

Fall 2010

Traditional forms of anti-virus like signature recognition are lacking in that they can only fight against viruses after they have created a significant enough impact to be identified as malicious. Recently researchers have been applying modern machine learning techniques to the task of detecting malicious programs before they even have a chance to cause any damage. This paper investigates a selection of these techniques in a controlled environment on the Microsoft Windows platform. The selected techniques are tested on the same set of malicious code and the resulting performance and effectiveness of each is analyzed and discussed. The paper concludes with a discussion of potential directions the research could be taken in the future.
CHAPTER I

INTRODUCTION

Malware has long been and continues to be a danger to anyone using a personal computing device (PC, PDA, etc.). New forms of malware are created everyday and the rate is only expected to increase (at least within certain types of malware, such as ID Fraud Trojans [1]). The proliferation of the Internet into so many facets of everyday life creates an environment rich with opportunity for malware to spread quickly and cause substantial damage.

Problem Background

Traditional defense against malware performed by tools such as anti-virus programs has been in the form of signature recognition [2]. When a new piece of malware is discovered, its binary signature is determined through analysis of the instructions that make up the executable code. The newly discovered signature is then pushed out to anti-virus clients operating on customers’ personal computers. Unfortunately, this is only a reactive approach. Between the time that the malware is released and the time anti-virus clients are updated with its signature, it can run rampant and unchecked. The result is a potential for malicious code to cause millions of dollars in damage, infect millions of machines, and compromise untold amounts of private information, even on machines with up to date anti-virus software.
What is needed here is a *proactive* approach. Ideally, programs on a personal computer could be scanned, analyzed, and classified as malware before their signatures are known and before they have a chance to do damage. Recent research has been devoted to this very idea. Many researchers have applied well known techniques in machine learning to the problem and results from many of these approaches are promising (discussed further in Chapter II, “Bayes’ Classification”).

**Problem Statement**

No one technique is flawless at the task of detecting malicious code and the constantly changing nature of malware will prevent this from ever being true. A given technique may be well suited for only certain types of malware. Therefore, to more effectively utilize machine learning techniques to detect malware, it may be necessary to combine them and focus them where they are most effective.

To better understand the relative effectiveness between the various approaches documented in available research, it is necessary to test all on the same control group of malicious and benign executables in the same controlled environment. Note that although this would be the desired approach, within the research examined for this project, discrepancies and inconsistencies between the individual works prevented this goal from being achieved:

1. Not all of the existing research documents the source(s) from which virus test sets were acquired. Even within the group that does document their sources, there are some that don’t provide any detailed information about the viruses that were obtained and/or tested.
2. A great deal of inconsistency exists in both the executable platform on which the malware runs, and in the features chosen to be input to the machine learning techniques.

Project Purpose

The purpose of this project was to investigate these machine learning techniques in the domain of classifying benign and malicious executables, and form a comparison of their effectiveness in a controlled environment. To control the environment in which these techniques were tested as much as possible, the following constraints were observed:

1. Each machine learning technique was trained and tested on the exact same training and testing set, respectively.

2. The focus of the investigation was malicious code that exists in Microsoft Windows Portable Executable (PE) format. The reason for this is that Windows has such a large share of the personal computing market and consequently there are a huge number of existing viruses written for it.

3. Inputs were restricted to a single feature type: Win32 API calls. For a Win32 program, all interaction with the surrounding environment is accomplished through calls to the Win32 API. Therefore a piece of malware that has the ability to cause damage/destruction to its host machine would have to do so through this API. The Win32 calls invoked by a program form a profile of that program’s functionality.
Project Goals

Through the investigation of the machine learning techniques covered in this project, three specific goals were hoped to be achieved:

1. **Individual technique assessment:** Determination of each technique’s individual ability to detect malicious code on the Win32 platform.

2. **Overall performance:** Determination of the techniques that are best suited for the task of detecting malicious code as stand-alone mechanisms, in addition to which techniques are least desirable for the task.

3. **Input feature assessment:** An indication of how well the chosen input feature (Win32 API functions) performed as input for the targeted machine learning techniques, and therefore how well they represented a functional view of Win32 applications.

Preliminary Background

Terms

**Anti-virus Software.** A piece of software engineered to search through computer programs, identify malicious code, and take action to mitigate or prevent the effects of that code.

**Machine Learning.** A class of artificial intelligence techniques that modify their own data or structure as a result of their past experience or performance.

**Malware.** Short for malicious software, malware is a catch-all term referring to hostile and intrusive computer programs that perform actions without the informed consent of the user. Programs that fall into this category include viruses, trojans, spyware,
worms and more. Sometimes these are collectively referred to as “computer viruses” as well.

**Virus Signature.** A signature, in the malware world, is a series of characteristic instructions that uniquely identify a given piece of malware. Anti-virus software looks for these signatures to identify code as malware.

**Win32 API.** The Win32 API (Application Programmer Interface) is a collection of pre-existing executable code written specifically for Windows by Microsoft that can be used by programmers to accomplish a myriad of tasks. File output, interaction with the user, network communication and much more is accomplished through the Win32 API and therefore programs written for the Win32 platform must hook into it at some point to do anything useful.

**Win32 PE Executable.** Microsoft’s proprietary executable format. All programs built for the Win32 platform are stored in this format, and therefore have a very precise and well known method of connecting and interacting with the operating system.

**Win32 Platform.** The programming environment that is available within the Microsoft Windows family of operating systems. Programs that are written for this platform cannot run under other platforms like UNIX, even though they may be built for the same type of processor.

**Techniques to be Covered**

Testing all known AI techniques in the area of malicious code detection is simply not feasible, so this project focuses on a subset of well-known machine learning techniques:
• Decision Trees [3]
• Bayes’ Classification [4]
• Artificial Neural Networks [5]
• Support Vector Machines [6, 7]
• Self Organizing Maps [8]
• N-gram Analysis [9]

Outline of Remainder of Paper

The remainder of this paper is organized as follows: Chapter II is dedicated to a discussion of modern artificial intelligence techniques and current research in applying those techniques to the task of identifying malicious code. The first half provides background information for the various techniques used. The reader may skip to “Past/Current Application of Techniques to This Problem Domain” if already familiar with these machine learning approaches and wishes to proceed to related works using their application in this problem domain. Chapter III discusses the solution developed for this project: its purpose, design and implementation, and then proceeds to discuss and analyze the results of each test in the context of that test. Chapter IV discusses the results in the context of the entire project; comparisons and analysis are made between different techniques and their results. Chapter V is dedicated to the project wrap-up with final thoughts and conclusions. Chapter VI discusses potential directions the research could be taken in the future.
CHAPTER II

BACKGROUND

Decision Trees

Overview

Decision trees are visual and analytical decision support data structures [3]. In very simplified terms, they are a graphical approach to solving if-then problems [10]. One of their most common uses is classification. A decision tree is no more than a conventional tree data structure comprised of nodes and leaves. There is a single root node, and all the nodes in the tree represent attribute tests [3]. For each of these attributes there is either a child node or child leaf, and these children correspond to the outcome of said attribute tests.

The process of building a decision tree model begins with a data set, called the training set. Each element or record contains a set of attributes, one member of which is the class for that element. Record sets are homogeneous; all of the records have the same structure [11]. Given this data set, a model is derived for the class attributes as a function of the values of the other attributes. The goal is to be able to classify previously unseen records as accurately as possible. To determine the model’s accuracy, a new data set of the same structure as the training set, called the test set, is used. The testing stage as known as deduction and the training stage is known as induction [11].
Table 1 contains a sample data set correlating three weather attributes to the playing or canceling of a football game. The “Game status” column corresponds to the class attribute of each record. Figure 1 and Figure 2 both show decision trees based on the data in Table 1. They both produce the same output, but have significantly different structures:

<table>
<thead>
<tr>
<th>Weather</th>
<th>Temp.</th>
<th>Windy</th>
<th>Game status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>70</td>
<td>No</td>
<td>Played</td>
</tr>
<tr>
<td>Sunny</td>
<td>60</td>
<td>Yes</td>
<td>Played</td>
</tr>
<tr>
<td>Sunny</td>
<td>45</td>
<td>Yes</td>
<td>Cancelled</td>
</tr>
<tr>
<td>Cloudy</td>
<td>60</td>
<td>No</td>
<td>Played</td>
</tr>
<tr>
<td>Rainy</td>
<td>70</td>
<td>No</td>
<td>Played</td>
</tr>
<tr>
<td>Rainy</td>
<td>75</td>
<td>No</td>
<td>Played</td>
</tr>
<tr>
<td>Cloudy</td>
<td>50</td>
<td>No</td>
<td>Cancelled</td>
</tr>
<tr>
<td>Sunny</td>
<td>50</td>
<td>Yes</td>
<td>Cancelled</td>
</tr>
<tr>
<td>Rainy</td>
<td>60</td>
<td>No</td>
<td>Cancelled</td>
</tr>
</tbody>
</table>

Creating an arbitrary decision tree for a given data set is a trivial task. Creating an optimized one is not.

**Algorithms**

The two most prominent algorithms for deriving decision trees from arbitrary data sets are ID3 and C4.5. C4.5 is an extension of ID3 and both algorithms were developed by J. R. Quinlan [12].

1. **ID3.** ID3 is based on information theory and its basic principles are as follows:
   - Each node should be associated with the non-categorical attribute that is most informative among the attributes not yet considered in the path from the root. This quality of the tree decides how “good” the tree is [12].
Fig. 1. Decision tree A.

Fig. 2. Decision tree B.
• Simple trees are preferred over complex trees, based on the idea that simple trees are more accurate classifiers of future inputs [13].

• A measure called Entropy is used to determine how informative a node is [12].

2. **C4.5.** C4.5 was also developed by J. R. Quinlan and is simply an extension to ID3. It accounts for unavailable values, continuous attribute value ranges, pruning of decision trees, rule derivation, and more [13]. It generates more robust and efficient trees than ID3.

3. **Algorithm description.** As stated before, the heart of the algorithm builds a decision tree from an arbitrary data set $D$, where $D$ is a collection of homogeneous objects that belong to at least two classes, and uses a divide and conquer approach to partition the set into the resulting tree [13]. Given tree $D$, let $B$ be a test with possible outcomes $\{b_1, b_2, b_3, \ldots b_t\}$ that produces a non-trivial partition of $D$, and let $d_i$ be the cases in $D$ that have outcome $b_i$ in $B$. $B$ will produce a partition $\{d_1, d_2, d_3, \ldots d_n\}$ such that $d_i = \{x|B(x)=b_i\}$. This results in the following tree [13] (Figure 3)

![Fig. 3. Data set partitioning.](image)
Figure 3 shows a partitioning of data set B. Quinlan uses the notion of *expected information* from communication theory to determine the quality of a partitioning instance. In order to determine how to maximize information of a particular node, the following questions can be asked [11]:

1. How will the attribute test condition be specified?
2. What is the best split among outcomes?

The answer to (1) depends on a couple of factors [11]:

- Type of the attribute (nominal, ordinal, continuous)
- Number of ways to split (binary or multiple)

With nominal and ordinal attributes, splitting can occur evenly along outcome boundaries, or multiple outcomes can be grouped into a single outcome branch so that the number of child branches is at least one less than the number of possible outcomes.

Figure 4 and Figure 5 show two different splits among three different types of weather. With continuous attributes, a couple different approaches can be used [11]:

- **Discretization**: Divide the continuous range into discrete sections, resulting in a set of ordinal values corresponding to each section.

![Fig. 4. Split along outcome boundaries.](image)
Fig. 5. Grouping of outcomes.

- **Binary Decision**: A single splitting point can be chosen in the continuous range, thereby creating two ordinal values.

As for (2), the algorithm relies on greedy search, selecting the candidate test that maximizes a heuristic splitting criterion [12]. It follows the principle of Occam's razor in attempting to create the smallest decision tree possible. The reasoning behind this is that a simpler tree will likely have a higher predictive accuracy [13]. The splitting criterion is based on a concept known as *Entropy*. The ID3 and C4.5 algorithms are based on knowledge gained from communication theory, in which expected information is a number describing a set of messages $M = \{m_1, m_2, m_3, \ldots, m_n\}$. Each member of $M$, $m_i$ has a probability of $P(m_i)$ and contains certain information $I(m_i)$. The correlation between the two values is $I(m_i) = -\log_2 P(m_i)$ [12]. This means that the information value is an inverse function of the probability. The *Entropy* of a message set is the sum of the information of the messages multiplied by their probability [11]:

$$
Entropy = -\sum_{i=1}^{n} P(m_i) \log_2 P(m_i)
$$
This formula produces a larger number from calculations on a highly non-homogeneous set, and lower numbers from highly homogeneous sets.

Since expected information can be calculated, information gain from a parent node to its children can also be calculated. If a parent node \( P \) is split into \( k \) partitions, where \( n_i \) is the number of records in partition \( i \), the information gain can be calculated as [11]:

\[
GAIN = \text{Entropy}(P) - \sum_{i=1}^{n} \frac{n_i}{n} \text{Entropy}(i)
\]

This measures the reduction in Entropy achieved by the split among the children of \( P \). The idea is to choose the split that maximizes the reduction, and therefore maximizes \( GAIN \). The primary drawback of this calculation is that it tends to prefer splits that result in a large number of partitions where each is small but pure [11]. To overcome this drawback, a measure called \( GAIN RATIO \) is used [11]:

\[
GAIN RATIO = \frac{GAIN}{\text{SplitINFO}} \quad \text{where} \quad \text{SplitINFO} = -\sum_{i=1}^{k} \frac{n_i}{n} \log \frac{n_i}{n}
\]

This new formula adjusts \( GAIN \) by the entropy of the partitioning (\( \text{SplitINFO} \)) so that partitioning with higher entropy is penalized.

With these two algorithms come two assumptions:

- A decision tree derived from a data set \( D \) will classify unseen objects proportionally to their occurrence in \( D \) [13]. Given class set \( C = \{C_1, C_2, \ldots, C_n\} \) as the classes present in data set \( D \), and class frequency set \( F \) where the frequency of class \( C_i \) in \( D \) is \( F_i \), an unseen object classified using the tree for \( D \) will belong to class \( C_i \) with probability \( F_i/(F_1 + F_2 + \ldots + F_n) \)
Given two arbitrary classes $P$ and $N$, a decision tree can be regarded as the source of a message, $P$ or $N$ [11]. If $p =$ the frequency of $P$, and $n =$ the frequency of $N$, then the entropy associated with the message set $M = \{P,N\}$ is:

$$Ent\text{ropy} = \frac{-p}{(p + n)} \log_2 \left( \frac{p}{(p + n)} \right) - \frac{n}{(p + n)} \log_2 \left( \frac{n}{(p + n)} \right)$$

### Advantages and Drawbacks to Decision Trees

1. **Advantages**
   - They are computationally inexpensive to construct [11].
   - They are very fast at classifying unknown records [11].
   - Accuracy is comparable to other data mining techniques [11].

