Data Mining Methods for Detection of New Malicious Executables

Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo
Computer Science Department, Columbia University
{eeskin,mgs,ezk,sal}@cs.columbia.edu

Abstract

A serious security threat today is malicious executables, especially new, unseen malicious executables. Many of these new malicious executables are undetectable by current anti-virus systems because they do not contain signatures for these new instances of malicious programs. These new malicious executables are created every day, and thus pose a serious security threat. We present a framework that detects new, previously unseen malicious executables. Comparing our detection methods with a traditional signature-based method, our method more than doubles the current detection rates for new malicious executables.

1 Introduction

A malicious executable is defined to be a program that performs a malicious function, such as compromising a system's security, damaging a system or obtaining sensitive information without the user's permission. Our goal is to automatically design and build a scanner using *data mining* methods that accurately detect malicious executables before they have been given a chance to run. Data mining methods detect patterns in large amounts of data, such as byte code, and uses these patterns to detect future instances in similar data. Our framework used *classifiers* to detect new malicious executables. A classifier is a rule set, or detection model, generated by the data mining algorithm that was trained over a given set of training data.

One of the primary problems faced by the virus community is to devise methods for detecting new malicious programs that have not yet been analyzed [24]. Eight to ten malicious programs are created every day and most cannot be detected until signatures have been generated for them [25]. During this time period, systems protected by signature based algorithms are vulnerable to attacks.

Malicious executables are also used as attacks for many types of intrusions. In the DARPA 1999 intrusion detection evaluation, many of the attacks on the Windows platform were caused by malicious programs [17]. Recently a malicious piece of code created a hole in a Microsoft's internal network [21]. That attack was initiated by a malicious executable that opened a back-door into Microsoft's internal network resulting in the theft of Microsoft's source code.

Detecting malicious executables is the key to protecting a system from these types of attacks. Automatic signature based scanners are already on the market, but they are not as accurate in detecting new malicious executables when compared to our methods. These automatic algorithms they rely on *signatures* of known malicious executables to generate detection models. Signature-based methods create a unique tag for each malicious program so that future examples of it can be correctly classified with a small error rate. These methods do not generalize well to detect new malicious binaries. The next evolution in malicious program detection is detecting these new malicious executables accurately and automatically.

We designed a framework that used data mining algorithms to train multiple classifiers on a set of malicious and benign executables to detect new examples. The binaries were first statically analyzed to extract properties of the binary, and then the classifiers trained over a subset of the data.

Our goal in the evaluation of this method was to simulate the task of detecting new malicious executables. To do this we separated our data into two sets: a *training set* and a *test set*. The training set was used by the data mining algorithms to generate classifiers to classify previously unseen binaries as malicious or benign. A test set is a subset of dataset that had no examples in it that were seen during the training of an algorithm. This subset was used to test an algorithms' performance over similar, unseen data and its performance over new malicious executables. The data we tested our detection models on was similar to the training data in that they were malicious executables gathered from public sources.

We implemented a traditional signature-based algorithm to compare with the the data mining algorithms. Using standard statistical cross-validation techniques this framework had a detection rate of 97.76%, over double the detection rate of a signature-based scanner.

1.1 Background

Detecting malicious executables is not a new problem in security. Early methods used signatures to detect malicious programs. These signatures were composed of many different properties: filename, text strings, or byte code. Research also centered on protecting the system from security holes these malicious programs created.

Experts were typically employed to analyze suspicious programs by hand. Using their expert knowledge signatures were found that made a malicious executable example different from other malicious executables or benign programs. One example of this type of analysis was performed by Spafford in 1988 [22]. He used his expertise to analyze the Internet Worm and provided detailed notes on the spread of it over the Internet, the unique signatures in the worm's code, the method of the worm's attack, and a comprehensive description of system failure points.

Although accurate, this method of analysis is expensive, and slow. If only a small set of malicious executables will ever circulate then this method will work very well, but the *Wildlist* [20] is always changing and expanding. The Wildlist is a list of malicious programs that are currently estimated to be circulating at any given time.

Current approaches to detecting malicious programs match them to a set of known malicious programs. The anti-virus community relies heavily on known byte-code signatures to detect malicious programs. More recently these byte sequences were determined by automatically examining known malicious binaries with probabilistic methods.

At IBM Kephart and Arnold [7] developed a statistical method for automatically extracting malicious executable signatures. Their research was based on speech recognition algorithms and was shown to perform almost as good as a human expert at detecting known malicious executables. Their algorithm was eventually packaged with IBM's anti-virus software.

Lo et al. [13] presented a method for filtering malicious code based on "tell-tale signs" for detecting malicious code. These are manually engineered based on observing the characteristics of malicious code. Similarly, filters for detecting properties of malicious executables have been proposed for UNIX systems [8] as well as semi-automatic methods for detecting malicious code [3].

Unfortunately, a new malicious program may not contain any known signatures so traditional signature based methods may not detect a new malicious executable. In an attempt to solve this problem different IBM researchers applied *Artificial Neural Networks*, ANN, to the problem of detecting boot sector malicious binaries [23]. An artificial neural network is a classifier that models neural networks explored in human cognition. Because of the limitations of their implementation of the classifier they were unable to analyze anything other than small boot sector viruses which comprise about 5% of all malicious binaries.

Using an ANN classifier with all bytes from the boot sector malicious executables as input, IBM researchers were able to identify 80–85% of unknown boot sector malicious executables successfully with a low false positive rate ($< 1\%$). They were unable to find a way to apply ANNs to the other 95% of computer malicious binaries.

Our method is different because we analyzed the entire set of malicious executables instead of only boot sector viruses.

