Journal of Machine Learning Research 7 (2006) 2721-2744 Submitted 3/06; Revised 9/06; Published 12/06

Learning to Detect and Classify Malicious Executables in the Wild*

J. Zico Kolter KOLTER@CS.STANFORD.EDU
Department of Computer Science
Stanford University
Stanford, CA 94305-9025, USA

Marcus A. Maloof MALOOF@CS.GEORGETOWN.EDU
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA

Editor: Richard Lippmann

Abstract

We describe the use of machine learning and data mining to detect and classify malicious executables as they appear in the wild. We gathered $1,971$ benign and $1,651$ malicious executables and encoded each as a training example using *n*-grams of byte codes as features. Such processing resulted in more than 255 million distinct *n*-grams. After selecting the most relevant *n*-grams for prediction, we evaluated a variety of inductive methods, including naive Bayes, decision trees, support vector machines, and boosting. Ultimately, boosted decision trees outperformed other methods with an area under the ROC curve of 0.996. Results suggest that our methodology will scale to larger collections of executables. We also evaluated how well the methods classified executables based on the function of their payload, such as opening a backdoor and mass-mailing. Areas under the ROC curve for detecting payload function were in the neighborhood of 0.9, which were smaller than those for the detection task. However, we attribute this drop in performance to fewer training examples and to the challenge of obtaining properly labeled examples, rather than to a failing of the methodology or to some inherent difficulty of the classification task. Finally, we applied detectors to 291 malicious executables discovered after we gathered our original collection, and boosted decision trees achieved a true-positive rate of 0.98 for a desired false-positive rate of 0.05. This result is particularly important, for it suggests that our methodology could be used as the basis for an operational system for detecting previously undiscovered malicious executables.

Keywords: data mining, concept learning, computer security, invasive software

1. Introduction

Malicious code is "any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function" (McGraw and Morisett, 2000, p. 33). Such software has been used to compromise computer systems, to destroy their information, and to render them useless. It has also been used to gather information, such as passwords and credit card numbers, and to distribute information, such as pornography, all without the knowledge of a system's

*. This work is based on an earlier work: Learning to Detect Malicious Executables in the Wild, in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, © ACM, 2004. `http://doi.acm.org/10.1145/1014052.1014105`.

©2006 J. Zico Kolter and Marcus A. Maloof.

Trend Micro, Inc.
EX1008-1

users. As more and more novice users obtain sophisticated computers with high-speed connections to the Internet, the potential for further abuse is great.

Malicious executables generally fall into three categories based on their transport mechanism: viruses, worms, and Trojan horses. Viruses inject malicious code into existing programs, which become "infected" and, in turn, propagate the virus to other programs when executed. Viruses come in two forms, either as an infected executable or as a virus loader, a small program that only inserts viral code. Worms, in contrast, are self-contained programs that spread over a network, usually by exploiting vulnerabilities in the software running on the networked computers. Finally, Trojan horses masquerade as benign programs, but perform malicious functions. Malicious executables do not always fit neatly into these categories and can exhibit combinations of behaviors.

Excellent technology exists for detecting known malicious executables. Software for virus detection has been quite successful, and programs such as McAfee Virus Scan and Norton AntiVirus are ubiquitous. Indeed, Dell recommends Norton AntiVirus for all of its new systems. Although these products use the word *virus* in their names, they also detect worms and Trojan horses.

These programs search executable code for known patterns, and this method is problematic. One shortcoming is that we must obtain a copy of a malicious program before extracting the pattern necessary for its detection. Obtaining copies of new or unknown malicious programs usually entails them infecting or attacking a computer system.

To complicate matters, writing malicious programs has become easier: There are virus kits freely available on the Internet. Individuals who write viruses have become more sophisticated, often using mechanisms to change or obfuscate their code to produce so-called *polymorphic viruses* (Anonymous, 2003, p. 339). Indeed, researchers have recently discovered that simple obfuscation techniques foil commercial programs for virus detection (Christodorescu and Jha, 2003). These challenges have prompted some researchers to investigate learning methods for detecting new or unknown viruses, and more generally, malicious code.