2. **Drawbacks**
   - The ID3 and C4.5 algorithms are not guaranteed to find the simplest trees because they only use heuristics [13].
   - They operate as a black box mechanism; an optimized tree for a complex will be very difficult to interpret visually [12].
   - The method for building the trees is not incremental; new training data cannot be considered without considering the classification of previous data. This is due to the fact that the new data may produce classifications that conflict with the current structure of the tree; partitioning decisions must be made in the context of the entire data set.
Bayes’ Classification

Introduction

Bayesian networks are a graphical method of representing probabilistic relationships among a set of variables [4]. These mathematical relationships can aid in decision making in the face of uncertainty [14]. The logic driving these networks is based on Bayes' rule, which is outlined in the following section. Bayesian networks possess many qualities that make them viable options in the arena of data analysis (classification, regression, and clustering). These qualities are outlined below.

1. Bayesian networks can readily handle incomplete data sets [15]. In an arbitrary regression or classification problem involving two input variables that are strongly anti-correlated, standard supervised learning techniques perform well as long as all inputs are measured in every case [15]. In the case of one of the inputs not being observed, however, most models will result in an inaccurate prediction. This is due to the fact that they do not encode the correlation between variables. Bayesian networks do exactly this.

2. Bayesian networks allow for the learning and modeling of casual relationships [15]. Learning about casual relationships aids in the understanding of a given problem domain. Knowledge of casual relationships allows for predictions in the presence of interventions, and allows for inferences to be made about the effects of variables on other variables without direct experimentation [15].

3. Bayesian networks allow for the straightforward and logical combination of both prior/domain knowledge and data. This ability is very useful and important when real data are scarce or expensive. Bayesian networks employ casual semantics that allow
for the straightforward encoding of prior knowledge. Prior knowledge and data can be combined with well known Bayesian statistic techniques because the strength of casual relationships is encoded via probabilities [15].

4. Bayesian networks offer an efficient solution to the problem of over fitting [15]. When employing this type of model there is no need to exclude any portion of the source data during training. The Bayesian approach allows for models to be transformed so that all data, including outliers, can be incorporated.

To fully understand Bayesian networks, it is important to understand the distinction between the classical approach to statistics and probability and the Bayesian approach. In the classical world, a probability is a property of the physical world. An example: given a bag of 5 red marbles and 5 blue marbles, what is the probability that a randomly extracted marble is red? In the classical world, it is 50%, but it may not be so in the Bayesian world. In the Bayesian world, the probability of a given event is a person’s degree of belief in that event [15]. It is a property of the person who assigns the probability. An important difference between classical and Bayesian statistics is that repeated trials are not necessary to measure Bayesian probability, since it is based on belief.

**Probability**

The workings of Bayes’ classifiers are founded in conditional probability theory. Consider two events, $A$ and $B$, where the occurrence of event $B$ depends on the occurrence of event $A$. The probability of $B$ occurring given that $A$ has occurred is denoted as $P(B|A)$ [14]. The probability of both $A$ and $B$ occurring can be written as:

$$P(A \cap B) = P(A)P(B|A)$$
Essentially this says that the probability of both $A$ and $B$ occurring is equal to the probability of $A$ occurring multiplied by the probability of $B$ occurring given that $A$ has occurred. To extend this to beyond two events is simply [14]:

$$P(E_1 \cap E_2 \cap E_3 \cap \ldots \cap E_n) = P(E_1)P(E_2|E_1)P(E_3|E_1, E_2) \ldots P(E_n|E_{n-1}, \ldots, E_1)$$

Often the piece of information we are interested in is the probability that an event will occur given that some other event has occurred. The first equation can be rewritten to show this [14]:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

In the same vein, the probability of event $A$ occurring given event $B$ has occurred is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Rearranging and combining the two preceding equations yields:

$$P(A|B)P(B) = P(A \cap B) = P(B|A)P(A)$$

Removing the middle term and dividing through by $P(A)$ yields:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

This final equation is known as Bayes’ rule [14], and is the basis for Bayesian networks and Bayes’ classification. In the real world, there are countless examples of conditional probability where Bayes’ probability can be applied. However these problems are typically NP-hard therefore caution must be exercised when determining how to approach such problems [16].
Bayesian Networks

A Bayesian network is a graphical model that encodes the joint probability distribution for a set of variables, and represents a Bayes’ classifier in graphical form [17]. The aforementioned probability can be either classical or Bayesian. For a given set of variables $X = \{x_1, x_2, ... x_n\}$, a Bayesian network for $X$ will include two items:

1. An acyclic graph $S$. This structure encodes the conditional independence information about variables in $X$, and its nodes are in one-to-one correspondence with those variables.

2. A set of $P$ local probability distributions for each variable.

Together, $S$ and $P$ define the joint probability distribution of $X$. The absence of arcs between nodes in $S$ represents conditional independence between the corresponding variables in $X$. The probability distribution is formalized as [15]:

$$p(x) = \prod_{i=1}^{n} p(x_i | pa_i)$$

(1)

Where $pa_i$ is the parent node of $x_i$ in $S$. Applying the chain rule of probability to the above equation yields:

$$p(x) = \prod_{i=1}^{n} p(x_i | x_1, x_2, ..., x_{i-1})$$

(2)

For each $x_i \in X$, there will be a sub set $\prod_i \subseteq \{x_1, x_2, ..., x_{i-1}\}$, such that $x_i$ and $\{x_1, x_2, ..., x_{i-1}\} | \prod_i$ are conditionally independent given $\prod_i$ [15]. This can be written as:

$$p(x_i | x_1, x_2, ..., x_{i-1}) = p(x_i | x_{\prod_i})$$

(3)
Combining equations (2) and (3) yields:

\[ p(x) = \prod_{i=1}^{n} p(x_i | \pi_i) \]  

When equations (1) and (3) are compared it can be seen that the variable sets 
\{\Pi_1, \Pi_2, ..., \Pi_n\} correspond to the node parents \{pa_1, pa_2, ..., pa_n\}. These equations fully specify the arcs in \( S \) [15].

**Visualization**

As an example, take a volcano that could erupt this year, and could erupt next year. These are not independent events because if the volcano erupts this year, the pressure buildup that produced the eruption will be relieved, thereby delaying any subsequent eruptions (obviously a very simplified version of reality). What is the probability that the volcano erupts both years? To determine this value, we need to know the probability that the volcano will erupt this year (ignoring any events of the past), and the conditional probability that the volcano will erupt next year given that it erupted this year. Suppose events \( E_1 \) = an eruption this year, \( E_2 \) = an eruption next year, \( P(E_1) = .4 \) and \( P(E_2|E_1) = .1 \). The calculation to determine \( P(E_1 \cap E_2) \) is:

\[ P(E_1 \cap E_2) = P(E_1)P(E_2|E_1) \]

from the first equation in part B. Based on this formula, the joint probabilities for all combinations of possibilities can be calculated.

Table 2 shows an arbitrary joint probability table for \( E_1 \) and \( E_2 \). The values for \( P(E_2|\neg E_1) \) and \( P(\neg E_2|\neg E_1) \) were the same as \( P(E_1) \) and \( P(\neg E_1) \) respectively. This information can be expressed as a network diagram (Figure 6).
### TABLE 2
**JOINT BAYESIAN PROBABILITIES**

<table>
<thead>
<tr>
<th></th>
<th>E2</th>
<th>~E2</th>
<th>Marginal Probability E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>P(E1 ∩ E2) = .04</td>
<td>P(E1 ∩ ~E2) = .36</td>
<td>.4</td>
</tr>
<tr>
<td>~E1</td>
<td>P(~E1 ∩ E2) = .24</td>
<td>P(~E1 ∩ ~E2) = .36</td>
<td>.6</td>
</tr>
<tr>
<td>Marginal probability of E2</td>
<td>.28</td>
<td>.72</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 displays the exact same information as Table 2, but in traditional Bayesian network form. An advantage to viewing it in graphical form is that conditional dependencies and lack thereof can be seen through the existence or absence of arcs, respectively [16]. The type of network displayed in Figure 6 is a general Bayesian network and is flexible regarding how conditional independencies between attributes can be represented. Other more restrictive and constrained forms do exist, however. One of the most common of these is the Naïve Bayes’ (NB) classifier. The structure of such a classifier is shown in Figure 7.
Figure 7 shows a traditional Naive Bayes' classifier. From a given set of training data, an NB classifier will learn the conditional probability of each attribute $A_i$ relative to a given class $C$ [17]. During classification for a given instance of the attribute set $A_1, A_2, ... A_n$, Bayes' rule is applied to compute the likelihood of $C$ for the values of those attributes. There is a strong independence assumption in this calculation: all attributes are conditionally independent from each other given the value of $C$ [17].

Extensions to the NB model, such as Tree Augmented Naïve Bayes (TAN) and Forest Augmented Naïve Bayes (FAN), have been developed to relax the restrictions of NB and make it more adaptable to real life scenarios [18].

**Limitations**

1. **Network foundations.** When a Bayesian network is based on prior knowledge, the performance of that network will only be as good as the prior knowledge is accurate [16]. Extreme expectations of the quality of these beliefs will distort the entire network, and therefore caution must be exercised when selecting the statistical model on which the network is based.
2. **Computational complexity.** To calculate the probability of a branch in a general Bayesian network, all branches must be calculated [16]. The process of network discovery is an NP-hard problem, and therefore may be too costly to perform for a large number of variables. More constrained forms such as NB, TAN, and FAN help to mitigate this problem by introducing restrictions that reduce this complexity [18].

**Neural Networks**

In the simplest terms, a neural network is a computational model that is intended to mimic the human brain in the way that it processes information [19]. The human brain is made up of billions of neurons. Each one of these neurons is connected to thousands of other neurons. These connections can be strengthened or weakened to reinforce learning and represent knowledge. This results in an extremely complex computational network with an extraordinary amount of processing power.

Neural networks in real life occur at a much smaller scale than that of the human brain, or even the brains of small animals. While the human brain contains about $10^{11}$ neurons, each with connections to about $10^4$ other neurons, a typical neural network may only contain about a dozen neurons [14]. Other networks may contain hundreds or even thousands of neurons for use in specific application domains, but this is still far simpler than the brain. Processing power just isn’t currently available to match what the human brain can do.

Neural networks are part of the connectionist area of artificial intelligence research. They are not well suited for straight numerical processing. They are great for more low level, or “fuzzy” tasks like speech recognition, and in the case of this project,
numerical function approximation. This is due to the parallel nature of neural networks, which makes them able to handle problems with simultaneous constraints; constraints which may be somewhat unclear or not even known to their developers [20]. Neural networks cannot do anything that cannot be done with traditional computational models. However, they can do things that would otherwise be very difficult. This is the case in situations where there may be an algorithm, but it might be unknown or have too many variables.

**Popular Uses of Neural Networks**

1. **Function approximation.** Neural networks can be used to model mathematical functions. This is useful when trying to fit data to a function in order to better understand it, but the data set itself does not perfectly fit any reasonably simple or easily recognizable mathematical function. Instead the function can be approximated because it has so many variables or unknowns.

2. **Classification.** Neural networks are much better suited than traditional symbolic AI techniques at “fuzzy” classification techniques, such as pattern and sequence recognition [20]:
   a. Pattern recognition: Face or object recognition; visual identification.
   b. Sequence recognition: Gesture, speech, or handwritten text recognition (OCR).

3. **Decision making.** Neural networks can be used in video games for decision making. In a well known instance, a neural network was used to play backgammon. It played itself over and over and through this process it learned how to be a champion level player [21].
Structure

As previously stated, a neural network’s physical layout is inspired by the topology of neurons and their connections in the human brain. It consists of a collection of basic processing units which are connected to each other via weighted connections. It is over these connections that the units communicate. Defined by [19] a neural network has:

1. A set of processing units (neurons or “cells”)
2. A state of activation for each unit, which is the output for that unit.
3. Connections between the units. In a feed-forward non-recurrent network, units are connected to other units in either the next or previous layer, but not within the same layer. Each connection has a weight, which determines the signal strength between those two units.
4. An activation function for a unit. It determines the output of the units based on the effective input to the unit.
5. An external input, also called the bias, for each unit.

Each processing unit takes its input from some of the units to which it is connected and uses it to compute its output signal, which is propagated to other units. The units each fall into one of three categories [19]:

1. **Input units**: These units receive input from outside the network and propagate output into the network.
2. **Output units**: These units receive input from inside the network and propagate output externally.
3. **Hidden units**: These units receive input from inside the network and their output stays within the network as well.

**Operation**

Input to an input unit is simply whatever value is received from the external source. When computing the input to a unit that takes its input from other units, the most basic and commonly used method is to take a weighted sum of all the inputs to that unit and then add the bias or offset term for that unit. That is for unit $k$, the effective input is:

$$s_k(t) = \sum w_{jk}(t) y_j(t) + bias_k$$

Where $s_k(t)$ is the effective input for unit $k$, $w_{jk}(t)$ is the weight of the connection from unit $j$ to unit $k$, $y_j(t)$ is the value of unit $j$ and $bias_k$ is the bias or offset term for unit $k$. This rule is one of the more commonly used rules and units that use this method are known as sigma units [19], and the rule is called the sigma rule.

Once the effective input for a unit has been calculated, it can be used to determine the activation of the unit and therefore the unit’s output. To this end, each unit has an activation function which takes the effective input and produces the output value for that unit. There are multiple classes of functions that may be used for this purpose, but generally some sort of threshold function is used. Often a smoothly limiting threshold function called the sigmoid function is used:

$$F(s_k) = \frac{1}{1 + e^{-s_k}}$$

This produces a smooth S-shaped function that ranges between 0 and 1 [19].
Training and Back-Propagation

Performance of a neural network is influenced by two different issues: the representational power of the network and the network’s learning algorithm. The representational power of the network is a measure of the network’s ability to represent a desired function. Of course, neural networks can only approximate functions and will never be completely error free. The learning algorithm of the network is how its weights are adjusted to arrive at the ideal error level.

Back-propagation is a technique that is used to adjust all the weights in a feed-forward neural network with the goal of minimizing output error [5]. Back-propagation begins with the output units and works backwards towards the input units, adjusting weights along the way. Back-propagation is an implementation of a method known as gradient descent where the goal is to adjust weights based on their contribution to the error in the output. The idea is to make a change in the weight proportional to the negative derivative of the error as measured on the current input with respect to each weight [22].