Our technique is similar to data mining techniques that have already been applied to Intrusion Detection Systems by Lee et al. [11, 12]. Their methods were applied to system calls and network data to learn how to detect new intrusions. They reported ample detection rates as a result of applying data mining to the problem of IDS. We applied a similar framework to the problem of detecting new malicious executables.

2 Methodology

The goal of this work was to explore a number of standard data mining techniques to compute accurate detectors for new (unseen) data. We gathered a large set of programs from public sources and separated the problem into two classes: *malicious* and *benign* executables. Every example in our data set is a Windows or MS-DOS format executable, although the framework we present is applicable to other formats. To standardize our data-set, we used MacAfee's [14] virus scanner and labeled our programs as either malicious or benign executables.

We split the dataset into two subsets: the *training set* and the *test set*. The data mining algorithms used the training set while generating the rule sets. We used a test set to test the accuracy of the classifiers over unseen examples.

Next, we extracted a binary profile from each example in our dataset, and from the binary profiles we extracted *features* to use with classifiers. Features in a data mining framework are properties extracted from each example in the data set, such as byte sequences, that a classifier uses to generate detection models. Using different features, we trained a set of data mining classifiers to distinguish between benign and malicious programs.

The framework supports different methods for feature extraction and different data mining classifiers. We used system resource information, strings and byte sequences that were extracted from the malicious executables in the data set as different types of features. We also used three learning algorithms:

- an inductive rule-based learner that generated boolean rules based on feature attributes
- a probabilistic method that generated probabilities that an example was in a class given features
- a multi-classifier system that combined the outputs from several classifiers to generate a model.

To compare the data mining methods with a traditional signature-based method we designed an automatic signature generator. Since the virus scanner that we used to label the data set had been trained over every example in our data set, it was necessary to implement a similar signature-based method to compare with the data mining algorithms. There was no way to use an off-the-shelf virus scanner, and simulate the detection of new malicious executables because these commercial scanners contained signatures for all the malicious executables in our data set. Like the data mining algorithms, the signature-based algorithm was only allowed to generate signatures over the set of training data. This allowed our data mining framework to be fairly compared to traditional scanners over new data.

To quantitatively express the performance of our method we show tables with the counts for *true positives*, *true negatives*, *false positives*, and *false negatives*. A true positive (TP) is an malicious example that is correctly tagged as malicious, and a true negative (TN) is a benign example that is correctly classified. A false positive)(FP) is a benign program that has been mislabeled by an algorithm as a malicious program, while a false negative (FN) is a malicious executable that has been mis-classified as a benign program.

The results that are presented in this paper are the false positive rate and the detection rate. The false positive rate is the number of benign examples that are mislabeled as malicious divided by the total number of benign examples. The detection rate is the number of malicious examples that are caught divided by the total number of malicious examples.

2.1 Dataset Description

The data set consisted of a total of 4,301 programs split into 3,301 malicious binaries and 1,000 clean programs. There were no duplicate programs in the data set and every example in the set is labeled either malicious or benign by the commercial virus scanner. All labels are assumed to be correct.

All programs were gathered either from the FTP sites, or personal computers in the Data Mining Lab here at Columbia University.

The malicious executables were downloaded from various FTP sites and were labeled by a commercial virus scanner with the correct class label (malicious or benign) for our method. 5% of the data set was composed of Trojans and the other 95% consisted of viruses.Most of the clean programs were gathered from a freshly installed Windows 98 machine running MSOffice 97 while others are small executables downloaded from the Internet.

We also examined a subset of the data that was in *PE* [15], Portable Executable, format. The data set consisting of PE format executables was composed of 206 benign programs and 38 malicious executables.

After verification of the data set the next step of our method was to extract features from the programs.

3 Feature Extraction

In this section we detail all of our choices of features. We statically extracted different features that represented different information contained within each binary. These features were then used by the algorithms to generate detection models.

We first examine only the subset of PE executables using LibBFD, which is explained next. Then we used more general methods to extract features from all types of binaries.

3.1 LibBFD

Our first intuition into the problem was to extract information from the binary that would dictate its behavior. The problem of predicting a program's behavior is reduceable to the halting problem and hence undecidable [1]. Perfectly predicting a program's behavior is unattainable but estimating what a program can or cannot do is possible. For instance if a Windows executable does not call the User Interfaces DLL (USER32.DLL), then we could assume that the program does not have the standard Windows user interface. This is of course an over-simplification of the problem because the author of that example could have written or linked to another user interface library, but it did provide us with some insight to an optimal feature set.

To extract resource information from Windows executables we used GNU's Bin–Utils [4]. GNU's Bin–Utils suite of tools can analyze PE binaries within Windows. In PE, or COFF (Common Object File Format), program headers are composed of a COFF header, an Optional header, an MS-DOS stub, and a file signature. From the PE header we used libBFD, a library within Bin–Utils, to extract information in *object format*. Object format for a PE binary gives the file size, the names of Dynamically Linked Libraries (DLLs), and the names of function calls within those DLLs and Relocation Tables. From the object format, we extracted a set of features to compose a feature vector for each binary.

To understand how resources affected a binary's behavior we performed our experiments using three types of features:

1. The list of DLLs used by the binary

2. The list of DLL function calls made by the binary

3. The number of different system calls used within each DLL

The first approach to binary profiling (seen in Figure 1) used the DLLs loaded by the binary as features. The feature vector comprised of 30 boolean values representing whether or not a binary used a DLL. Typically, not every DLL was used in all of the binaries, but a majority of the binaries called the same resource such as GDI32.DLL which almost every binary called. GDI32.DLL is the Windows NT Graphics Device Interface and is a core component of WinNT.

$$\neg advapi32 \wedge avicap32 \wedge ... \wedge winmm \wedge \neg wsock32$$

Figure 1: First Feature Vector: A conjunction of DLL names

The example vector given in Figure 1 is composed of at least two unused resources: ADVAPI32.DLL, the Advanced Windows API, and WSOCK32.DLL, the Windows Sockets API. It also uses at least two resources: AVICAP32.DLL, the AVI capture API, and WINNM.DLL, the Windows Multimedia API.