Our previous efforts to address this problem (Kolter and Maloof, 2004) resulted in a fielded prototype, built using techniques from machine learning (e.g., Mitchell, 1997) and data mining (e.g., Hand et al., 2001). The Malicious Executable Classification System (MECS) currently detects unknown malicious executables "in the wild," that is, as they would appear undetected on a user's hard drive, without preprocessing or removing any obfuscation. To date, we have gathered $1,971$ system and non-system executables, which we will refer to as "benign" executables, and $1,651$ malicious executables with a variety of transport mechanisms and payloads (e.g., key-loggers and backdoors). Although all were for the Windows operating system, it is important to note that our approach is not restricted to this operating system.

We extracted byte sequences from the executables, converted these into n-grams, and constructed several classifiers: IBk, naive Bayes, support vector machines (SVMs), decision trees, boosted naive Bayes, boosted SVMs, and boosted decision trees. In this domain, there is an issue of unequal but unknown costs of misclassification error, so we evaluated the methods using receiver operating characteristic (ROC) analysis (Swets and Pickett, 1982), using area under the ROC curve as the performance metric. Ultimately, boosted decision trees outperformed all other methods with an area under the curve of 0.996.

We delivered MECS to the MITRE Corporation, the sponsors of this project, as a research prototype, and it is being used in an operational environment. Users interact with MECS through a command line. They can add new executables to the collection, update learned models, display ROC curves, and produce a single classifier at a specific operating point on a selected ROC curve.

Trend Micro, Inc.
EX1008-2

In this paper, we build upon our previous work (Kolter and Maloof, 2004) by presenting results that suggest our estimates of the detection rate for malicious executables hold in an operational environment. To show this, we built classifiers from our entire collection, which we gathered early in the summer of 2003. We then applied all of the classifiers to 291 malicious executables discovered after we gathered our collection. Detection rates for three different false-positive rates corresponded to results we obtained through experimentation. Boosted decision trees, for example, achieved a detect rate of 0.97 for a desired false-positive rate of 0.05.

We also present results suggesting that one can use our methodology to classify malicious executables based on their payload's function. For example, from 520 malicious executables containing a mass-mailer, we were able to build a detector for such executables that achieved an area under the ROC curve of about 0.9. Results were similar for detecting malicious executables that open backdoors and that load viruses.

With this paper, we make three main contributions. We show how to use established methods of text classification to detect and classify malicious executables. We present empirical results from an extensive study of inductive methods for detecting and classifying malicious executables in the wild. We show that the methods achieve high detection rates, even on completely new, previously unseen malicious executables, which suggests this approach complements existing technologies and could serve as the basis for an operational system.

In the three sections that follow, we describe related work, our data collection, and the methods we applied. Then, in Sections 5–7, we present empirical results from three experiments. The first involved detecting malicious executables; the second, classifying malicious executables based on the function of their payload; and the third, evaluating fully trained methods on completely new, previously unseen malicious executables. Finally, before making concluding remarks, we discuss in Section 8 our results, challenges we faced, and other approaches we considered.

2. Related Work

There have been few attempts to use machine learning and data mining for the purpose of identifying new or unknown malicious code (e.g., Lo et al., 1995; Kephart et al., 1995; Tesauro et al., 1996; Schultz et al., 2001; Kolter and Maloof, 2004). These have concentrated mostly on PC viruses (Lo et al., 1995; Kephart et al., 1995; Tesauro et al., 1996; Schultz et al., 2001), thereby limiting the utility of such approaches to a particular type of malicious code and to computer systems running Microsoft's Windows operating system. Such efforts are of little direct use for computers running the UNIX operating system, for which viruses pose little threat. However, the methods proposed are general, meaning that they could be applied to malicious code for any platform, and presently, malicious code for the Windows operating system poses the greatest threat, mainly because of its ubiquity.

In an early attempt, Lo et al. (1995) conducted an analysis of several programs—evidently by hand—and identified *telltale signs*, which they subsequently used to filter new programs. While they attempted to extract patterns or signatures for identifying any class of malicious code, they presented no experimental results suggesting how general or extensible their approach might be. Researchers at IBM's T.J. Watson Research Center have investigated neural networks for virus detection (Kephart et al., 1995) and have incorporated a similar approach for detecting boot-sector viruses into IBM's Anti-virus software (Tesauro et al., 1996).

Trend Micro, Inc.
EX1008-3

Method	TP Rate	FP Rate	Accuracy (%)
Signature + `hexdump`	0.34	0.00	49.31
RIPPER + DLLs used	0.58	0.09	83.61
RIPPER + DLL function used	0.71	0.08	89.36
RIPPER + DLL function counts	0.53	0.05	89.07
Naive Bayes + `strings`	0.97	0.04	97.11
Voting Naive Bayes + `hexdump`	0.98	0.06	96.88

Table 1: Results from the study conducted by Schultz et al. 2001.