The following derivation is largely adapted from [19]. The function for the output of a given unit is:

\[ y_k(t) = F(s_k(t)) \quad \text{where} \quad s_k(t) = \sum w_{jk}(t) y_j(t) + bias_k \]

The function \( y_k(t) \) represents the total output and \( F() \) is the activation function. An error rule is needed to define what the error is for a given unit, based on its actual output and the desired output. A common rule called the least mean squared (LMS) rule is often used for this:
Where \( N_o \) is the number of output units, \( d_o \) is the desired output of unit \( o \) and \( y_o \) is the actual output of unit \( o \). The change in weight needed for a given connection in order to reduce error will be proportional to the derivative of the error of the unit that connection connects to with respect to the current weight of that connection:

\[
\Delta w_{jk} = \gamma \frac{\partial E}{\partial w_{jk}}
\]

where \( \gamma \) is a term known as the learning rate. Using the chain rule the following can be written:

\[
\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}
\]

By the sigma rule defined above, the second factor is:

\[
\frac{\partial s_k}{\partial w_{jk}} = y
\]

When \( \delta_k \) is defined as

\[
\delta_k = -\frac{\partial E}{\partial s_k}
\]

an update rule is derived that will result in a gradient descent on the surface of the error equation if weight changes are made according to:

\[
\Delta w_{jk} = \gamma \delta_k y_j
\]

This equation is known as the weight update rule. The problem now is figuring the value for \( \delta \) for each \( k \) in the network. It turns out there is a simple way to recursively compute these values by propagation error signals back through the network. Going back to the chain rule a computation can be made to build a partial derivative as
the product of two factors, one factor reflecting the change in error as a function of the output of the unit and one reflecting the change in the output as a function of changes in the input:

\[ \delta_k = - \frac{\partial E}{\partial s_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial s_k} \]

Using the equation:

\[ y_k(t) = F(s_k(t)) \]

the second factor is computed as:

\[ \frac{\partial y_k}{\partial s_k} = F'(s_k) \]

The second factor is simply the derivative of the activation function for unit \( k \) evaluated at \( s_k \). Now the first factor is computed. For this, two separate cases must be observed. The first case is for an output unit:

\[ \frac{\partial E}{\partial y_k} = -(d_o - y_o) \]

Substituting this and the second factor computed earlier in the equation for \( \delta \), produces:

\[ \delta_o = -(d_o - y_o) F'(s_o) \]

for any output unit \( o \). If the unit under question is not a unit, but instead is a hidden unit \( h \), the equation is a bit different. The error measure in this case is written as a function of the net inputs from the hidden to the output layer. Skipping a little of the detail (the full derivation can be found in [19]), the result is:

\[ \delta_h = F'(s_h) \sum_{o=1}^{N_o} \delta_o \cdot w_{ho} \]
The rule for $\delta_o$ and $\delta_h$ result in a recursive way to compute the $\delta$’s for all units in the network (excluding the input units). These values are then used to compute the weight changes for each connection using the weight update rule shown above. In this project, the sigmoid function is most commonly used as the activation function for all of the processing units. Below are the equations for $\delta$ in this case:

$$\delta_o = (d_o - y_o) y_o (1 - y_o)$$

is the equation for output units, and the equation for hidden units is

$$\delta_h = y_h (1 - y_h) \sum_{o=1}^{N_o} \delta_o w_{ho}$$

These are the equations that are used to compute the error factor for all units in neural networks which employ the sigmoid activation function. They are then plugged into the weight update rule to actually adjust the weights for each connection.

There are still a couple of items relating to the weight update rule that need to be addressed. First is the learning rate, represented by $\gamma$ in the equation. True gradient descent requires infinitely small adjustments in the weights so that the target value is not over shot, then possibly overshot again moving in the opposite direction (a phenomenon known as oscillation) [19]. The learning rate is simply a way to adjust the rate at which weights are adjusted to avoid oscillation, as well as a learning process that is too slow. An ideal learning rate depends on the specifics of the network and the problem being addressed, but the goal is to choose one that is as large as possible without causing oscillation.

The second item that needs to be addressed is the momentum factor. The role of momentum is to help escape local minima. It is an optional part of the equation and is
not shown in the version above. Below the momentum factor is represented by \( a \) in a form of the weight update rule with momentum factored in:

\[
\Delta w_{jk}(t+1) = \gamma \delta_k y_j + a \Delta w_{jk}(t)
\]

The reason weights are adjusted is to minimize the error function by following the graph curve down to a minimum through gradient descent. The problem is that the minimum that is found might only be a local minimum. Adding the momentum factor is thought to help find the global minimum [14]. However, it is not guaranteed to do so. Basically a small fraction of the previous adjustment for a given weight to the next adjustment, which gives a little “push” to the process in order to help it overshoot the local minima.

Training Alternatives

An alternative to back-propagation and gradient descent that will be mentioned briefly here is simulated annealing. Simulated annealing was inspired by the metallurgic process of annealing, which it attempts to emulate. It is a process that, like gradient descent, attempts to find the global minima for a given error function [23]. The algorithm works as follows:

1. Set \( C := \) some arbitrary value
2. Do while true:
   a. For \( TI \) iterations, repeat:
      i. Randomize error function inputs \( I \) (in this case, network weights) over range \( R \)
      ii. If \( I \) represents a better solution than \( C \), set \( C := I \).
b. Decrease range $R$ by $M$

c. If $R \leq$ some lower bound $B$, then exit loop

3. Loop

4. Done

Values for $T1$, $M$, and $B$ can be adjusted and refined according to the particular context in which the network is used. Essentially this algorithm directs the range over which the weights of the network are randomized towards an optimal solution [23]. It progressively reduces the size and location of this range, with the goal of converging it to the global minima.

Support Vector Machines

Support Vector Machines (SVMs) are a member of a class of learning algorithms known as Kernel Methods (KMs) [24]. SVMs are in fact the best known member of this class. KMs are well suited for pattern analysis, the goal of which is to find and study various types of data relations (e.g., clusters, rankings, correlations, and classifications). Support vector machines close to their current form first appeared in [6] and were later refined in [7]. In recent years, learning methods utilizing positive definite kernels have increased in popularity, especially in the area of machine learning [25]. These methods have a stronger mathematical slant than many other machine learning methods (such as neural networks) and have consequently generated significant interest in the statistics and mathematics community for these methods [26]. KMs approach the problem of pattern analysis by mapping input data into a high dimensional feature space, where each coordinate corresponds to one feature of the data items. KMs incorporate
mappings done by independent kernels, which utilize kernel functions. These kernel functions enable them to operate in the feature space without ever computing the coordinates of the data in that space, and are what allow KM techniques to model non-linear relationships [26]. The approach is systematic, reproducible, and based on sound statistical learning theory. Training involves the optimization of a convex cost problem: there are no local minima to complicate the learning process [24].

SVMs are very well suited for data mining tasks such as classification, regression, and novelty detection. In real life scenarios, SVMs have been successfully put to use in areas such as:

a. Particle identification
b. Face recognition
c. Text categorization
d. Bioinformatics
e. Database marketing

The power of the SVM approach is rooted in three key concepts: **margins**, **duality**, and as mentioned earlier, **kernels** or **kernel functions** [24]. In simplified terms, an SVM is trained on an example set of binary data (members of which take on one of two values), and then uses a specialized algorithm to build a mathematical model that predicts into which category new examples will fall. An SVM represents data elements as points in space that are divided by a gap that is made to be as wide as possible. In more formal terms, for an \( n \)-dimensional data set, SVMs construct an \( n-1 \) dimensional hyperplane that separates the data points of one class from the other in an \( n \)-dimensional input space [24].
First, it will be shown how these concepts apply to simple linear classification, and then they will be expanded to more complex tasks.

**Linearly Separable Classification**

The most basic case of classification occurs with a data set that is linearly separable. A linearly separable dataset can be divided by a linear hyperplane. As an example, take a set of data points \( X = \{x_0, x_1, \ldots, x_n\} \), each having a corresponding value of \( y=1 \) or \( y=-1 \). Each of the data elements can be represented by an n-dimensional vector.

The classification function can be written as: \( f(x) = \text{sign}(w \cdot x - b) \), where \( w \) is the normal vector to the discriminant plane, and \( b \) is the offset of the plane from the origin.

Figure 8 shows two possible choices for the hyperplane. There are an infinite number of \( n-1 \) dimensional hyperplanes that can separate the data; the goal is to find the hyperplane that represents the largest separation between the two classes of data. In Figure 8, the solid line represents the plane that achieves this separation and, in simple

![Fig. 8. Multiple discriminant planes.](image)

words, can be described as the plane that is farthest from both classes. There are multiple approaches by which this plane can be derived:

1. **Convex hull method**: The convex hull of a class of data points is the smallest subset of those points that contains the entire set (geometrically speaking) [24]. Figure 9 shows these convex hulls. The next step is to determine the two closest points on the hulls (indicated by points (c) and (d) in Figure 9). If a plane is constructed that bisects these two points (whose normal vector \( \mathbf{w} = \mathbf{c} - \mathbf{d} \)), it will be a maximal margin plane between the two classes. These points can be found by solving the following quadratic problem (QP) [24]:

\[
\min_{\mathbf{a}} \frac{1}{2} \| \mathbf{c} - \mathbf{d} \|^2 \quad \text{where} \quad \mathbf{c} = \sum_{\gamma_i \in \text{class 1}} \mathbf{a}_i \mathbf{x}_i \quad \text{and} \quad \mathbf{d} = \sum_{\gamma_i \in \text{class -1}} \mathbf{a}_i \mathbf{x}_i \quad \text{satisfy} \quad \sum_{\gamma_i \in \text{class 1}} \mathbf{a}_i = 1 \quad \text{and} \quad \sum_{\gamma_i \in \text{class -1}} \mathbf{a}_i = 1
\]

\[
\text{for} \quad a_i \geq 0 \quad \text{and} \quad i = 1, \ldots, n
\]

where \( a_i \) is a non-negative Lagrange Multiplier.
2. **Margin maximization:** This alternative to the convex hull method involves finding a separating hyperplane that is “furthest” from both data classes by maximizing the margin between two parallel supporting hyperplanes [24]. A separating hyperplane will be valid for a given class if all the data points for that class lie on one side of the plane. As stated earlier, this hyperplane can be represented by the form $w \cdot x - b = 0$. The equations for the two parallel supporting hyperplanes for which the goal is to maximize the margin would be $w \cdot x - b = 1$ and $w \cdot x - b = -1$ respectively. Together these can be written as $y_i(w \cdot x_i - b) \geq 1$. To find the separating hyperplane that is “furthest” from both data sets, these support planes are “pushed apart” until they “bump” into the support vectors for their respective classes. The support vectors (represented by dashed lines in Figure 10) are formed by the first point(s) reached by the expanding planes (circled in Figure 10) [24].

![Fig. 10. Maximizing margin.](image)

Geometry can be used to find the distance between these planes:

\[
\text{Distance} = \frac{2}{\|w\|} \quad \text{therefore goal is to minimize} \quad \frac{\|w\|}{2}
\]

This yields the following quadratic problem [24]:

\[
\min_{w,b} \quad \frac{1}{2} \|w\|^2 \quad \text{such that} \quad \begin{align*}
  w \cdot x &\geq b + 1 & \text{if } y_i \in \text{class 1} \\
  w \cdot x &\geq b - 1 & \text{if } y_i \in \text{class } -1
\end{align*}
\]

The constraints here can be simplified to \(y_i(w \cdot x_i - b) \geq 1\).

The solutions produced by both methods are identical. Both solutions employ support vectors: in the convex hull method the support vectors are determined by the closest points; in the maximum margin method they determine the boundaries of the expanding supporting hyperplanes. This is an example of the mathematical concept of \textbf{duality}. The maximal margin solution can be rewritten in Lagrangian dual form:

\[
\min_a \quad \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j a_i a_j x_i \cdot x_j - \sum_{i=1}^{n} a_i \quad \text{such that} \quad \sum_{i=1}^{n} a_i y_i = 0 \quad \text{for } a_i \geq 0 \quad \text{and} \quad i = 1,...,n
\]

This is equivalent to scaling the closest points in the convex hull solution. When solved, they both yield the same normal to the separating plane [24]:

\[
w = \sum_{i=1}^{n} y_i a_i x_i \quad \text{for } a_i > 0
\]

and threshold \(b\) determined by the support vectors. Theoretically both problems are well understood and many solution algorithms exist for each. In practice, the latter method of maximum margins is preferable since it has very simple constraints and is simpler when it comes to adding non-linear extensions.

The mathematics behind the approaches shown above is well founded. The statistical foundations of these methods have proven that there are bounds on the
generalization error [24]. In other words, there are mathematical limits to the error rate that will result when new examples are fed into the SVM. For linear functions, maximizing the margin between the supporting hyperplanes reduces the function complexity. It follows that when the margin is maximized, the bounds on generalization error are minimized. The dimensionality of the data is not a factor of this attribute so this concept applies to data with a large number of dimensions/attributes, and problems caused by over fitting high-dimensional are significantly reduced.

Linearly Inseparable Classification

For cases where the data are not linearly separable, the methods shown above will fail (Figure 11 shows this type of situation). However, with certain modifications and/or relaxations on constraints, they can be adapted to the linearly inseparable case.

These modifications are outlined below.

Fig. 11. Linearly inseparable data.

1. **Convex hull:** If the data are not linearly separable, the convex hulls of the two classes will intersect. In this case, *reduced convex hulls* are the solution. Reduced convex hulls are constructed by assigning an influence to each point and restricting that influence to some upper bound \( D < 1 \). Mathematically this can be expressed as [24]:

\[
\begin{align*}
  d &= \sum_{y_i \in \text{class 1}} a_i x_i, \\
  \text{where} \\
  \sum_{y_i \in \text{class 1}} a_i &= 1 \\
  \text{and} \\
  D &\geq a_i \geq 0
\end{align*}
\]

If \( D \) is made to be adequately small, the reduced convex hulls comprised of values for \( d \) will not overlap. To find the two closest points on the reduced hulls, we again must incorporate \( D \):

\[
\min_w \frac{1}{2} \left\| \sum_{y_i \in \text{class 1}} a_i x_i - \sum_{y_i \in \text{class -1}} a_i x_i \right\|^2 \\
\text{such that} \\
\sum_{y_i \in \text{class 1}} a_i = 1 \\
\sum_{y_i \in \text{class -1}} a_i = 1
\]

for \( D \geq a_i \geq 0 \)

2. **Margin maximization:** The standard margin maximization method with also fail if the data are not linearly separable. Here again the constraints of the original approach must be relaxed to ensure that each data point is on the appropriate side of the separating hyperplane. Data points that do not satisfy this criterion are considered errors. The goal is to maximize the margin while minimizing the error. This is achieved by adding a “slack” variable \( z_i \) to the problem [24]:

\[
\min_{w,b,z} \frac{1}{2} \|w\|^2 + \frac{1}{C} \sum_{i=1}^{n} z_i \\
\text{such that} \\
y_i (w \cdot x_i - b) + z_i \geq 1
\]

for \( z_i \geq 0 \) and \( i = 1, \ldots, n \)

Again the maximum margin solution can be written in Lagrangian dual form:

\[
\min_w \frac{1}{2} \left( \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j a_i a_j x_i \cdot x_j - \sum_{i=1}^{n} a_i \right) \\
\text{such that} \\
\sum_{i=1}^{n} a_i y_i = 0 \\
\text{for} \ C \geq a_i \geq 0 \quad \text{and} \quad i = 1, \ldots, n
\]
Again this is equivalent to the convex hull solution. The solution shown above is the most commonly used formulation for this classification [24].

**Non-linear Classification Through Kernels**

For some data sets, even the relaxed constraints outlined in section B will fail (an example of this is shown in Figure 12). In this case, a quadratic function is needed.

![Fig. 12. Linearly inseparable data requiring quadratic.](image)

The standard method for converting a linear classification algorithm to a non-linear function is to extend the data by means of non-linear transformations [24]. In the extended feature space, linear classification algorithms can be applied. These algorithms translate to non-linear functions in the original input space. For example: given a two dimensional input space with feature set \([r, s]\), a quadratic discriminant can be constructed by mapping the original two dimensional input space to the five dimensional input space \([r, s, rs, r^2, s^2]\). In the new input space, a linear discriminant can be constructed that translates to a non-linear discriminant in the original two dimensional input space (Figure 13).
Features: \([r, s]\)  
\[w \cdot x = w_1r + w_2s\]

Features: \([r, s, rs, r^2, s^2]\)  
\[w \cdot \theta(x) = w_1r + w_2s + w_3rs + w_4r^2 + w_5s^2\]

<table>
<thead>
<tr>
<th>2-dimensional space</th>
<th>Extended 5-dimensional space</th>
</tr>
</thead>
</table>

Fig. 13. Mapping the original two dimensional input space to the five dimensional input space.