The second approach to binary profiling (seen in Figure 2) used DLLs and their function calls as features. This approach was similar to the first, but with added function call information. The feature vector was composed of 2,229 boolean values. Because some of the DLL's had the same function names it was important to record which DLL the function came from.

$$advapi32.AdjustTokenPrivileges() \wedge advapi32.GetFileSecurityA() \wedge ... \wedge wsock32.recv() \wedge wsock32.send()$$

Figure 2: Second Feature Vector: A conjunction of DLL's and the functions called inside each DLL

The example vector given in Figure 2 is composed of at least four resources. Two functions were called in ADVAPI32.DLL: AdjustTokenPrivileges() and GetFileSecurityA(), and two functions in WSOCK32.DLL: recv() and send().

The third approach to binary profiling (seen in Figure 3) counted the number of different function calls used within each DLL. The feature vector included 30 integer values. This profile gives a rough measure of how heavily a DLL is used within a specific binary. Intuitively, in the resource models we have been exploring, this is a macro-resource usage model because the number of calls to each resource is counted instead of detailing referenced functions.

$$advapi32 = 2 \wedge avicap32 = 10 \wedge ... \wedge winmm = 8 \wedge wsock32 = 0$$

Figure 3: Third Feature Vector: A conjunction of DLL's and a count of the number of functions called inside each DLL

The example vector given in Figure 3 describes an example that calls two functions in ADVAPI32.DLL, ten functions in AVICAP32.DLL, eight functions in WINNM.DLL and no functions from WSOCK32.DLL.

All of the information about the binary was obtained from the program header. In addition, the information was obtained without executing the unknown program but by examining the static properties of the binary, using libBFD.

Since we could not analyze the entire dataset with libBFD we found another method for extracting features that works over the entire dataset. We describe that method next.

3.2 GNU Strings

During the analysis of our libBFD method we noticed that headers in PE-format were in plain text. This meant that we could extract the same information from the PE-executables by just extracting the plain text headers. We also noticed that non-PE executables also have strings encoded in them. We theorized that we could use this information to classify the full 4,301 item data set instead of the small libBFD data set.

To extract features from the first data set of 4,328 programs we used the GNU *strings* program. The strings program extracts consecutive printable characters from any file. Typically there are many printable strings in binary files. Some common strings found in our dataset are illustrated in Table 1.

kernel	microsoft	windows	getmodulehandlea
getversion	getstartupinfoa	win	getmodulefilenamea
messageboxa	closehandle	null	dispatchmessagea
library	getprocaddress	advapi	getlasterror
loadlibrarya	exitprocess	heap	getcommandlinea
reloc	createfilea	writefile	setfilepointer
application	showwindow	time	regclosekey

Table 1: Common strings extracted from binaries using GNU strings

Through testing we found that there were similar strings in malicious executables which distinguished them from clean programs, and similar strings in benign programs which distinguished them from malicious executables. The strings contained in a binary may consist of re-used code fragments, author signatures, file names, system resource information, etc. This method of detecting malicious executables is already used by the anti-malicious executable community to create signatures for malicious executables.

Extracted strings from an executable are not very robust as features because they can be changed easily, so we analyzed another feature, byte sequences.

3.3 Byte Sequences using Hexdump

Byte sequences are the last feature that we used, and they also analyzed the entire 4,301 member data set. We used *hexdump* [16], a tool that transforms binary files into hexadecimal files. The byte sequence feature is the most informative because it represents the machine code in an executable instead of resource information like libBFD features. Secondly, analyzing the entire binary gives more information for non-PE format executables than the strings method.

After we generated the hexdumps we had features in the form of Figure 4 where each line represents a short sequence of machine code instructions.

```
1f0e 0eba b400 cd09 b821 4c01 21cd 6854
7369 7020 6f72 7267 6d61 7220 7165 6975
6572 2073 694d 7263 736f 666f 2074 6957
646e 776f 2e73 0a0d 0024 0000 0000 0000
454e 3c05 026c 0009 0000 0000 0302 0004
0400 2800 3924 0001 0000 0004 0004 0006
000c 0040 0060 021e 0238 0244 02f5 0000
0001 0004 0000 0802 0032 1304 0000 030a
```

Figure 4: Example Hexdump

We again assumed that there were similar instructions in malicious executables that differentiated them from benign programs, and the class of benign programs had similar byte code that differentiated them from the malicious executables.

4 Algorithms

In this section we describe all the algorithms used in this paper including the signature based method that we use to compare our data mining algorithms to. We used three different data mining algorithms to generate classifiers using different features: RIPPER, Naive Bayes, and a Multi-Classifier system.

We detail the signature based method first.

4.1 Signature Methods

We examine signature-based methods to compare our results to traditional anti-virus methods. Signature-based detection methods are the most commonly used algorithms in the industry [25]. These signatures are picked to differentiate one malicious executable from another, and from benign programs. These signatures can generated by an expert in the field or an automatic method. Typically a signature is picked to illustrate the unusual properties of a specific malicious executables.

We implemented a signature-based scanner with this method. First we calculated the byte-sequences that were only found in the malicious executable class. These byte-sequences were then concatenated together to make a unique signature for each malicious executable example. Thus each malicious executable signature contained only byte-sequences found in the malicious executable class. To make the signature unique the byte-sequences found in each example were concatenated together to form one signature. This was because a byte-sequence that is only found in one class during training could possibly be found in the other class during testing [7], and lead to false positives in testing.