More recently, instead of focusing on boot-sector viruses, Schultz et al. (2001) used data mining methods, such as naive Bayes, to detect malicious code. The authors collected $4,301$ programs for the Windows operating system and used McAfee Virus Scan to label each as either malicious or benign. There were $3,301$ programs in the former category and $1,000$ in the latter. Of the malicious programs, 95% were viruses and 5% were Trojan horses. Furthermore, 38 of the malicious programs and 206 of the benign programs were in the Windows Portable Executable (PE) format.

For feature extraction, the authors used three methods: binary profiling, string sequences, and so-called *hex dumps*. The authors applied the first method to the smaller collection of 244 executables in the Windows PE format and applied the second and third methods to the full collection.

The first method extracted three types of resource information from the Windows executables: (1) a list of Dynamically Linked Libraries (DLLs), (2) function calls from the DLLs, and (3) the number of different system calls from each DLL. For each resource type, the authors constructed binary feature vectors based on the presence or absence of each in the executable. For example, if the collection of executables used ten DLLs, then they would characterize each as a binary vector of size ten. If a given executable used a DLL, then they would set the entry in the executable's vector corresponding to that DLL to one. This processing resulted in $2,229$ binary features, and in a similar manner, they encoded function calls and their number, resulting in 30 integer features.

The second method of feature extraction used the UNIX `strings` command, which shows the printable strings in an object or binary file. The authors formed training examples by treating the strings as binary attributes that were either present in or absent from a given executable.

The third method used the `hexdump` utility (Miller, 1999), which is similar to the UNIX octal dump (`od -x`) command. This printed the contents of the executable file as a sequence of hexadecimal numbers. As with the printable strings, the authors used two-byte words as binary attributes that were either present or absent.

After processing the executables using these three methods, the authors paired each extraction method with a single learning algorithm. Using five-fold cross-validation, they used RIPPER (Cohen, 1995) to learn rules from the training set produced by binary profiling. They used naive Bayes to estimate probabilities from the training set produced by the `strings` command. Finally, they used an ensemble of six naive-Bayesian classifiers on the `hexdump` data by training each on one-sixth of the lines in the output file. The first learned from lines 1, 6, 12, ...; the second, from lines 2, 7, 13, ...; and so on. As a baseline method, the authors implemented a signature-based scanner by using byte sequences unique to the malicious executables.

The authors concluded, based on true-positive (TP) rates, that the voting naive-Bayesian classifier outperformed all other methods, which appear with false-positive (FP) rates and accuracies in

Trend Micro, Inc.
EX1008-4

Table 1. The authors also presented ROC curves (Swets and Pickett, 1982), but did not report the areas under these curves. Nonetheless, the curve for the single naive Bayesian classifier appears to dominate that of the voting naive Bayesian classifier in most of the ROC space, suggesting that the best performing method was actually naive Bayes trained with strings.

However, as the authors discuss, one must question the stability of DLL names, function names, and string features. For instance, one may be able to compile a source program using another compiler to produce an executable different enough to avoid detection. Programmers often use methods to obfuscate their code, so a list of DLLs or function names may not be available.

The authors paired each feature extraction method with a learning method, and as a result, RIPPER was trained on a much smaller collection of executables than were naive Bayes and the ensemble of naive-Bayesian classifiers. Although results were generally good, it would have been interesting to know how the learning methods performed on all data sets. It would have also been interesting to know if combining all features (i.e., strings, bytes, functions) into a single training example and then selecting the most relevant would have improved the performance of the methods.

There are other methods of guarding against malicious code, such as *object reconciliation* (Anonymous, 2003, p. 370), which involves comparing current files and directories to past copies; one can also compare cryptographic hashes. One can also audit running programs (Soman et al., 2003) and statically analyze executables using predefined malicious patterns (Christodorescu and Jha, 2003). These approaches are not based on data mining, although one could imagine the role such techniques might play.