The resulting classification function is:  
\[f(x) = \text{sign}(w \cdot \theta(x) - b), \text{ or } \text{sign}(w_1r + w_2s + w_3rs + w_4r^2 + w_5s^2 - b)\].

Data sets with a large number of dimensions bring about two potential problems [24]. The first is that over-fitting can become an issue. SVMs typically mitigate this problem since they operate on margin maximization and therefore function-complexity reduction. The second problem is that it is not practical to compute \(\theta(x)\). SVMs get around this problem through the use of kernels. A kernel, in this context, is a specialized mathematical function that replaces the dot product in the original Lagrangian dual form. The trick is to choose a kernel that best suits the problem domain. All the benefits of the original SVM problem solving techniques are preserved.

Self Organizing Maps

Self Organizing Maps (SOMs) are a special kind of neural network that are trained using unsupervised learning to produce low dimensional (typically between one and three dimensions) views of the input space. This makes them useful for producing low dimensional views of high dimensional data. SOMs are different from other artificial neural networks in that they use a neighborhood function to preserve the topological properties of the input space [27]. Areas of computer science where SOMs have been
applied include: pattern recognition, robotics, process control, and processing of semantic information [8].

The architectures employed by neural networks in general can roughly be divided into three categories: feed forward, feedback, and competitive or unsupervised [8]. Networks in the feed forward category simply transform their input into output through a process that is determined by external, supervised adjustment of system parameters. With feedback networks, the original input defines the initial state of the system. The final state of the system is determined by cycling the inputs through the state changes and arithmetic transformations of the system until an asymptotic state is reached and the final output is determined. With networks in the cooperative/unsupervised category, neighboring nodes in the network derive the output through mutual interaction and adaptively transform into specialized detectors of specific signal patterns. SOMs belong to this last category [8].

SOMs operate in two modes like most other neural networks: training and mapping. The training process of an SOM builds the map from input examples through a competitive process known as vector quantization. The resulting map is usually of one, two, or three dimensions and consists of neurons or nodes. Associated with each node is a vector of weights of the same dimension as the input. Typically, the nodes are arranged in either a rectangular or hexagonal grid. When a new piece of input is fed into the system, its corresponding position in the map is determined by finding the node with the closest weight vector to that piece of input [8]. The locations of the responses tend to become ordered in a manner that is consistent with having a coordinate system placed over the network. Due to the mutual interaction between neighboring nodes, their spatial location
coordinates correspond to a particular domain of input signal patterns. The goal of the training process is to cause different physical locations of the network to respond similarly to certain types of input; the motivation behind this organization is based on how sensory information is handled in separate parts of the human brain [28]. The end result is a compressed input space with its topological and metric relationships preserved [29]. Figure 14 and Figure 15 show two different potential layouts for SOMs.

Fig. 14. A rectangular SOM.

Training Process

Before the training process begins, the weights of the nodes are initialized to one of two kinds of values [8]:

a. Small random values.

b. Values sampled from the input space spanned by the two largest component eigenvectors.

The second option often results in faster learning because the initial weights already represent a good approximation of the ultimate SOM.
As mentioned previously, the training process of an SOM is a competitive one. When an input data point is fed to the system, the Best Matching Unit (BMU) must be determined. To find the BMU, the input point’s Euclidian distance to all weight vectors in the network is calculated. The network node with a weight vector most similar to the input is chosen as the BMU. The weights of the BMU and the nodes physically near it are adjusted to be close to the input vector. The magnitude of this adjustment decreases with distance from the BMU, as well as with time. The training algorithm works as follows [29]:

a. Choose an input vector $x$

b. From the set of nodes in the map, $m$, find node $i \in m$, $m = \{0, 1, \ldots, k\}$ to satisfy:

$$|w_i - x| \leq |w_k - x| \quad \forall k$$

(1)

c. The update rule for a node $k$ given BMU $i$ is:

$$w_k(\text{new}) = w_k(\text{old}) + a \times N(i, k) \times (x-w_k)$$

(2)

where $N$ is the neighborhood function and $a$ ($0 < a < 1$) is the learning rate factor. The learning rate factor decreases monotonically with each iteration. The neighborhood
function acts like a smoothing kernel and depends on both time and the outcome of (1). It is a decreasing function of the lattice distance between nodes \( i \) and \( k \). A common neighborhood function is the Gaussian function:

\[
N(i,k) = \left( \frac{||r_i - r_k||^2}{2\sigma^2} \right)
\]

where \( r_i \) and \( r_k \) correspond to locations on the map grid, and \( \sigma \) corresponds to the width of the function, which like \( a \), decreases monotonically [29].

Regardless of the type of function it may be, the neighborhood function shrinks with time. It takes place on a global scale near the beginning of the training process and shrinks to just a couple of nodes by the time the node weights are converging to their final values.

The process is repeated for each input vector for a specified number of iterations. The above process is a description of the incremental approach. A batch version of the algorithm is an option, and often runs much faster than the incremental version. The batch version operates as follows [29]:

a. For each node \( i \) in the map, create a list of input vectors that have BMUs that fall within the neighbor set for that node.

b. For each map node, take the mean of that node’s respective list and apply it to that node.

c. Repeat the above steps for a specified number of iterations.

N-gram Analysis

N-grams are used to analyze large sequences of items by breaking them down into smaller manageable subsequences. N-grams can be used to describe objects as
vectors, which makes it possible to apply mathematical and statistical functions. This is due to the fact that vectors have well defined and studied mathematical attributes, whereas general objects don’t. This is rooted in Claude Shannon’s work in information theory [9]. The items in question can be a large number of things, including syllables, letters, words, and numbers [30]. N-grams are used in various areas mathematics and statistics [31]:

a. Natural language processing
b. Genetic sequence analysis
c. Speech recognition
d. Information retrieval

Given a token alphabet \( A \), where \(|A|\) is the number of tokens in the alphabet and \( n > 0 \), an n-gram is defined as a subsequence of \( n \) tokens. N-grams are often continuous but can be non-continuous subsequences. By determining the frequency at which n-grams appear in a sequence of tokens, an n-gram frequency vector representation of that sequence can be obtained [31]. An n-gram of size 1 is referred to as a “unigram”; an n-gram of size 2 is called a “bigram”; size 3 is a “trigram”; and beyond size 3 is just “n-gram” [30].

As an example, take the series of English words “her and her.” The token alphabet derived from this sentence would be:

\[
A = \{"her", "and ", "her", " \}\}
\]

with the last token being the space character. The sentence produces the following n-gram frequencies:
One of the requirements for deriving a token alphabet is that the object to be vectorized must be describable as a sequence of tokens. Because one of the more common uses for n-grams is in the arena of document analysis using the Latin alphabet [31], often the token alphabet is simply assumed to be the Latin alphabet. For the sentence shown above, the Latin alphabet would produce the following n-gram frequencies:

**Unigrams:**
- "h":2, "e":2, "r":2, " ":2, "a":1, "n":1, "d":1

**Bigrams:**
- "he":2, "er":2, "r":1, "a":1, "an":1, "nd":1, "d":1, "h":1

**Trigrams:**
- "her":2, "er":1, "r":1, "an":1, "and":1, "nd":1, "d":1, "h":1
- "d":1, "he":1

**4-grams:**
- "her":1, "er":1, "r":1, "an":1, "and":1, "nd":1, "d":1, "h":1

N-gram models are a type of probabilistic model for predicting the next item in a given sequence of tokens. In mathematical terms, an n-gram model predicts token $x_i$ based on the tokens preceding it ($x_{i-1}, x_{i-2}, \ldots, x_{i-n}$) [32]. In terms of probability, this is equivalent to $P(x_i | x_{i-1}, x_{i-2}, \ldots, x_{i-n})$. In the arena of language modeling, this means each word is only dependent on the last $n$ words.

In addition to language processing, n-grams can be used for approximate matching. Convert a given sequence to a set of n-grams allows vector functions to be
performed in an efficient manner. Taking the English alphabet again as an example: choosing an n-gram length of three produces a $26^3$-dimensional space. This representation results in a loss of information about the string, but if two strings have a similar vector representation (as measured by cosine distance), then they are likely to be similar. Other metrics, like z-scores have also been applied to matching with n-grams. Z-scores work by examining how many standard deviations each n-gram differs from its mean occurrence.

N-gram models have been criticized because they do not in any way represent long-range dependency relationships. As stated earlier, a given token depends on only the previous $n-1$ tokens. Languages often have cases of unbounded dependencies, which means in some cases n-gram models cannot adequately represent them.

Past/Current Application of Techniques to This Problem Domain

Decision Trees

In [33], pure decision-tree based techniques were tested on Win32 executables. The focus was not on Win32 API calls, but on sequences of processor instructions. A static analysis approach was used to extract op-code sequences from the executables and derive the features used for training and testing. The benign executables were obtained from a clean installation of Windows XP and the malicious executables were obtained from VX Heavens [34], the same source as was used for this project. The initial feature set was pruned by eliminating features that only belonged to one class (either benign or malicious but not both), and by eliminating features that appear less than 10% of the time. Basic decision trees were tested, along with enhanced classification
techniques such as Bagging and Random Forest. In the tests, the basic decision trees yielded a detection rate of 93.4% with a “false alarm” rate of 13.4%, which is relatively high. The enhanced methods of Bagging and Random Forest showed much improvement over basic decision trees: Bagging produced a 94.3% detection rate with a false alarm rate of 6.7% percent and Random Forest produced a 95.6% detection rate with a false alarm rate of only 3.8%.

Bayes’ Classification

In work done by researchers from SUNY Stony Brook and Columbia [35], Naïve Bayes classifiers (among other techniques) were used to classify executable code. The executables that were tested were either MS-DOS or Windows executables. The data set consisted of 3,265 malicious executables and 1001 benign executables. The benign executables came from a clean install of Windows 98 and the malicious executables, composed primarily of viruses, came from various FTP sounds around the Internet. Rules for the classifiers were derived by examining plain text strings within each executable in the training set. Essentially all executable code (both op-codes and Win32 API calls) was ignored. Rules took the format \( P(F|C) \) where \( F \) was a given string and \( C \) was the class of executable (benign or malicious). An instance of this rule indicated the likelihood \( F \) was present in an executable belonging to class \( C \). Classifiers were implemented in multiple instances on different portions of the training data to form a multi-classifier mechanism. For each classifier, the derived rule set was ultimately different because the training data was different. Combining these classifiers to classify previously unseen executables resulted in a detection rate of 97.76% with a false positive rate of 6.01%.
Neural Networks

Neural networks have been researched as both stand-alone anti-malware mechanisms and as parts of multi-technique combinations. In [36], researchers at IBM implemented neural networks in the detection of boot sector viruses. The code found in boot sectors is not tied to any specific operating system or development system so these tests were not connected in any way to Win32 executables. The feature set was comprised of trigrams, which are short byte sequences that frequently appear in infected boot sectors, but not clean ones. Using this approach, the researchers were able to detect 80-85% of previously unknown boot sector viruses with a false positive rate of less than 1%.

In [37], researchers at IBM implemented an approach that combined both neural networks and N-gram analysis. The primary goal of the project was to reduce or possibly even eliminate false positives that inevitably arise when using only individual classifiers to detect malicious code. The researchers combined individual classifiers using a voting system, and were successful in reducing the risk of false positives to an arbitrarily low level with only a slight increase in the false negative rate. Win32 executables were the test subject, and a huge number of benign samples were incorporated (over 50,000), but only a small number (undisclosed) of viruses were also included. From these executables, several distinct and separate input sets were generated; all based on N-grams derived from the executables’ binary code. Three and four length N-grams, along with complete and incomplete coverage versions were some of the sets generated and tested. A total of eight neural networks were used for training against these sets. During testing, examples from the test set were fed through all eight networks and a
voting system was used to determine the final results. The final results indicated that overall the approach was not sufficient as a solo method of detecting viruses, but could be used to augment other heuristic based scanners.

Support Vector Machines

Support Vector Machines have been applied to the task of detecting malicious code in multiple instances. In [38], SVMs were utilized in a manner that was similar to the approach employed by this project. The focus was on Win32 executables and the feature set was made up of Win32 API calls. In addition, the viruses on which the SVMs were tested were obtained from VX Heavens. Full Win32 API call logs were created from running executables through a monitoring program called APISPY [39]. This application was considered for this project but found to be inadequate. The feature set was extracted from these logs by first assigning an index to each Win32 function call and sliding a small window over the index set produced from a given log. The window was small, about 6 or 7 indexes in length, and the indexes visible in the window were grouped to form a single feature. Sliding the window across the full length of the call log for an executable created the feature list for that executable. Other techniques were employed to remove abnormal or anomalous data before training began. A total of 532 benign and malicious executables were used for training, and 100 additional were used for testing. The experiment achieved false positive rates as low as 5.66% and false negative rates as low as 3.21%.

SVMs have also been researched in tandem with other AI techniques for detecting malware. In [40], SVMs were utilized along with neural networks using N-grams as input for classification. The results were combined using Dempster-Shafer
theory of evidence [41] to make the final decision. Like [38], the work focused on Win32 executables, with the malicious code being obtained from VX Heavens. Unlike [38] the input features were derived from the binary execution code, and not Win32 API calls. The results showed that the combined method was slightly more effective than the individual methods: the combined method showed a false positive rate of about 7% at a true positive rate of 90% while the individual SVM showed a false positive rate of about 13% at a true positive rate of 90% and the neural network showed a false positive rate of about 16% at a true positive rate of 90%.

**Self Organizing Maps**

In work done by InSeon Yoo [42], SOMs were trained to recognize and visualize Win32 viruses. The research focused on viruses that attach themselves to legitimate executables. SOMs were used to locate areas of the benign executables that were infected, and visualize how the executable as a whole changed. The SOMs focused on processor op-codes, not Win32 API calls. The results showed that variants of known viruses could be detected without using signatures and instead by detecting patterns that exist throughout entire families of viruses. Detection of obfuscated members of known virus families was shown to be possible. The study showed that families of viruses have “virus masks” which are somewhat like DNA signatures.

**N-gram Analysis**

N-grams have been used in tandem with other techniques in the detection of malicious code as well as a standalone technique. In the multi-technique realm, researchers at IBM used N-grams to break down Win32 executables, and neural networks to classify them as being malicious or benign [37]. Using N-gram analysis, op-code based
features were extracted from both benign and malicious programs, and then fed through neural networks to train the networks. Results indicated that it was not sufficient as a solo method of detecting viruses, but could be used to augment other heuristic based scanners.

In “Detection of New Malicious Code Using N-grams Signatures” [43], n-grams were used as a stand-alone malware detection mechanism. In conjunction with a database that contained known malicious and benign code, N-gram analysis was used to extract the most frequent N-grams, which acted as signatures. When new code was analyzed, it was classified based on the category it fit best. CNG (Common N-gram) classification was used to achieve this.
CHAPTER III

SOLUTION IMPLEMENTATION

Overview

As discussed in Chapter I, the objective of this project was to investigate the effectiveness of the machine learning techniques discussed in Chapter II at identifying malicious code in controlled environment by restricting the executable platform to Win32 and restricting input to an executable’s Win32 API call profile. For the basis of this project, “effectiveness” and “performance” are general terms meant to indicate the overall accuracy of a particular technique. They take into account false positive and false negative rates, as well as the average error rates of the outputs from these machine learning mechanisms.