The method described above for the commercial scanner was never intended to detect unknown malicious binaries, but the data mining algorithms that follow were built to detect new malicious executables.

4.2 RIPPER

The next algorithm we used, RIPPER [2] is an inductive rule learner. This algorithm generated a detection model composed of resource rules that was built to detect future examples of malicious executables. This algorithm used libBFD information as features.

RIPPER is a rule-based learner that builds a set of rules that identify the classes while minimizing the amount of error. The error is defined by the number of training examples misclassified by the rules.

An inductive rule learner works as follows. An inductive algorithm learns what a malicious executable is given a set of training examples. The four features seen in Table 2 are:

1. "Does it have a GUI?"
2. "Does it perform a malicious function?"
3. "Does it compromise system security?"
4. "Does it delete files?"

and finally the class question "Is it a malicious executable?".

Has a GUI?	**Malicious Function?**	**Compromise Security?**	**Deletes Files?**	**Is it a malicious executable?**
yes	yes	yes	no	yes
no	yes	yes	yes	yes
yes	no	no	yes	no
yes	yes	yes	yes	yes

Table 2: Example Inductive Training Set. Intuitively all malicious executables share the second and third feature, "yes" and "yes" respectively.

The defining property of any inductive learner is that no a priori assumptions have been made regarding the final concept. The inductive learning algorithm makes as its primary assumption that the data trained over is similar in some way to the unseen data.

A hypothesis generated by an inductive learning algorithm for this learning problem has four attributes. Each attribute will have one of these values:

1. $\top$, truth, indicating any value is acceptable in this position,
2. a value, either yes, or no, is needed in this position, or
3. a $\bot$, falsity, indicating that no value is acceptable for this position

For example, the hypothesis $\langle \top, \top, \top, \top \rangle$ and the hypothesis $\langle yes, yes, yes, no \rangle$ would make the first example true. $\langle \top, \top, \top, \top \rangle$ would make any feature set true and $\langle yes, yes, yes, no \rangle$ is the set of features for example one.

The algorithm we describe is *Find-S* [18]. Find-S finds the most specific hypothesis that is consistent with the training examples. For a positive training example the algorithm replaces any attribute in the hypothesis that is inconsistent with the training example with a more general attribute. Of all the hypothesis values 1 is more general than 2 and 2 is more general than 3. For a negative example the algorithm does nothing. Positive examples in this problem are defined to be the malicious executables and negative examples are the benign programs.

The initial hypothesis that Find-S starts with is $\langle \perp, \perp, \perp, \perp \rangle$. This hypothesis is the most specific because it is true over the fewest possible examples, none. Examining the first positive example in Table 2, $\langle yes, yes, yes, no \rangle$, the algorithm chooses the next most specific hypothesis $\langle yes, yes, yes, no \rangle$. The next positive example, $\langle no, no, no, yes \rangle$, is inconsistent with the hypothesis in its first and fourth attribute ("Does it have a GUI?" and "Does it delete files?") and those attributes in the hypothesis get replaced with the next most general attribute, $\top$.

The resulting hypothesis after two positive examples is $\langle \top, yes, yes, \top \rangle$. The algorithm skips the third example, a negative example, and finds that this hypothesis is consistent with the final example in Table 2. The final rule for the training data listed in Table 2 is $\langle \top, yes, yes, \top \rangle$. The rule states that the attributes of a malicious executable, based on training data, are that has a malicious function and compromises system security. This is consistent with the definition of a malicious executable we gave in the introduction. It does not matter in this example if a malicious executable deletes files, or if it has a GUI or not.

Find-S is a relatively simple algorithm while RIPPER is more complex. RIPPER looks at both positive and negative examples to generate a set of hypotheses that more closely approximate the target concept while Find-S generates one hypothesis that approximates the target concept.

4.3 Naive Bayes

The next classifier we describe is a Naive Bayes classifier [5]. The naive Bayes classifier computes the likelihood that a program is malicious given the features that are contained in the program. This method used both strings and byte-sequence data to compute a probability of a binary's maliciousness given its features.

Nigam et al. [19] performed a similar experiment when they classified text documents according to which newsgroup they originated from. In this method we treated each executable's features as a text document and classified based on that. The main assumption in this approach is that the binaries contain similar features such a signatures, machine instructions, etc.

Specifically, we wanted to compute the class of a program given that the program contains a set of features F. We define C to be a random variable over the set of classes: *benign*, and *malicious* executables. That is, we want to compute $P(C|F)$, the probability that a program is in a certain class given the program contains the set of features F. We apply Bayes rule and express the probability as:

$$P(C|F) = \frac{P(F|C) * P(C)}{P(F)} \tag{1}$$

To use the naive Bayes rule we assume that the features occur independently to one another. If the features of a program F include the features $F_1, F_2, F_3, ..., F_n$, then equation (1) becomes:

$$P(C|F) = \frac{\prod_{i=1}^{n} P(F_i|C) * P(C)}{\prod_{j=1}^{n} P(F_j)} \tag{2}$$

Each $P(F_i|C)$ is the percentage of the time that the string F_i occurs in a program of class C. $P(C)$ is the proportion of the class C in the entire set of programs.

The output of the classifier is the highest probability class for a given set of strings. Since the denominator of (1) is the same for all classes we take the maximum class over all classes C of the probability of each class computed in (2) to get:

$$\text{Most Likely Class} = \max_{C} \left(P(C) \prod_{i=1}^{n} P(F_i|C) \right) \tag{3}$$

Most Likely Class is the class in C with the highest probability and hence the most likely classification of the example with features F.