The approach taken in this project differed from previous related works in two distinct ways. First, it examines a larger set of machine learning techniques than was encountered in any single study. Second, it uses an input format that was not encountered anywhere in the research.

The set of malicious code used in this project was obtained from VX heavens [34]. This site offered a plethora of different types of malware, but the subset that was utilized was the virus subset containing Win32 executables. The set of benign code was
derived by researching popular software suites and packages in use today and incorporating parts or all of them.

Using a custom Win32 API monitoring application developed specifically for this project, each executable was run inside a VMWare Workstation 6.0 virtual machine [44] running the latest version of 32-bit Windows XP. Running each executable produced a profile of its Win32 API calls, which was stored in a database. From the database, the log data could be easily transformed into whatever input format each technique required.

For each technique, one of two routes was taken:

1. A custom algorithm was developed to execute the technique.
2. 3rd party software was used to execute the technique.

The respective route taken for each technique was executed against the call log data and an analysis of effectiveness was performed.

Solution Design

The purpose of the executable analysis phase was to trace the Win32 API functions called by each executable during their execution. A dynamic analysis approach was required over a static one not only to track each instance of a call to a given function, but also to circumvent the obfuscation techniques malicious executables sometimes employ to hide the functions they call. In order to accomplish this task a mechanism was needed to monitor each program during execution and log the desired information. Multiple software solutions currently exist and are available for this exact purpose. They were each given a test run and, unfortunately, none of them were adequate for the needs of this project. Many tested simply missed some of the Win32 API calls made by Win32
executables. This shortcoming was verified by developing custom programs with known Win32 function call profiles and monitoring them with said software solutions. Others did not track the right kind of calls, or did not produce a useable form of output.

The result of these shortcomings was the need to develop a custom solution to monitor program interaction with the Win32 API. The final solution was developed in C++ using Microsoft Visual Studio and incorporated the Detours library from Microsoft. The Detours library facilitates the interception of Win32 API calls dynamically at runtime [45]. It works by replacing the first few instructions of the target function with an unconditional jump to a user provided detour function. The instructions in the target function that are replaced are preserved in a trampoline function. The trampoline function also contains a branch back to the remainder of the target function. The idea is that the detour function can perform any pre-processing that may be necessary, call the trampoline function, and then perform any post-processing after it finishes [45].

Figure 16 displays the program flow of a call to an unmodified API function (A), and that of a call to a patched API function (B). Figure 17 is a depiction of how the disassembled code would appear before and after the modification.

The custom monitoring software injected custom code into running programs that would log calls to each function it was configured to track. It was configurable as to which Win32 libraries would be tracked and which functions within those libraries would be tracked. The Win32 API as a whole is made up of hundreds of separate libraries; however the core of the API is contained within a much smaller subset. The members of this subset were determined by debugging sample executables from the benign and malicious sets and observing the libraries that were loaded into their respective address
Fig. 16. Dynamic API call interception.


Fig. 17. Executable code transformation.

spaces by default and during execution. Figure 18 lists the libraries that were tracked for this project.

![Tracked Win32 libraries](image)

Fig. 18. Tracked Win32 libraries.

The analysis phase required a significant amount of time, due to multiple factors:

1. The unpredictable nature of each virus meant that conceivably anything could happen to the analysis virtual machine, including data corruption, system restarts, and exceedingly long running times. At the end of each virus run, the virtual machine had to be reset to its original state to erase any effects of that virus.

2. When analyzing benign executables, care was taken to sufficiently test a majority of a given executable’s functionality. This was necessary in order to ensure an accurate Win32 profile representation of that program.

Input Features

Each line in the logs produced by the execution monitoring software developed for this was formatted as follows:

<library name>:<function name> <call count (hexadecimal format)>

The choice was made not to log each unique call to a Win32 function, but to instead only log the total number of times a function was called. This format meant that
function call order information was essentially eliminated. This choice was due to multiple factors:

1. Logging each call individually would yield an extremely large call log for each executable. Several Win32 functions contained call counts in the seven to eight figure range. This would produce log files on the order of gigabytes in size for some programs. The large size would lead to storage space issues, as well as significantly increased processing times for both training and testing.

2. The reduced size of the chosen format allowed the chosen AI techniques to accept the entire resulting data set from a given log as a single piece of input, whereas preserving call order information would have required the partitioning of results from a single log into multiple pieces of input.

3. The chosen format was unique in that it was not encountered anywhere in the research, and therefore had the potential to provide a new perspective on the application of AI techniques to malicious code detection. Most of the examined studies did not used Win32 API calls as input and the studies that did preserved call ordering information.

Figure 19 is a sample excerpt from the log created by monitoring the Win32.HLLP.Semisoft.a virus.

Analysis Data

A total of 550 malicious and 550 benign executables were analyzed and logged. The virus set from VX heavens contained well over 1000 viruses, but only 550 ran to completion without crashing and produced logs with substantial number of Win32 API calls. For each of the malicious and benign sets of executables, 50 logs were
randomly chosen for testing purposes and the remaining logs were put towards training. The result was a total of 1000 logs in the training set and 100 in the testing set. The option to create an additional set for cross-validation was skipped due the relatively small number of executables in the training set when compared to the size of the feature set. Creating a cross-validation set would reduce this number further.

While the monitoring software was configured to track 20 different Win32 libraries, function calls from only 13 of them were actually encountered during analysis. The number of functions called at least once in the analysis phase is displayed below, grouped by library name. Figure 20 depicts this information visually.

Three distinct data sets were derived from the function call logs:

1. **Data set A**: The raw, unaltered results from the call logs.

2. **Data set B**: This set was transformed so that the function call count range was normalized to a range of 0 – 2000. The resulting value for each call count was then increased by 1000 to give functions with low call counts some significance. This greatly decreased the significance of the call count for a function, and increased the significance that it was simply called.
The median call count per executable in the raw data was found to be 242. This was the primary number used in transformation from the raw data to its form in set B. The median was doubled to find the new, statistically adjusted upper bound. This upper bound was then doubled twice, just to give a little more significance to those functions called a huge number of times. If a function had a call count larger than this new upper bound, it was simply capped at the boundary. The total number of entries in all function call logs was 114706. The number of entries with call counts naturally under the 2000 ceiling was 106982. This meant that 93.2% of the entries had call counts that were naturally within the 2000 upper boundary prior to normalization. The goal of this transformation was to prevent the extremely large call counts that a few functions incurred from overwhelming the rest of the data.

Fig. 20. Number of Win32 functions encountered during run-time analysis.
3. **Data set C**: For this set, the significance of the function call count was completely eliminated. A positive constant value was given for the existence of a function for a given executable, and zero otherwise.

Each AI technique was implemented to produce one of two predefined output values. These values corresponded to the two classes of executable: benign and malicious. For testing, these values were defined as \{B, M\} (Benign and Malicious).

**Application and Results**

**Decision Trees**

The software used for constructing and testing decision trees was C4.5 R8, an implementation of C4.5 written by J. R. Quinlan [46]. This software offered five customization options for building the trees:

- Use of probabilistic thresholds for determining splits in continuous attributes
- Iterative mode training with specified number of trials
- Use of GAIN criterion instead of GAIN ratio (default uses GAIN ratio)
- Pruning confidence level (default 25%)
- Minimum number of objects in all tests (In all tests, at least two branches must contain a minimum number of objects. The default is 2, but this option modifies that number).

**Training process.** C4.5 R8 was well suited to this particular problem domain. The training process for each test was very fast, requiring literally seconds to complete both the testing and training stages. C4.5 R8 accepted both ordinal and continuous values and was ready to accept the data sets generated for this project as they were.
**Variables.** All five parameters listed above were experimented with on a per test basis. The goal was to adjust these values in a manner that allowed for the determination of which had the most significant impact on performance, as well as the optimal values for each.

**Inputs.** The dimensionality of the data set for the decision trees was 2381, corresponding to the number of unique Win32 functions logged during the executable analysis phase.

- **Data Set A:** The value for each feature was the raw call count for the corresponding function. The class to which the data element belonged was either “benign” or “malicious.”

- **Data Set B:** Same as A, except the inputs were the adjusted call counts from data set B.

- **Data Set C:** Same as A and B, except the value for the features became restricted to one of two values: 0 for a function that was never called for a given executable; and 1000 for a function that was called one or more times.

**Output.** For this series of tests output values B and M were defined to be the same as the class designations (“benign” or “malicious,” respectively). All decision trees were trained to produce output $O$ such that $O \in \{B,M\}$. The decision trees were, by design, binary classifiers and produced output that was exactly B or M.

**Results.** Since the decision trees were precise binary classifiers, one approach was used in determining accuracy: the number of elements classified correctly versus the number classified incorrectly.
1. **Data Set A.** Data set A was the worst performer of all three sets. False positives reached a low of 5% as did false negatives, but overall accuracy peaked at 90%. Table 3 shows the full results.

<table>
<thead>
<tr>
<th>Test</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
<th>Test 6</th>
<th>Test 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilistic thresholds:</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>No</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Iterative mode:</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>No</td>
<td>no</td>
<td>Yes: 5,10,20</td>
<td>no</td>
</tr>
<tr>
<td>GAIN criterion:</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Pruning confidence:</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%-100%</td>
<td>25%</td>
</tr>
<tr>
<td>Minimum test objects:</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3,4,5</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>7,6,7</td>
<td>5</td>
<td>8</td>
<td>8,7,5</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>7,8,8</td>
<td>5</td>
<td>5</td>
<td>5,8,5</td>
</tr>
<tr>
<td>Overall accuracy %:</td>
<td>83</td>
<td>88</td>
<td>83</td>
<td>86,86,85</td>
<td>90</td>
<td>87</td>
<td>87,85,90</td>
</tr>
</tbody>
</table>

2. **Data Set B.** The results from set B were on average an improvement from set A. However, the overall accuracy rate still peaked at 90%. Table 4 shows the full results.

<table>
<thead>
<tr>
<th>Test</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilistic thresholds:</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Yes: 5,10,20</td>
</tr>
<tr>
<td>Iterative mode:</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Yes</td>
<td>no</td>
</tr>
<tr>
<td>GAIN criterion:</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Pruning confidence:</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>Minimum test objects:</td>
<td>2</td>
<td>2</td>
<td>3,4,5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>8</td>
<td>6</td>
<td>5,6,6</td>
<td>5,3,6</td>
<td>5</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>5</td>
<td>6</td>
<td>5,6,6</td>
<td>7,9,5</td>
<td>5</td>
</tr>
<tr>
<td>Overall accuracy %:</td>
<td>87</td>
<td>88</td>
<td>90,88,88</td>
<td>88,88,89</td>
<td>90</td>
</tr>
</tbody>
</table>

3. **Data Set C.** Data set C showcased a continued improvement over both sets A and B. Overall accuracy reached a new high of 92%. Table 5 shows the full results.
### TABLE 5
DECISION TREE RESULTS, SET C

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
<th>Test 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilistic thresholds:</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Iterative mode:</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Yes: 5,10,20</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>GAIN criterion:</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Pruning confidence:</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>Minimum test objects:</td>
<td>2</td>
<td>2</td>
<td>3,4,5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>6</td>
<td>4</td>
<td>4,5,4</td>
<td>7,7,7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>6</td>
<td>5</td>
<td>5,5,4</td>
<td>7,7,7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Overall accuracy %:</td>
<td>88</td>
<td>91</td>
<td>91,90,92</td>
<td>86,86,85</td>
<td>92</td>
<td>92</td>
</tr>
</tbody>
</table>

Results analysis. The respective differences between the results from sets A, B, and C seem to indicate that the function call counts were a significant factor in accuracy. While the overall accuracy from set A, which showcased the raw call counts, reached as high as 90%, it also had the lowest overall accuracy of any test at 83%. The change from set A to B did not produce a new high in overall accuracy, but the overall average jumped from 86.3% to 88.4%. The results from set C, which completely eliminated the influence of call counts, showcased the best overall accuracy of 92%, and an average of 89.3%. Essentially, involving the function counts only served to reduce performance.

Other than the data set used to derive the decision tree, the variables that had significant positive effects on overall performance were using GAIN criterion over the GAIN ratio criterion and adjusting the minimum number of test objects. Using basic GAIN instead of the GAIN ratio consistently improved the performance of the decision trees in all data sets. For each data set, the only variable changed between the first two tests was the use of GAIN criterion over GAIN ratio criterion, and the result was an
increase in performance in each instance. This was not expected, since the use of the GAIN ratio over basic GAIN is supposed to overcome the drawback commonly associated with basic GAIN (a large number of partitions where each is small but pure) and penalize higher entropy [11]. Adjusting the minimum number of test objects had a similar effect, although the optimal value was different for data set C (5) than it was for A and B (3). Adjusting the other three variables either had no effect at all (pruning confidence and use of probabilistic thresholds) or a negative effect (use of iterative mode).

Ultimately, the results from set C showed the best performance, including the highest overall accuracy of 92% and the highest average, at 89.3%. The higher level of accuracy can most likely be attributed to the fact that elimination of call counts from the picture removed the necessity to deal with continuous values. The value for each Win32 API function was simply a “yes” or “no,” which means only ordinal values were required in the decision making process. This exclusive use of ordinal values resulted in simpler decision making, which according to the principle of Occam’s razor, is likely to result in a more accurate tree [13]. Although the performance margin of set C over A and B was small, with the use of standard GAIN criterion and a minimum number of tests = 5, set C was still clearly the best performer.

Bayes’ Classification

The software used for constructing and testing Bayes’ classifiers was jBNC [47] a Java based application geared towards Naïve Bayes’ classification and its variants. This software offered four different classifier types as well as five different quality measures:
Classifier Type

- Naïve Bayes (NB)
- Tree augmented naïve Bayes (TAN)
- Forest augmented naïve Bayes (FAN)
- Selective tree augmented naïve Bayes (STAN)
- Selective forest augmented naïve Bayes (SFAN)

Quality Measure

- Heckerman-Geiger-Chickering (HGC)
- Standard Bayesian (SB)
- Local criterion (LC)
- Leave-One-Out cross validation (LOO)
- n-fold t-times cross validation (VC\textsubscript{n,t})

Training process. The training process was very resource and time intensive. The NB, TAN, and FAN classifier types required between 30 minutes and an hour for training in each test. Beyond those types the resource requirements were such that the tests could not even complete. An example: tests with STAN were allowed to run for almost three days before they were simply cancelled. The software itself used almost four gigabytes of RAM.

Variables. The three classifier types mentioned previously (NB, TAN, and FAN) and the first three quality measures (HGC, SB, LC) were incorporated into testing. The goal was to experiment with various combinations of these two parameters to discover which had the most significant impact on performance and which produced
optimal accuracy. Ultimately the testing of STAN and SFAN was not successful due to the huge resource and time requirements. The three quality measures mentioned previously were successfully tested within the NB, TAN, and FAN classifier types.

**Inputs.** The dimensionality of the data set for the Bayes classifiers was 2381, corresponding to the number of unique Win32 functions logged during the executable analysis phase. The one drawback to jBNC was that it did not accept continuous values. This meant data set A could not be used at all, and data set B could only be used if an ordinal set were created to contain each possible value for the call count. This would mean an ordinal set with 2000 values and an exponential increase processing requirements. The decision was made to only use data set C.

**Output.** For this series of tests output values B and M were defined to be the same as the class designations (“benign” or “malicious,” respectively). All classifiers were trained to produce output $O$ such that $O \in \{B,M\}$. The classifier software, jBNC, produced two output values: the number of correctly classified samples vs. the number of incorrectly classified and a margin of error. The margin of error was always proportional to the accuracy and was therefore not very useful.

**Results.** Since the Bayes’ classifiers were binary classifiers in the context of this project, one approach was used in determining their accuracy: the number of elements classified correctly versus the number classified incorrectly. Ultimately, the choice of quality measure had no effect on the performance of the classifiers, so one set of results from each classifier type is displayed. Table 6 shows the results.

Overall, the Bayes’ classification approach did not produce very encouraging results. As stated previously, only three out of the five classifier types produced tests that
TABLE 6
BAYES’ CLASSIFICATION RESULTS, DATA SET C

<table>
<thead>
<tr>
<th></th>
<th>Naïve Bayes</th>
<th>TAN</th>
<th>FAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>73%</td>
<td>86%</td>
<td>74%</td>
</tr>
<tr>
<td>Margin of error</td>
<td>4.462%</td>
<td>3.487%</td>
<td>4.408%</td>
</tr>
</tbody>
</table>

ran to completion. Of the three that actually completed, the highest accuracy achieved was only 86% (TAN). The software (jBNC) did not offer any further customization options beyond classifier type and quality measure; therefore, performance could not be improved beyond what was observed here.

Ultimately, the only variable that had any effect on performance was the classifier type. Out of the three that were successfully tested, TAN performed the best. TAN relaxes the independence assumptions of traditional NB and allows arcs between attribute nodes (N-1 arcs specifically, N being the number of attribute nodes) to indicate dependence between them. These additional arcs form a tree over the original classifier. TAN has been shown to outperform traditional NB [18], so that aspect of the results obtained here was expected. The relatively poor performance of FAN made sense as well. FAN is a variant of TAN with even more relaxed constraints. Whereas TAN requires arcs between each attribute (N-1 total arcs), FAN requires between 0 and N-1 arcs [18]. FAN therefore falls somewhere in between NB and TAN in terms of network structure, and this was reflected in its performance.

The selective variants of TAN and FAN (STAN and SFAN, respectively) required much more time than their non-selective counterparts. These variants add the ability to discard or ignore attributes that are deemed irrelevant [48]. The process of searching for an optimal subset of attributes to retain may have been the cause of the
lengthy time requirements for these classifier types. Such a large set of attributes (2381) would result in a huge number of candidate subsets for elimination. Given the fact that these naïve and simplified versions of a Bayes’ network required so much time, a test with a full Bayes’ network was not attempted.

**Neural Networks**

The simulation software used to train and test the neural networks was Neuro Solutions 5 [49]. This software offered a plethora of customization options for neural networks, and the following eight were incorporated into training and testing:

1. Network type
2. Cross validation
3. Hidden layer size
4. Momentum
5. Step size
6. Activation function(s)
7. Update type
8. Training epochs

The type of neural network selected was a generalized feed-forward neural network. This was due to the fact that this type of network is one of the simplest neural networks. The chances for complications and errors due to complexity and unknowns would be kept to a minimum by taking this approach.

Values for other parameters in the above list were adjusted and varied on a per-test basis with the goal of determining optimal values for each. Testing of all combinations of these parameters or even a substantially large set of them was not feasible due to the large amount of time required to train and test each network.
The general network structure was kept constant across all tests. In each case, the network had the following properties:

a. An input layer made up of 2381 nodes, each corresponding to one of the unique Win32 functions encountered during the executable analysis phase.

b. A single hidden layer, whose node count varied from test to test.

c. A single output node.

Each network’s single output node was trained to produce $O$, such that $O \in \{B,M\}$. For all tests, $B$ was set to 50, and $M$ was set to 1000.

Training process. In instances where the size of the hidden layer was greater than 500 nodes, training of the network required at least 7 hours. As a result, size of the hidden layer was always held to a fraction of the size of the input layer; varying between 300 and 600 nodes. Increasing the size significantly beyond 600 was not feasible.

Testing across all data sets employed the batch update approach to training the network. This meant back-propagation was not applied to the network to adjust the weights until the entire training set had been pushed through the network and a cumulative error factor computed [38]. This method was chosen over on-line updating after a single test-run of the on-line method showed it to be severely inadequate under the circumstances of this project in terms of error rate.

Variables. The variables that were adjusted on a per test basis included the following: hidden layers size, step size, momentum, epoch count, and activation function. They were adjusted in a manner that attempted to move them towards a value that optimized the output of the network (lowered overall network error).
Inputs. Input values for each data set are as follows:

- **Data Set A**: Each node in the input layer represented one of the 2381 Win32 functions that were recorded during the entire executable analysis period. The values for the nodes were the count of each call made during the execution of a given executable.

- **Data Set B**: Same as data set A, except the input values were the adjusted function call counts from set B.

- **Data Set C**: Same as A and B, except the input values became restricted to one of two values: 0 for a function that was never called for a given executable; and 1000 for a function that was called one or more times.

Outputs. All output from the trained neural network had some margin of error. This meant that the output would never be exactly what it was trained to be. In order to convert the output to an element of \{B,M\}, a threshold $T$ in between the two values was introduced.

The process used to determine the value of $T$ was one of trial and error. The first approach was to simply set $T = (B + M)/2$. This resulted in $T = 525$. This initial value of $T$ was run against the first test results from data set A and produced a false negative rate of 8% with a false positive rate of 46%. An inspection of these results indicated that $T$ could be adjusted to 600 without significantly affecting the false negative rate, a move which improved the false positive rate to 28%, and increased the false negative rate by only 2%.

An inspection of the results from the final run of the training data in the first test corroborated the use of this value of $T$. In order to avoid biasing the value of $T$
towards a specific data set and network configuration, the value $T = 600$ was used in all subsequent tests, rather than determining a unique optimal value for each.

**Results.** When determining the performance of a particular network, two approaches were utilized:

- **Binary classification based on threshold:** Outputs, regardless of their actual value, were converted to either $B$ or $M$ based on their value relative to $T$. This resulted in a simple “yes” or “no” answer to whether or not a given executable was malicious.

- **General error percentage:** The output for a given executable was compared to the desired output for that executable, and the difference was used to formulate an error percentage. This resulting value represented the probability that the tested executable was benign or malicious.

1. **Data Set A.** For data set A (the raw data) the results were the most unfavorable out of all three sets of data. Despite the low false negative rate (3% - 5%), every test had a high false positive rate, ranging from 14% up to 20%. Table 7 shows the results.

2. **Data Set B.** Data set B showed significant improvement over A and contained the best performing test out of all data sets (test 1), with an accuracy of 91%. The false positive rate ranged from 5% to 11%, and the false negative rate ranged from 4% to 7%. Table 8 shows the results.

3. **Data Set C.** Data set C showed slight improvement in some respects, and some slight regression is others. The false positive rate was higher (ranging from 9% to 10%) but the false negative rate ranged from only 3% to 4%. Table 9 shows the results.
### TABLE 7
**NEURAL NETWORK RESULTS, SET A**

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training epochs:</td>
<td>1000</td>
<td>1500</td>
<td>750</td>
<td>1000</td>
</tr>
<tr>
<td>Learning rate:</td>
<td>.1</td>
<td>.08</td>
<td>.1</td>
<td>.08</td>
</tr>
<tr>
<td>Momentum:</td>
<td>.7</td>
<td>.5</td>
<td>.7</td>
<td>.85</td>
</tr>
<tr>
<td>Activation function:</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
</tr>
<tr>
<td>Hidden layer size:</td>
<td>500</td>
<td>600</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>Benign Avg. Error %</td>
<td>35.06%</td>
<td>42.68%</td>
<td>42.39%</td>
<td>43.39%</td>
</tr>
<tr>
<td>Malicious Avg. Error %</td>
<td>29.71%</td>
<td>29.26%</td>
<td>29.05%</td>
<td>26.37%</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>14</td>
<td>20</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Overall Avg. Error %:</td>
<td>32.39%</td>
<td>35.97%</td>
<td>35.72%</td>
<td>29.88%</td>
</tr>
<tr>
<td>Within threshold (out of 100):</td>
<td>81</td>
<td>77</td>
<td>78</td>
<td>83</td>
</tr>
</tbody>
</table>

### TABLE 8
**NEURAL NETWORK RESULTS, SET B**

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training epochs:</td>
<td>1000</td>
<td>500</td>
<td>500</td>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>Learning rate:</td>
<td>.1</td>
<td>.1</td>
<td>.08</td>
<td>.07</td>
<td>.1</td>
</tr>
<tr>
<td>Momentum:</td>
<td>.7</td>
<td>.7</td>
<td>.75</td>
<td>.8</td>
<td>.7</td>
</tr>
<tr>
<td>Activation function:</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
</tr>
<tr>
<td>Hidden layer size:</td>
<td>500</td>
<td>300</td>
<td>500</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>Benign Avg. Error %</td>
<td>16.62%</td>
<td>24.90%</td>
<td>18.45%</td>
<td>26.24%</td>
<td>19.45%</td>
</tr>
<tr>
<td>Malicious Avg. Error %</td>
<td>13.70%</td>
<td>16.64%</td>
<td>16.33%</td>
<td>14.88%</td>
<td>14.37%</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Overall Avg. Error %:</td>
<td>15.16%</td>
<td>20.77%</td>
<td>17.39%</td>
<td>20.56%</td>
<td>16.91%</td>
</tr>
<tr>
<td>Within threshold (out of 100):</td>
<td>91</td>
<td>83</td>
<td>88</td>
<td>84</td>
<td>89</td>
</tr>
</tbody>
</table>
### TABLE 9  
**NEURAL NETWORK RESULTS, SET C**

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training epochs:</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Learning rate:</td>
<td>.1</td>
<td>.1</td>
<td>.08</td>
<td>.1</td>
<td>.08</td>
</tr>
<tr>
<td>Momentum:</td>
<td>.7</td>
<td>.75</td>
<td>.7</td>
<td>.7</td>
<td>.85</td>
</tr>
<tr>
<td>Activation function:</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
<td>Sigmoid &amp; Linear</td>
<td>Sigmoid</td>
<td>Sigmoid</td>
</tr>
<tr>
<td>Hidden layer size:</td>
<td>500</td>
<td>600</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Benign Avg. Error %</td>
<td>20.88%</td>
<td>24.3%0%</td>
<td>33.32%</td>
<td>23.38%</td>
<td>20.88%</td>
</tr>
<tr>
<td>Malicious Avg. Error %</td>
<td>12.55%</td>
<td>11.78%</td>
<td>11.50%</td>
<td>9.94%</td>
<td>16.25%</td>
</tr>
<tr>
<td>False positives (out of 100):</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>False negatives (out of 100):</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Overall Avg. Error %:</td>
<td>16.72%</td>
<td>18.04%</td>
<td>22.41%</td>
<td>16.66%</td>
<td>18.57%</td>
</tr>
<tr>
<td>Within threshold (out of 100):</td>
<td>88</td>
<td>88</td>
<td>86</td>
<td>87</td>
<td>86</td>
</tr>
</tbody>
</table>

**Results Analysis.** Again, the differences between data sets A, B, and C showed that the call count of the functions logged was a significant factor in accuracy. In fact, it seemed to be the single largest factor affecting the accuracy of the network. Data set A was the only set to incorporate the raw call counts, and showcased the worst performance out of all three data sets. When the input to the network was changed from the raw function call count in data set A to data set B, a significant improvement in performance was observed. The transformation from B to C resulted in a small overall decline in performance.

Adjusting the customizable options presented by Neuro Solutions impacted network performance as well, but not as significantly as the choice of data set. From a purely observational standpoint, the variable that seemed to impact the results the least was the number of training epochs, provided that number was above the 500 mark. When
tests were run with concurrent cross validation with the test set, the cross validation data showed that the most dramatic performance increases occurred before reaching the 500 epoch mark. Other observations that supported this conclusion:

- The first and fifth tests in data set B had identical parameters, except the fifth test ran for half as many epochs (500 vs. 1000). The extra 500 iterations only increased overall accuracy by 2%, and false positives only decreased by 1%.

- The first set in C ran for twice as many epochs as test 4 in C (1000 vs. 500) and the results were very close: false positive rates of 9% and 10% respectively, and a false negative rate of 4% for the first test and 3% for the fourth test. Again, the only parameter that differed between the two tests was the number of training epochs.

The ideal size of the hidden layer seemed to be right around 500. In each data set the test with most accurate results had a hidden layer with 500 nodes. In fact, the size of the hidden layer seemed be the most critical factor of performance outside the choice of data set. Between test 2 and 3 in data set A, only the hidden layer is kept steady at 600 while the number of training epochs, step size, and momentum are altered significantly. The resulting error rate was constant between the two. Tests 2 and 3 in set B also are consistent with this finding, showing that when the step size and momentum are constant and only the hidden layer size is changed (from 500 to 300 respectively), the accuracy is significantly affected.

In the end, the exact optimal parameters for the neural network could not conclusively be determined because to achieve that goal would require a great deal more time than was available. The exact effect of adjusting the momentum and step size was not clear given the observed results. The default step size for a neural network in Neuro
Solutions 5 is 0.1 and the default momentum is 0.7. The best performing test was test 1 in set B and it used these values. It was the only test to break the 90% accuracy barrier. A comparison of tests 1 and 4 in set A shows a decrease in learning rate along with an increase in momentum that results in an increase of overall accuracy (81% to 83%). However, the same adjustment between tests 4 and 5 in set C results in a decrease in overall performance. A similar difference between tests 3 and 5 in set B shows a similar decrease in performance.

While the results that were obtained are not comprehensive enough to determine exact optimal parameter values, they do show a trend towards an ideal hidden layer size (500), an ideal data set (Set B), a value of about .1 for the step size and a value of about .7 for the momentum. The fact that set B was the ideal data set supports the idea that the function call count is a relevant factor in the classification of benign and malicious executables, but only when it is transformed in such a manner that it does not overwhelm the significance of the functions being called.

**Support Vector Machine**

The software used for constructing and testing SVMs was SVM Light [50]. This software offered several options for customizing SVM training parameters including:

1. **Performance options:**
   - Factor by which errors on positive examples outweigh errors on negative examples (default 1)
   - Option of biased hyperplane \((x\cdot w+b>0)\) instead of unbiased hyperplane \((x\cdot w>\theta)\) (default: biased)
• Option to remove inconsistent training examples and retrain (default: no)
• Option to compute leave-one-out estimates (default: no)

2. **Kernel options:**
   • Linear (default)
   • Polynomial
   • Radial basis
   • Sigmoid / tanh
   • User defined kernel

3. **Optimization options:**
   • Maximum size of QP-subproblems (default 10)
   • Size of cache for kernel evaluations (default 40 MB)
   • Number of iterations a variable needs to be optimal before considered for shrinking (default 100)
   • Max iterations with no progress (default 100,000)

**Training process.** SVM light was very well suited for the problem domain of this project. The training for each test was straightforward and very fast, never taking more than a minute to complete, except for tests in set A, which took over an hour.