To train the classifier, we recorded how many programs in each class contained each unique feature. We used this information to classify a new program into an appropriate class. We first used feature extraction to determine the features contained in the program. Then we applied equation (3) to compute the most likely class for the program.

We used the Naive Bayes algorithm and computed the most likely class for byte sequences, and strings.

4.4 Multi-Naive Bayes

The next data mining algorithm we describe is Multi-Naive Bayes. This algorithm was essentially a collection of Naive Bayes algorithms that voted on an overall classification for an example. Each Naive Bayes algorithm classified the examples in the test set as malicious or benign and this counted as a vote. The votes were combined by the Multi-Naive Bayes algorithm to output a final classification for all the Naive Bayes.

This method was required because even using a machine with one gigabyte of RAM, the size of the binary data was too large to fit into memory. The Naive Bayes algorithm required a table of all strings or bytes to compute its probabilities. To correct this problem we divided the problem into smaller pieces that would fit in memory and trained a Naive Bayes algorithm over each of the subproblems.

The subproblem was to classify based on every 6th line of machine code instead of every line in the binary. For this we trained six Naive Bayes classifiers so that every byte-sequence line in the training set had been trained over. In general, $NaiveBayes_i$ trained on every line where (i+6) mod 6 equaled 0.

The Multi-Naive Bayes promotes a vote of confidence between all of the underlying Naive Bayes classifiers. Each classifier gives a probability of a class C given a set of bytes F which the Multi-Naive Bayes uses to generate a probability for class C given F over all the classifiers.

We want to compute the likelihood of a class C given bytes F and the probabilities learned by all $NaiveBayes_i$. In equation (4) we compute the overall probability for $P_{NB}(C|F)$, the Multi-Naive Bayes probability of class C given a set of bytes F.

$$P_{NB}(C|F) = \prod_{i=1}^{|NB|} P_{NB_i}(C|F) * P_{NB_i}(C) \tag{4}$$

$P_{NB^i}(C|F)$ (generated from equation (2)) is the probability for class C computed by the classifier $NaiveBayes_i$ given F divided by the likely of a class C computed by $NaiveBayes_i$. These probabilities were multiplied together to compute $P_{NB}(C|F)$, the final probability of C given F. $|NB|$ is the size of the set NB such that $\forall NB_i \epsilon NB$.

The output of the multi-classifier given a set of bytes F is the class of highest probability over the classes given the probabilities of the underlying Naive Bayes classifiers and $P_{NB}(C)$ the prior probability of a given class.

$$\text{Most Likely Class} = \max_{C} \left(P_{NB}(C) * P_{NB}(C|F)\right) \tag{5}$$

Most Likely Class is the class in C with the highest probability hence the most likely classification of the example with features F.

5 Rules

Each data mining algorithm generated its own rule set to evaluate new examples. These detection models were the final result of our experiments. Each algorithm's rule set could be incorporated into a scanner to detect malicious programs. The generation of the rules only needed to be done periodically and the rule set distributed in order to detect new malicious executables.

5.1 RIPPER

RIPPER's rules were built to generalize over unseen examples so the rule set was more compact than the signature based methods. For the data set that contained 3,301 malicious executables the RIPPER rule set contained the five rules in Table 5.

$malicious := \neg user32.EndDialog() \wedge kernel32.EnumCalendarInfoA()$
$malicious := \neg user32.LoadIconA() \wedge \neg kernel32.GetTempPathA() \wedge \neg advapi32.*$
$malicious := shell32.ExtractAssociatedIconA()$
$malicious := msvbvm.*$
$benign :- otherwise$

Figure 5: Sample Classification Rules using features found in Figure 2

Here, a malicious executable was consistent with one of four hypotheses:

1. didn't call user32.EndDialog() but did call kernel32.EnumCalendarInfoA()
2. didn't call user32.LoadIconA(), kernel32.GetTempPathA(), or any function in advapi32.dll
3. called shell32.ExtractAssociatedIconA(),
4. called any function in msvbbm.dll, the Microsoft Visual Basic Library

A binary is labeled benign if it is inconsistent with all of the malicious binary hypotheses in Figure 5.

5.2 Naive Bayes

The Naive Bayes rules were more complicated than the RIPPER, and signature based hypotheses. These rules took the form of $P(F|C)$, the probability of an example F given a class C. The probability for a string occurring in a class is the total number of times it occurred in that class's training set divided by the total number of times that the string occurred over the entire training set. These hypotheses are illustrated in Figure 6.

$P(\text{“}windows\text{”}|benign) = 45/47$
$P(\text{“}windows\text{”}|maliciousexecutable) = 2/47$
$P(\text{“}*.COM\text{”}|benign) = 1/12$
$P(\text{“}*.COM\text{”}|maliciousexecutable) = 11/12$

Figure 6: Sample classification rules found by Naive Bayes.

Here, the string "windows" was predicted to more likely to occur in a benign program and string "*.COM" was more than likely in a malicious executable program.

However this leads to a problem when a string (e.g. "CH2OH-CHOH-CH2OH") only occured in one set, for example only in the malicious executables. The probability of "CH20H-CHOH-CH20H" occurring in any future benign example is predicted to be 0, but this is an incorrect assumption. If a Chemistry TA's program was built to print out "CH20H-CHOH-CH20H", glycerol, it will always be tagged a malicious executable even if it has other strings in it that would have labeled it benign.

In Figure 6 the string "*.COM" does not occur in any benign programs so the probability of "*.COM" occurring in class benign is approximated to be 1/12 instead of 0/11. This approximates real world probability that any string could occur in both classes even if during training it was only seen in one class [7].