**Variables.** Each of the customizable parameters listed above were experimented with on a per test basis. The goal was to adjust these values in a manner that allowed for the determination of which had the most significant impact on performance, as well as the optimal values for each. In the end however, these parameters had little to no effect on the overall accuracy of the SVM.
**Inputs.** The dimensionality of the data set for the SVMs was 2381, corresponding to the number of unique Win32 functions logged during the executable analysis phase.

- **Data Set A:** The value for each feature was the raw call count for the corresponding function. The class to which the data element belonged was either 1 for benign code, or -1 for malicious code.

- **Data Set B:** Same as set A, except input values were the adjusted call counts of data set B.

- **Data Set C:** Same as set B, except input values became restricted to one of two values: 0 for a function that was never called for a given executable; and 1000 for a function that was called one or more times.

**Output.** For this series of tests output values B and M were defined to be the same as the class designations (1 or -1, respectively). All SVMs were trained to produce output $O$ such that $O \in \{B,M\}$. The SVMs were, by design, binary classifiers. However, the output often had a margin of error and therefore was not exactly 1 or -1. A threshold of 0 was chosen to separate the classes since it fell directly between the two values.

**Results.** Like neural networks, two approaches were employed when determining performance of the SVMs:

- **Binary classification:** After factoring in the selected separation threshold, all output values were assigned to either the benign class or the malicious class, regardless of their actual value produced from the SVM.

- **General error percentage:** SVM Light provided an overall accuracy percentage for each SOM. This percentage was calculated by determining the magnitude
of error for each test input, using that input’s desired output value, and the actual output value.

1. **Data Set A.** The results from set A were essentially useless, regardless of which test was being performed. The huge variance in function call count simply overwhelmed the learning mechanism of the SVM. The training of SVMs with data from set A took over three hours and the result was an SVM that was never much above 50% in accuracy. Since adjusting the various training parameters had no effect on the results, and the results themselves had no value, only one test is displayed below for set A. Figure 21 shows the results.

<table>
<thead>
<tr>
<th>Test 1: Default parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results:</td>
</tr>
<tr>
<td>Run-time: 11338.60 sec.</td>
</tr>
<tr>
<td>Kernel evaluations: 43359</td>
</tr>
<tr>
<td>Overall accuracy: 50.51%</td>
</tr>
<tr>
<td>No. correct: 51/100</td>
</tr>
<tr>
<td>False positives: 0</td>
</tr>
<tr>
<td>False negatives: 49</td>
</tr>
</tbody>
</table>

Fig. 21. Support vector machine results, set A.

2. **Data Set B.** The results for set B were a huge improvement over set A. The influence of the huge inequality in function count was reduced to almost nothing. Training took only seconds to complete. However, like set A, adjusting the various training parameters had a very small effect on the performance of the SVM, and therefore only the “significant” tests are displayed below. Figure 22 shows the results.

3. **Data Set C.** The results from set C showed a very slight improvement from set B. With set C, adjusting the various training parameters had only a slight effect on the performance of the SVM. Figure 23 shows the results.
Results analysis. Disregarding the exceedingly inaccurate results from test A, the SVM approach showed excellent results. Test 4 in data set C showed an excellent accuracy rate at almost 96%, with a false positive rate of only 3% (the lowest of all tests) and a false negative rate of 2%. The other tests were very close; in accuracy they were all very close to 95%, except test 1 of set B, which was even better at 97%. However, these
other tests also has slightly higher false positive rates. Again, the function call count was a huge factor between data set A and the other two sets. The transition from the adjusted call counts in set B to the elimination of call counts in set C resulted in a negligible difference.

All things considered, there was not clear winner out of these tests. Test 4 in set C showed the lowest false positive rate, but test 1 in set B showed the highest overall accuracy rate. Test 4 also showed the highest number of correct identifications (95/100). Had there been more data elements in the training set or test set, the distinction between these two sets might have been clearer.

Due to the fact that the performance was so high when operating with the default settings, there was not much room for experimentation. For the data obtained in this project, the linear kernel appeared to be plenty adequate (all above results made use of this kernel). Tests were run with all built-in kernel types (polynomial, radial-basis, and sigmoid/tanh) and none of them showed an improvement over the linear kernel. This fact was evidence to the linear separability of the data.

**Self Organizing Maps**

The software used for constructing and testing SOMs was an add-on to MATLAB called SOM Toolbox [51]. Six customization options offered by the software were utilized:

- Training algorithm (incremental or batch)
- Map dimensions
- Nearest neighbor function (gaussian, cutgauss, ep, or bubble)
- Lattice (rectangular or hexagonal)
- Shape (sheet, cylinder, or toroid)
- Number of training iterations

**Training process.** The SOM toolbox contains a large number of functions and scripts written for the purposes of constructing, testing, and visualizing SOMs. There were far more available than was needed for this project; in the end only a handful were utilized. The single most important function in this subset was the generalized function `som_make()`. Its purpose is to construct an SOM from a given set of data and train it. It was chosen over its more specialized derivatives because it is simple, it offered all the parameter options listed above, and it suited the needs of this project well. Other functions offered by the SOM toolbox that were used for this project included functions to load and normalize the data, as well as functions to compute accuracy and performance measures.

Both the batch and incremental training algorithms (described in section 2.5) were employed during training. Training of an SOM using the batch approach often completed in minutes, whereas the incremental approach took hours.

**Variables.** Each of the customizable parameters listed above was experimented with on a per test basis. The goal was to adjust these values in a manner that allowed for the determination of which had the most significant impact on performance, as well as the optimal values for each. Like the SVM tests, these parameters ultimately had a very small effect on the overall accuracy of the SOM.

**Inputs.** The dimensionality of the data set for the SOMs was 2381, corresponding to the number of unique Win32 functions logged during the executable
analysis phase. The SOM software required each input vector to be associated with a label; the label acted as the class designation for that input vector.

- **Data Set A:** The value for each feature of the input vector was the raw call count for the corresponding function. Each input vector was assigned to the either the “benign” class or the “malicious” class.

- **Data Set B:** Same as set A, except input values were the adjusted call counts of data set B.

- **Data Set C:** Same as set B, except input values became restricted to one of two values: 0 for a function that was never called for a given executable; and 1000 for a function that was called one or more times.

**Output.** Through the training process, each node in the map was ultimately associated with one of the two previously mentioned classes (benign or malicious). When a testing example was fed into the system, the class associated with that example’s BMU became the output.

**Results.** Two approaches were employed when determining performance of the SOMs:

- **Binary classification:** Outputs were classified as either being benign or malicious based on the map node to which they were numerically most similar.

- **Overall map quality:** SOM toolbox provides an assessment function that quantifies the overall quality of a given SOM. It produces two values:
  
  - **Mapping precision:** This value is a measure of how accurately the map nodes respond to the given data set. It is calculated by finding the average distance between the input vectors and their BMUs. This variable is defined as $q$: 
where $x_i$ is an input vector and $m_c$ is a map vector.

- **Topographic error**: The proportion of all data vectors for which first and second BMUs are not adjacent units. This measures topology preservation. This variable is defined as $t$:

$$t = \frac{1}{n} \sum_{k=1}^{n} u(x_k)$$

where $u(x_k)$ is 1 if the first and second BMUs for input vector $x_i$ are not next to each other, and 0 otherwise.

1. **Data Set A**. The results from set A were the worst out of all three sets. In only one test, test 4, was the accuracy able to break the 80% mark. While the false negative rate wasn’t too bad (varying between 1% and 6%), the number of false positives was consistently high, reaching a maximum of 22%. Table 10 shows the results.

2. **Data Set B**. The results from set B saw a marked improvement over set A. The false positive rate was significantly reduced, reaching a low of 5%. The false negative rate saw a decrease as well, reaching a low of 2%. Table 11 shows the results.

3. **Data Set C**. The results from set C were essentially equal to those set B. The maximum accuracy level for both sets was 93% and both had similar false positive and false negative rates. Table 12 shows the results.

**Results analysis**. As expected, the difference between the raw call counts of data set A and the modified call counts of the other two data sets was a huge factor. The respective performances of sets B and C were very close, to point that their differences
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
<th>Test 6</th>
<th>Test 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map size:</td>
<td>25 x 25</td>
<td>30 x 30</td>
<td>35 x 35</td>
<td>35 x 35</td>
<td>40 x 40</td>
<td>45 x 45</td>
<td>45 x 45</td>
</tr>
<tr>
<td>Shape</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
</tr>
<tr>
<td>Lattice</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>rect.</td>
</tr>
<tr>
<td>Iterations:</td>
<td>15/40</td>
<td>15/40</td>
<td>25/65</td>
<td>65/100</td>
<td>65/100</td>
<td>115/160</td>
<td>150/200</td>
</tr>
<tr>
<td>False positives:</td>
<td>22/100</td>
<td>17/100</td>
<td>17/100</td>
<td>16/100</td>
<td>22/100</td>
<td>20/100</td>
<td>21/100</td>
</tr>
<tr>
<td>False negatives:</td>
<td>5/100</td>
<td>5/100</td>
<td>5/100</td>
<td>1/100</td>
<td>4/100</td>
<td>6/100</td>
<td>5/100</td>
</tr>
<tr>
<td>Map quality (q):</td>
<td>2.861e5</td>
<td>2.813e5</td>
<td>2.823e5</td>
<td>2.738e5</td>
<td>2.771e5</td>
<td>2.680e5</td>
<td>2.680e5</td>
</tr>
<tr>
<td>Map quality (t):</td>
<td>.03</td>
<td>.02</td>
<td>.03</td>
<td>.08</td>
<td>.12</td>
<td>.05</td>
<td>.07</td>
</tr>
<tr>
<td>Accuracy:</td>
<td>73%</td>
<td>78%</td>
<td>78%</td>
<td>83%</td>
<td>74%</td>
<td>74%</td>
<td>74%</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Test 1</td>
<td>Test 2</td>
<td>Test 3</td>
<td>Test 4</td>
<td>Test 5</td>
<td>Test 6</td>
<td>Test 7</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
</tr>
<tr>
<td>Map size:</td>
<td>30 x 30</td>
<td>30 x 30</td>
<td>35 x 35</td>
<td>35 x 35</td>
<td>40 x 40</td>
<td>45 x 45</td>
<td>45 x 45</td>
</tr>
<tr>
<td>Shape</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
</tr>
<tr>
<td>Lattice</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>rect.</td>
<td>hexa.</td>
</tr>
<tr>
<td>Iterations:</td>
<td>8/23</td>
<td>10/45</td>
<td>10/45</td>
<td>15/55</td>
<td>15/55</td>
<td>25/75</td>
<td>25/75</td>
</tr>
<tr>
<td>False positives:</td>
<td>7/100</td>
<td>7/100</td>
<td>6/100</td>
<td>6/100</td>
<td>5/100</td>
<td>5/100</td>
<td>6/100</td>
</tr>
<tr>
<td>False negatives:</td>
<td>2/100</td>
<td>1/100</td>
<td>3/100</td>
<td>2/100</td>
<td>3/100</td>
<td>3/100</td>
<td>5/100</td>
</tr>
<tr>
<td>Map quality ((q)):</td>
<td>(6.993e^3)</td>
<td>(7.073e^3)</td>
<td>(6.885e^3)</td>
<td>(6.917e^3)</td>
<td>(6.756e^3)</td>
<td>(6.688e^3)</td>
<td>(6.618e^3)</td>
</tr>
<tr>
<td>Map quality ((t)):</td>
<td>.11</td>
<td>.10</td>
<td>.04</td>
<td>.16</td>
<td>.10</td>
<td>.11</td>
<td>.18</td>
</tr>
<tr>
<td>Accuracy:</td>
<td>91%</td>
<td>92%</td>
<td>91%</td>
<td>92%</td>
<td>92%</td>
<td>92%</td>
<td>89%</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Test 1</td>
<td>Test 2</td>
<td>Test 3</td>
<td>Test 4</td>
<td>Test 5</td>
<td>Test 6</td>
<td>Test 7</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
<td>Batch</td>
</tr>
<tr>
<td>Map size</td>
<td>20 x 20</td>
<td>30 x 30</td>
<td>30 x 30</td>
<td>40 x 40</td>
<td>40 x 40</td>
<td>40 x 40</td>
<td>45 x 45</td>
</tr>
<tr>
<td>Shape</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
<td>sheet</td>
</tr>
<tr>
<td>Lattice</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
<td>hexa.</td>
</tr>
<tr>
<td>Iterations</td>
<td>10/40</td>
<td>10/40</td>
<td>20/55</td>
<td>20/55</td>
<td>30/75</td>
<td>50/95</td>
<td>50/95</td>
</tr>
</tbody>
</table>

False positives: 6/100 6/100 6/100 7/100 5/100 7/100 5/100 5/100 5/100 5/100 
False negatives: 3/100 2/100 2/100 3/100 3/100 3/100 3/100 3/100 2/100 2/100 
Map quality (q): 6.004 5.731 5.716 5.711 5.705 5.507 5.427 5.261 5.819 5.627 
Map quality (t): .08 .13 .09 .11 .10 .12 .12 .17 .07 .12 
Accuracy: 91% 92% 92% 90% 92% 90% 92% 92% 92% 93%
were negligible, but both were significantly better than set A. The manner in which the function call counts were incorporated into the training was the most significant factor in the performance of the SOM.

Like SVMs, when the results from testing with set A were ignored, the overall results were pretty good. Both sets B and C reached highs in overall accuracy of 93% with averages of 91% to 92%, and lows in false positive rates of 5% with averages of 6% to 7%. The default values for the SOMs seemed to produce the best results. Changes in the lattice, like in test 7 of set B, seemed to decrease the accuracy. Changes in the nearest neighbor function, as in tests 8 and 9 in set B, showed no real difference in performance. Changes from a batch to incremental training approach only resulted in longer training times, but no real improvement in performance. Surprisingly, increases in the map size did not produce an improvement. Test 3 in set C, with a map size of 30 x 30, produced equivalent results to that of test 8, whose map was much larger at 55 x 55. Ultimately, the adjustment of all of these parameters produced very small changes in performance.

While set A was clearly the worst performer, there was no clear winner between sets B and C. This indicated that the function call count was not relevant for SOM training and classification; it only serves to decrease the accuracy. The only significant data element, in the context of an SOM, was the identity of a function and not the number of times it was called.

N-Gram Analysis

For this test, a unique and unconventional approach was designed. As discussed in section 3.1.1, the output format chosen for this project eliminated function call ordering information to reduce log size and processing requirements. Extraction of n-
grams is traditionally implemented by parsing them from sequential, continuous tokens. This approach was not possible because of the loss of ordering information. Instead, it was decided to construct n-grams by pulling combinations of size \( n \) from the call logs with no regard to order.

For this particular technique, a custom solution was implemented. The solution was developed in C# using Microsoft Visual Studio. It was designed to extract combinations of size \( n \) from the function call logs and keep track of the occurrence frequency for each. Candidate tokens for the n-grams were limited to \( F \) functions pulled from an ordered occurrence frequency list of Win32 function indexes. The training process was simply the act of collecting these counts.

**Variables.** Only two variables were modified over the test iterations. The first was \( n \), the number of tokens in the n-gram. The range for \( n \) was constrained to \{2,3,4\}. The second variable, \( F \) was varied over a different range for each value of \( n \):

- **Bigrams:** \{100, 200, 250, 300, 500, 1000, 2000\}
- **Trigrams:** \{100, 200, 250, 300\}
- **4-grams:** \{100\}

**Inputs.** Since the only attribute considered for the n-grams was the index of the Win32 functions, the call count was simply ignored. This meant that all three data sets produced the exact same sets of n-grams. The inputs for the test set were simply the n-grams collected from the call logs for the executables in that set.

**Training process.** The training for each combination of \( n \) and \( F \) involved calculating all possible combinations of Win32 functions and counting how many times
each occurred in the benign and malicious portions of the training set. Each unique combination fell into one of three categories:

1. Only appearing in benign executables.
2. Only appearing in malicious executables.
3. Appearing in both benign and malicious executables.

Because of the way the n-grams were formed, at most one instance of a given n-gram would appear in a given executable. This meant that typical statistical and probabilistic methods could not be applied. So the following approach was taken: in the default training iterations for a given combination of $n$ and $F$, a value of 1 was added to an n-gram for each instance in a benign executable, and a value of -1 for each instance in a malicious executable. These values were adjusted further in additional training iterations within the same combination of $n$ and $F$ in an effort to increase performance.

Note: Ultimately, the running of 4-grams failed even with an $n$ value as low as 100 because the resource requirements were too large.

Output. After training, each unique n-gram discovered during the training process had an integer $n$ assigned to it based on many times it occurred in benign and malicious executables respectively. The outputs from testing for each executable was simply the total of $n$ over each n-gram discovered in that executable (new n-grams discovered in the testing set were simply discarded).

Results. The output format choice dictated only one type of approach to determining the performance of the n-gram method: binary classification. Size negative values were assigned for malicious n-grams and positive values were assigned to benign n-grams, 0 was the logical threshold between the two.
Ultimately, the results were all together quite poor, but expected. For all values of $n$ and $F$, the default training approach produced accuracy rates of 49%-50%. The false positive rate for all these tests was 0%, accompanied with a false negative rate of 100%. All executables were detected as benign. Neither the size of the n-gram nor the number of function indexes included as valid tokens changed this result. In light of these abysmal results, a small tweak was made in the training process: the value for an instance of an n-gram that was found in malicious executables was increased to -2, regardless if it was also found in a benign executable or not. This change resulted in increased accuracy, shown in Table 13.

<table>
<thead>
<tr>
<th>n-gram type</th>
<th>F=100</th>
<th>F=200</th>
<th>F=250</th>
<th>F=300</th>
<th>F=500</th>
<th>F=1000</th>
<th>F=2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigram</td>
<td>33/43</td>
<td>28/44</td>
<td>28/44</td>
<td>28/43</td>
<td>28/43</td>
<td>28/42</td>
<td>28/40</td>
</tr>
<tr>
<td>Trigram</td>
<td>36/36</td>
<td>34/37</td>
<td>34/39</td>
<td>34/39</td>
<td>32/39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-gram</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

The above total shows the false positive and false negative rates for various combinations of $n$ and $F$ separated by a slash. The results above were the best that were achieved by tweaking the values discussed in the training section. Other values were experimented with, but produced less accurate results. In one series of tests, n-grams that were not present in both benign and malicious executables were removed from consideration. This resulted in the same level of accuracy as the default tests: approximately 50%. In still one more approach, n-grams with a small frequency of occurrence (less than 20%) were discarded. Again, the accuracy rate did not improve beyond approximately 50%.
Overall, the results show that the n-gram approach was not a viable option in the context of this project. This was expected, given a basic analysis of the n-gram statistics collected. For every combination of $n$ and $F$, the number of total potential combinations of function indexes was very high ranging from 7565 with $n=2$ and $F=100$ on up to 12,166,770 with $n=3$ and $F=300$. Tables 14, 15, and 16 show these results.

**TABLE 14**

<table>
<thead>
<tr>
<th>Exclusively Benign N-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-gram type</td>
</tr>
<tr>
<td>Bigram</td>
</tr>
<tr>
<td>Trigram</td>
</tr>
<tr>
<td>4-gram</td>
</tr>
</tbody>
</table>

**TABLE 15**

<table>
<thead>
<tr>
<th>Exclusively Malicious N-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-gram type</td>
</tr>
<tr>
<td>Bigram</td>
</tr>
<tr>
<td>Trigram</td>
</tr>
<tr>
<td>4-gram</td>
</tr>
</tbody>
</table>

In every test, the number of n-grams that were found solely in malicious executables was always exactly zero. This meant that all the malicious executables, as far as n-grams were concerned, looked like smaller versions of benign executables. Only through forcing a non-logical value for malicious n-grams could the accuracy rate be
TABLE 16

N-GRAMS IN BOTH BENIGN AND MALICIOUS EXECUTABLES

<table>
<thead>
<tr>
<th>n-gram type</th>
<th>F=100</th>
<th>F=200</th>
<th>F=250</th>
<th>F=300</th>
<th>F=500</th>
<th>F=1000</th>
<th>F=2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigram</td>
<td>4830</td>
<td>11772</td>
<td>14398</td>
<td>18632</td>
<td>30102</td>
<td>49506</td>
<td>69432</td>
</tr>
<tr>
<td>Trigram</td>
<td>328440</td>
<td>1259604</td>
<td>1860744</td>
<td>2515320</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>4-gram</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

coerced above 50%. This was not regarded as a viable option since it was merely an experimental move and was not based on any logical reasoning.

One problem with the n-gram approach used in this project is that the number of candidate tokens for the n-grams was too large. This resulted in a huge, sparse, and ultimately meaningless feature space. Because of the way the function call logs were formatted, the n-grams could not be constructed out of sequential function calls, and the resulting n-grams were not accurate representations of the original executables.

A second problem may have been the size of the n-grams. The maximum size successfully tested was $n=3$. Because the token alphabet had a potential size of 2831, this size for $n$ simply was not large enough. Because of resource constraints, $n$ could not be increased beyond this size and therefore was limited to a very small amount compared to the total alphabet size. Given a smaller alphabet, or the possibility for larger values of $n$, the results may have been better.
CHAPTER IV

PROJECT RESULTS

The preliminary results from this project support the idea that AI techniques can indeed be applied to the detection of unseen malicious code with a feature set derived from Win32 API calls, and provided evidence to the superiority of some techniques over others. Although none of the techniques that were tested performed with 100% accuracy, some performed quite well. Overall, the support vector machine was the technique that produced the best results, with detection accuracy reaching as high as 95% and false positives reaching as low as 3%. Other techniques, such as decision trees and self-organizing maps produced accuracy rates in low 90s after some tweaking. Not all techniques performed well however; Bayes’ classifiers only reached a high of 86% and for most configurations required far too much time to even produce results. The implementation using N-grams was a complete failure altogether and was not a viable option in the context of this project.

A few questions remained after the completion of testing. The first of these was how the inclusion of Win32 API functions that appeared only in benign executables or only in malicious executables may have skewed the final results. The concern was that the various techniques tested may have latched on to these functions during training and may have been reduced to simple signature detection mechanisms. To address this issue, a fourth data set D was constructed. It was a modification to set C, since in cases where
favorable results were achieved set C produced front running tests. The only difference from set C was that set D excluded Win32 API functions that were not found in both benign and malicious executables. Each AI technique was then tested with set D, although not as extensively as each of the other sets. The results from set D showed negligible differences from those of set C, which was evidence to that fact that the inclusion of functions only found in malicious or benign code did not invalidate the results.

Another question was: Is a false positive more significant than a false negative or vice versa? Or are they equivalent in importance and it is the overall error rate that matters? This question did not have an obvious answer and was one that couldn’t be answered by simply examining the data or running additional tests. A false positive is an error on the side of too much caution; the result is that a benign program will not run. This action will not result in any harm to the system. A false negative, on the other hand, is an error on the side of too little caution and allows for a malicious program to run. The result is that damage may indeed be done to the system. It should also be considered that a false positive will happen many, many times if it happens once, because duplicates of benign software like Microsoft Word or Photoshop exist on many machines around the world. One solution might be to keep a database of known false positives and check against it when a virus can reveals a potentially malicious program. For the sake of simplicity, both false negatives and false positives were treated as equals for this project.

Another concern was the absence of true cross validation from the project. The total number of executables that ran to completion and produced viable logs was 1100. The size of the feature space was 2381. Because of the size difference, the choice
was made to not make the already small training set any smaller by dedicating a portion to cross training. Ideally cross training would have been used to assist in determining optimal solutions for each technique.

One last issue was the amount of training data used. Given the huge number of potential Win32 functions to be called, obtaining a thorough representation of all types of programs requires a large data set. The data in this project was limited by the number of Win32 viruses available from the virus source. Potentially one of the most significant factors in the performance of many AI techniques is the size and make-up of the training set. Over 1000 executables were available in the Win32 Virus set from VX heavens, but ultimately only 550 were used after filtering out programs that crashed or produced minimal function call logs. The question remained: Were viruses in the final virus set an accurate representation of Win32 viruses in general? In order to be successful in the field against viruses that were in neither the training nor the testing set, this would need to be true. It should be noted that this potential shortcoming could be alleviated over time. While initially there may be some amount of inaccuracy in the training set as far as being a complete representation of Win32 viruses, the training set could be constantly updated and refined with new viruses as they are released. This would not only increase the size of the training set on which the techniques are based, it would keep it up to date as virus implementation techniques evolve.
CHAPTER V

CONCLUSION

The malware analyzed and tested for this project were of a single specific class: Viruses written for the Microsoft Windows platform. Traditional viruses were chosen as the subject of study for their well understood nature and long history of wreaking havoc on computer systems worldwide. However, viruses are by no means the only form of malware to be concerned with and within the last decade, with the Internet become so prevalent in every-day life, the definition of malware has been expanded to include newer and evolving types: Worms, Trojans, Bots, Spyware, Rootkits, and more [52]. As it stands, the solution designed for this project could not easily be applied to these other types of malware given their platform, operating environment requirements, and nature of execution, but its driving principles could be used to design similar tools for them.

The reason Win32 API calls were chosen as the target feature from which the inputs to the AI techniques would be generated was that they presented a functional view of programs. The Win32 function profile for a given executable represented a high level view of the actions and tasks that executable performed. A shift away from the brute force physical and structural analysis employed by many traditional anti-virus products was achieved and allowed for the use of intelligence-based approaches. Signature-based techniques are hard-wired to work with a very specific subject whereas intelligence-based
techniques can be adapted and extended to various platforms and malware types. Since a functional view could conceivably be derived for many or possibly all types of malware, techniques engineered to work at a functional level could be applied to them. A related advantage to this approach was the circumvention of malicious code obfuscation. Many times viruses or other types of malware will use obfuscation techniques such as function reordering, subtle morphing techniques, or encryption to hide the true executable instructions they contain. This makes the job of anti-virus programs that rely on signature-based methods quite difficult. It would not, however, have any impact on the accuracy of techniques showcased in this project because the analysis was dynamic and did not depend on how the malicious code physically appeared.

The results from this project were promising. They definitely show a trend as far as which AI techniques are better suited for the task of malware detection, at least within the constraints of this project. They also indicate that techniques like support vector machines have the potential to be quite good at it. This brings up the question: How good is good enough? What is an acceptable level of error? That it not such an easy question to answer and is heavily dependent on the purpose for which the software is applied. If the software is used to augment existing signature-based products, a small amount of error may be acceptable since the goal to simply to make existing techniques better. If the goal is security for systems containing very sensitive classified information, any amount of error may not be acceptable. Or it may be that false positives are acceptable, but not false negatives, since the latter would result in whatever harm the malware may be capable of inflicting whereas a false positive is just an annoyance.
CHAPTER VI

FUTURE RESEARCH

Preexisting Knowledge

Preexisting knowledge about the functions within the Win32 API was not in any way incorporated into this project. The nature and purpose of most of these functions, as well as their potential impact on the system are well known and documented and could potentially be used to aid analysis. One possible approach might be to assign a simple system of weights to the functions: Functions with the potential to significantly impact the system, in whatever sense that may be, should be given a higher weight than functions that have little or no impact.

As an alternative to a system of simple weights, a more intricate system of policies could be employed on a function by function basis. This is basically taking the aforementioned system of weights a few steps further and extending it beyond looking at the function as a whole and generalizing its functionality across all instances of calls made to that function. In this case, the focus would be on parameters used in each individual call to a given function. These parameters would more precisely dictate the behavior of that function and the impact to the system, and would lead to a more accurate and meaningful classification for that function. Policies could be developed around certain combinations of parameters, and instances of function calls would be checked.
against these policies to determine the probability that the function is called in a malicious context.

Obstacles to this approach include:

- The potentially huge amount of data to be collected and analyzed if each function call is treated on an individual basis, with different parameter combinations potentially yielding many policies for a single function.
- Extensive research about each function must be conducted before testing to create the policies for each function.

In an environment like Windows where there are thousands of function calls available, this would be a time consuming process.

Malware Pre-classification

In the domain of this particular project, program classification was very basic and non-granular. A given executable was either simply classified as malicious or benign. This is a limitation due to the source of the viruses that were tested. The only information available about each was the name of the virus and the fact that it was a Win32 executable. The result was a very cut and dry classification: a single value to indicate a benign executable, and an alternative single value to indicate a virus. Reality is not that black and white. Malicious executables can exist in varying degrees of “maliciousness,” and their Win32 function call profile will reflect that. If, for example, a given executable is technically classified as a virus because it exhibits a small amount of “maliciousness,” but for the most part acts like a benign executable, it really should be classified somewhere in between the values for benign and malicious, respectively. Forcing it to be
classified on one of either ends of the spectrum is not technically accurate. Thus trying to train an AI technique to produce such a coarse set of resultant output values, when in actuality the results should be much more granular, will result in a margin of increased inaccuracy over a situation where the “degree of maliciousness” for each virus is known and a finer/larger set of output possibilities is used.

Multi-technique Approaches

Instead of relying on single techniques, a way to improve accuracy might be to use combinations of techniques. It is quite possible that given technique might perform well in the same area that another is deficient. Combining techniques could be a way to weed out these deficiencies. The key to making this approach work would be determining the way to combine methods so that additional error or inaccuracy would not be introduced.

Deeper Testing

This project employed a breadth-oriented approach to testing the various machine learning techniques it covered. The testing of each was by no means exhaustive or complete; there was not enough time for that. Future research might include more focused and thorough testing for single techniques and various enhancements or alternate methods for them (such as CART, bagging, and boosting for decision trees).

Feature Pruning with Decision Trees

A distinct property of decision trees is that their organization is reflective of the quality of information present in the splitting of attributes at each node. Therefore it is
possible to examine the physical structure of decision trees to gain insight into the relative importance of each feature in classification. This knowledge could be applied to removing features that have little or no significance in the accuracy of the machine learning classifiers, and to determine which features are most important.

Input Feature Format

The chosen input format for this project carried with it some potential risks. It was a simplified version of reality in that it eliminated function call ordering information in order to reduce file size. Obviously, this was also a reduction in information, which potentially resulted in the machine learning techniques falling short of their true potential in terms of accuracy. Ideally, function call ordering information would be 100% preserved; an approach that would be possible only if the required processing power and storage space were available. In lieu of having those resources, it’s quite possible there exist s a much better simplification or reduction than the form chosen for this project. One possibility might a graph representation in which the functions are represented via nodes. There is a significant amount of room here for experimentation.
REFERENCES