5.3 Multi-Naive Bayes

The Multi-Naive Bayes algorithm generated rule sets are illustrated in Figure 7. These rules took the form of probabilities of the collected underlying Naive Bayes classifiers. Each estimated probability, $P_{NB}(C|X)$, of a class, C, given a binary program, X, is the product of all underlying Naive Bayes, $P_{NB_i}(C_i|X)$, for C_i equal to C.

$P_{NB}(benign|X) = \prod_{i=1}^{|NB|} P_{NB_i}(benign|X) = 45/47 * 33/40 * 30/35 * 15/16 * 10/40 * 24/36 = 10\%$
$P_{NB}(maliciousexecutable|X) = \prod_{i=1}^{|NB|} P_{NB_i}(maliciousexecutable|X) = 2/47 * 7/40 * 5/35 * 1/16 * 30/40 * 12/36 = 0.01\%$
$P_{NB}(benign|Y) = \prod_{i=1}^{|NB|} P_{NB_i}(benign|Y) = 5/50 * 2/20 * 3/300 * 15/30 * 1/22 * 2/3 = 1.5 * 10^{-6}\%$
$P_{NB}(maliciousexecutable|Y) = \prod_{i=1}^{|NB|} P_{NB_i}(maliciousexecutable|Y) = 45/50 * 18/20 * 297/300 * 15/30 * 21/22 * 1/3 = 12\%$

Figure 7: Classification rules found by Multi-Naive Bayes for examples X and Y.

In Figure 7, X was predicted by the Multi-Naive Bayes algorithm to be more likely to be a benign program because six of the six underlying Naive Bayes algorithms classified it as most likely benign. They predicted this class with likelyhoods of 45/47, 33/40, 30/35, 15/16, 10/40 and 24/36 respectively. Likewise they classified X as malicious with respective probabilities of 2/47, 7/40, 5/35, 1/16, 30/40, and 12/36.

Profile Type	True Positives (TP)	True Negatives (TN)	False Positives (FP)	False Negatives (FN)	Detection Rate	False Positive Rate	Overall Accuracy
Signature Method							
— Bytes	1121	1000	0	2180	33.96%	0%	49.31%
RIPPER							
— DLLs used	22	187	19	16	57.89%	9.22%	83.62%
— DLL function calls	27	190	16	11	71.05%	7.77%	89.36%
— DLLs with counted function calls	20	195	11	18	52.63%	5.34%	89.07%
Naive Bayes							
— Strings	3176	960	41	89	97.43%	3.80%	97.11%
Multi-Naive Bayes							
— Bytes	3191	940	61	74	97.76%	6.01%	96.88%

Table 3: These are the results of classifying new malicious programs organized by algorithm and feature. Multi-Naive Bayes using Bytes had the highest Detection Rate, and Signature Method with Strings had the lowest False Positive Rate. Highest overall accuracy was the Naive Bayes algorithm with Strings. Note that the detection rate for the signature-based methods are lower than the data mining methods.

6 Results and Analysis

We estimate our results over new data by using 5-fold cross validation [10]. Cross validation is the standard method to estimate likely predictions over unseen data in Data Mining. For each set of binary profiles we partitioned the data into 5 equal size partitions. We used 4 of the partitions for training and then evaluated the rule set over the remaining partition. Then we repeated the process 5 times leaving out a different partition for testing each time. This gave us a very reliable measure of our method's accuracy over unseen data. We averaged the results of these five tests to obtain a good measure of how the algorithm performs over the entire set.

To evaluate our system we were interested in several quantities:

1. True Positives (TP), the number of malicious executable examples classified as malicious executables
2. True Negatives (TN), the number of benign programs classified as benign.
3. False Positives (FP), the number of benign programs classified as malicious executables
4. False Negatives (FN), the number of malicious executables classified as benign binaries.

We were interested in the detection rate of the classifier. In our case this was the percentage of the total malicious programs labeled malicious. We were also interested in the false positive rate. This was the percentage of benign programs which were labeled as malicious, also called false alarms.

The Detection Rate is defined as $\frac{TP}{TP+FN}$, False Positive Rate as $\frac{FP}{TN+FP}$, and Overall Accuracy as $\frac{TP+TN}{TP+TN+FP+FN}$. The results of all experiments are presented in Table 3.

For all the algorithms we plotted the detection rate vs. false positive rate using *ROC* curves [9]. ROC (Receiver Operating Characteristic) curves are a way of visualizing the trade-offs between detection and false positive rates.

6.1 Signature Method

As is shown in Table 3, the signature method had the lowest false positive rate, 0% This algorithm also had the lowest detection rate, 33.96%, and accuracy rate, 49.31%.

Since we use this method to compare with the learning algorithms we plot its ROC curves against the RIPPER algorithm in Figure 8 and against the Naive Bayes and Multi-Naive Bayes algorithms in Figure 9.

6.2 RIPPER

The RIPPER results shown in Table 3 are equivalent to each other in detection rates and overall accuracy, but the method using features from Figure 2, a list of DLL function calls, has a higher detection rate.

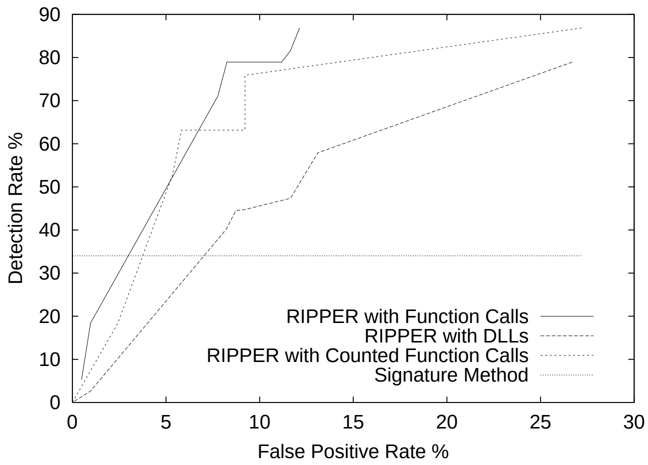

Figure 8: RIPPER ROC. Notice that the RIPPER curves have a higher detection rate than the comparison method with false-positive rates greater than 7%.

The ROC curves for all RIPPER variations are shown in Figure 8. The lowest line represents RIPPER using DLLs only as features, and it was roughly linear in its growth. This means that as we increase detection rate by 5% the false positive would also increase by roughly 5%.

The other lines are concave down so there was an optimal trade-off between detection and false alarms. For DLL's with Counted Function Calls this optimal point was when false positive rate was 10% and detection rate was equal to 75%. For DLLs with Function Calls the optimal point was when false positive rate was 12% and detection rate was less than 90%.

6.3 Naive Bayes

The Naive Bayes algorithm using strings as features performed the best out of the learning algorithms and better than the signature method in terms of false positive rate and overall accuracy (see Table 3). It is the most accurate algorithm with 97.11% and within 1% of the highest detection rate, Multi-Naive Bayes with 97.76%. It performed better than the RIPPER methods in every category.

In Figure 9, the slope of the Naive Bayes curve is initially much steeper than the Multi-Naive Bayes. The Naive Bayes with strings algorithm has better detection rates for small false positive rates. Its results were greater than 90% accuracy with a false positive rate less than 2%.

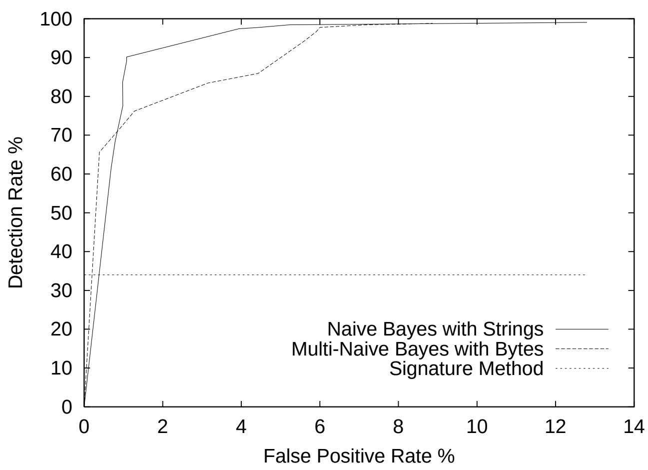

Figure 9: Naive Bayes and Multi-Naive Bayes ROC. Note that the Naive Bayes and Multi-Naive Bayes methods have higher detection rate than the signature method with a greater than 0.5% false positive rate.

6.4 Multi-Naive Bayes

The Multi-Naive Bayes algorithm using bytes as features had the highest detection rate out of any method we tested, 97.76%. The false positive rate at 6.01% was higher than the Naive Bayes methods (3.80%) and the signature methods ($< 1\%$).

The ROC curves in Figure 9 show a slower growth than the Naive Bayes with strings method until the false positive rate climbed above 4%. Then the two algorithms converged for false positive rates greater than 6% with a detection rate greater than 95%.

7 Defeating Detection Models

Although these methods can detect new malicious executables, if the detection model were to be compromised a malicious executable author could bypass detection.

First, to defeat the signature based method requires removing all malicious signatures from the binary. Since these are typically a subset of a malicious executable's total data, changing the signature of a binary would be possible although difficult.

Defeating the models generated by RIPPER would require generating functions that would change the resource usage. These functions do not have to be called by the binary but would change the resource signature of an executable.

To defeat our implementation of the Naive Bayes classifier it would be necessary to change a significant number of features in the example. One way this can be done is through encryption, but encryption will add overhead to small malicious executables.

We corrected the problem of authors evading a strings-based rule set by initially classifying each example as malicious. If no strings that were contained in the binary had ever been trained over then the final class was malicious. If there were strings contained in the program that the algorithm had seen before then the probabilities were computed normally according to the Naive Bayes rule from Section 4.3. This took care of the instance where a binary had encrypted strings, or had changed all of its strings.

The Multi-Naive Bayes method improved on these results because changing every line of byte code in the Naive Bayes detection model would be an even more difficult proposition than changing all the strings. Changing this many of the lines in a program would change the binary's behavior significantly. Removing all lines of code that appear in our model would be difficult and time consuming, and even then if none of byte sequences in the example had been trained over then the example would be initially classified as malicious.

The Multi-Naive Bayes is a more secure model of detection than any method discussed in this paper because we evaluate a binary's entire instruction set whereas signature methods looks for segments of byte sequences. It is much easier for malicious program authors to modify the lines of code that a signature represents than to change all the lines contained in the program to evade a Naive Bayes or Multi-Naive Bayes model. The byte sequence model is the most secure model we devised in our test.

8 Conclusions

The first contribution that we presented in this paper was a method for detecting previously undetectable malicious executables. We showed this by comparing our results with traditional signature based methods and with other learning algorithms. The Multi-Naive Bayes method had the highest accuracy and detection rate of any algorithm over unknown programs, 97.76%, over double the detection rates of signature-based methods. Its rule set was also more difficult to defeat than other methods because all lines of machine instructions would have to be changed to avoid detection.

The first problem with traditional anti-malicious executable methods is that in order to detect a new malicious executable, the program needs to be examined and a signature extracted from it and included in the anti-malicious executable software database. The difficulty with this method is that during the time required for a malicious program to be identified, analyzed and signatures to be distributed, systems are at risk from that program. Our methods may provide a defense during that time. With a low false positive rate the inconvenience to the end user would be minimal while providing ample defense during the time before an update of models is available.

Virus Scanners are updated about every month. 240–300 new malicious executables are created in that time (8–10 a day [25]). Our method would catch roughly 216–270 of those new malicious executables without the need for an update whereas traditional methods would catch only 87–109. Our method more than doubles the detection rate of signature based methods.

The methods discussed in this paper are being implemented as a network mail filter. We are implementing an network-level email filter that uses our algorithms to catch malicious executables before users receive them through their mail. We can either wrap the potential malicious executable or we can block it. This has the potential to stop some malicious executables in the network and prevent DoS (Denial of Service) attacks by malicious executables. If a malicious binary accesses a user's address book and mails copies of itself out over the network, eventually most users of the LAN will clog the network by sending each other copies of the same malicious executable. This is very similar to the old Internet Worm attack. Stopping the malicious executables from replicating on a network level would be very advantageous.

Since both the Naive Bayes and Multi-Naive Bayes methods are probabilistic we can also tell if a binary is *borderline*. A borderline binary is a program that has similar probabilities for both classes (i.e. could be a malicious executable or a benign program). If it is a border-line case we have an option in the network filter to send a copy of the malicious executable to a central repository such as CERT. There, it can be examined by human experts.

8.1 Future Work

Future work involves extending our learning algorithms to better utilize byte-sequences. Currently the Multi-Naive Bayes method learns over sequences of set length, but we theorize that rules with higher accuracy and detection rates could be learned over variable length sequences. There are some algorithms such as Sparse Markov Transducers [6] that can determine how long a sequence of bytes should be for optimal classification.

We are planning to implement the system on a network of computers to evaluate its performance in terms of time and accuracy in real world environments. We also would like to make the learning algorithms more efficient in time and space. Currently, the Naive Bayes methods have to be run on a computer with one gigabyte of RAM.

References

[1] Fred Cohen. *A Short Course on Computer Viruses*. ASP Press, 1990.

[2] William Cohen. Learning Trees and Rules with Set-Valued Features. *American Association for Artificial Intelligence (AAAI)*, 1996.

[3] R. Crawford, P. Kerchen, K. Levitt, R. Olsson, M. Archer, and M. Casillas. Automated Assistance for Detecting Malicious Code. *Proceedings of the 6th International Computer Virus and Security Conference*, 1993.

[4] Cygnus. GNU Binutils Cygwin. *Online publication*, 1999. http://sourceware.cygnus.com/cygwin.

[5] D.Michie, D.J.Spiegelhalter, and C.C.TaylorD. *Machine learning of rules and trees. In Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994.

[6] Eleazar Eskin, William Noble Grundy, and Yoram Singer. Protein Family Classification using Sparse Markov Transducers. *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, 2000.

[7] Jeffrey O. Kephart and William C. Arnold. Automatic Extraction of Computer Virus Signatures. *4th Virus Bulletin International Conference*, pages 178-184, 1994.

[8] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, and R. Olsson. Static Analysis Virus Detection Tools for UNIX Systems. *Proceedings of the 13th National Computer Security Conference*, pages 350–365, 1990.

[9] Zou KH, Hall WJ, and Shapiro D. Smooth non-parametric ROC curves for continuous diagnostic tests. *Statistics in Medicine*, 1997.

[10] R Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *IJCAI*, 1995.

[11] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from UNIX processes execution traces for intrusion detection. *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press, 1997.

[12] Wenke Lee, Sal Stolfo, and Kui Mok. A Data Mining Framework for Building Intrusion Detection Models. *IEEE Symposium on Security and Privacy*, 1999.

[13] R.W. Lo, K.N. Levitt, and R.A. Olsson. MCF: a Malicious Code Filter. *Computers & Security*, **14**(6):541–566., 1995.

[14] MacAfee. Homepage - MacAfee.com. *Online publication*, 2000. http://www.mcafee.com.

[15] Microsoft. Portable Executable Format. *Online publication*, 1999. http://support.microsoft.com /support/kb/articles/Q121/4/60.asp.

[16] Peter Miller. Hexdump. *Online publication*, 2000. http://www.pcug.org.au/ millerp/hexdump.html.

[17] MIT Lincoln Labs. 1999 DARPA intrusion detection evaluation.

[18] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[19] Kamal Nigam, Andrew McCallum, Sebastian Thrun, and Tom Mitchell. Learning to Classify Text from Labeled and Unlabled Documents. *AAAI-98*, 1998.

[20] Wildlist Organization. Virus descriptions of viruses in the wild. *Online publication*, 2000. http://www.f-secure.com/virus-info/wild.html.

[21] REUTERS. Microsoft Hack Shows Companies Are Vulnerable. *New York Times*, October 29, 2000.

[22] Eugene H. Spafford. The Internet worm program: an analysis. *Tech. Report CSD–TR–823*, 1988. Department of Computer Science, Purdue University.

[23] Gerald Tesauro, Jeffrey O. Kephart, and Gregory B. Sorkin. Neural Networks for Computer Virus Recognition. *IEEE Expert*, **11**(4):5–6. IEEE Computer Society, August, 1996.

[24] Steve R. White. Open Problems in Computer Virus Research. *Virus Bulletin Conference*, 1998.

[25] Steve R. White, Morton Swimmer, Edward J. Pring, William C. Arnold, David M. Chess, and John F. Morar. Anatomy of a Commercial-Grade Immune System. *IBM Research White Paper*, 1999. http://www.av.ibm.com/ScientificPapers/White/ Anatomy/anatomy.html.