Researchers have also investigated classification methods for the determination of software authorship. Most notorious in the field of authorship are the efforts to determine whether Sir Frances Bacon wrote works attributed to Shakespeare (Durning-Lawrence, 1910), or who wrote the twelve disputed Federalist Papers, Hamilton or Madison (Kjell et al., 1994). Recently, similar techniques have been used in the relatively new field of *software forensics* to determine program authorship (Spafford and Weeber, 1993). Gray et al. (1997) wrote a position paper on the subject of authorship, whereas Krsul (1994) conducted an empirical study by gathering code from programmers of varying skill, extracting software metrics, and determining authorship using discriminant analysis. There are also relevant results published in the literature pertaining to the plagiarism of programs (Aiken, 1994; Jankowitz, 1988), which we will not survey here.

Krsul (1994) collected 88 programs written in the C programming language from 29 programmers at the undergraduate, graduate, and faculty levels. He then extracted 18 layout metrics (e.g., indentation of closing curly brackets), 15 style metrics (e.g., mean line length), and 19 structure metrics (e.g., percentage of `int` function definitions). On average, Krsul determined correct authorship 73% of the time. Interestingly, of the 17 most experienced programmers, he was able to determine authorship 100% of the time. The least experienced programmers were the most difficult to classify, presumably because they had not settled into a consistent style. Indeed, they "were surprised to find that one [programmer] had varied his programming style considerably from program to program in a period of only two months" (Krsul and Spafford, 1995, §5.1).

While interesting, it is unclear how much confidence we should have in these results. Krsul (1994) used 52 features and only one or two examples for each of the 20 classes (i.e., the authors). This seems underconstrained, especially when rules of thumb suggest that one needs ten times more examples than features (Jain et al., 2000). On the other hand, it may also suggest that one simply needs to be clever about what constitutes an example. For instance, one could presumably use functions as examples rather than programs, but for the task of determining authorship of malicious

Trend Micro, Inc.
EX1008-5

programs, it is unclear whether such data would be possible to collect or if it even exists. Fortunately, as we discuss in the next section, a lack of data was not a problem for our project.

3. Data Collection

As stated previously, the data for our study consisted of $1,971$ benign executables and $1,651$ malicious executables. All were in the Windows PE format. We obtained benign executables from all folders of machines running the Windows 2000 and XP operating systems. We gathered additional applications from SourceForge (`http://sourceforge.net`) and download.com (`http://www.download.com`).

We obtained virus loaders, worms, and Trojan horses from the Web site VX Heavens (`http://vx.netlux.org`) and from computer-forensic experts at the MITRE Corporation, the sponsors of this project. Some executables were obfuscated with compression, encryption, or both; some were not, but we were not informed which were and which were not. For one small collection, a commercial product for detecting viruses failed to identify 18 of the 114 malicious executables. For the entire collection of $1,651$ malicious executables, a commercial program failed to identify 50 as malicious, even though all were known and in the public domain. Note that, for viruses, we examined only the loader programs; we did not include infected executables in our study.

As stated previously, we gathered this collection early in the summer of 2003. Recently, we obtained 291 additional malicious executables from VX Heavens that have appeared after we took our collection. As such, they were not part of our original collection and were not part of our previous study (Kolter and Maloof, 2004). These additional executables were for a real-world, online evaluation, which we motivate and discuss further in Section 7.

We used the `hexdump` utility (Miller, 1999) to convert each executable to hexadecimal codes in an ASCII format. We then produced *n*-grams, by combining each four-byte sequence into a single term. For instance, for the byte sequence `ff 00 ab 3e 12 b3`, the corresponding *n*-grams would be `ff00ab3e`, `00ab3e12`, and `ab3e12b3`. This processing resulted in $255,904,403$ distinct *n*-grams. One could also compute *n*-grams from words, something we explored and discuss further in Section 5.2. Using the *n*-grams from all of the executables, we applied techniques from text classification, which we discuss further in the next section.

4. Classification Methodology

Our overall approach drew techniques from machine learning (e.g., Mitchell, 1997), data mining (e.g., Hand et al., 2001), and, in particular, text classification (e.g., Dumais et al., 1998; Sahami et al., 1998). We used the *n*-grams extracted from the executables to form training examples by viewing each *n*-gram as a Boolean attribute that is either present in (i.e., `T`) or absent from (i.e., `F`) the executable. We selected the most relevant attributes (i.e., *n*-grams) by computing the *information gain* (*IG*) for each:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C_i} P(v_j, C_i) \log \frac{P(v_j, C_i)}{P(v_j)P(C_i)} ,$$

where C_i is the ith class, v_j is the value of the jth attribute, $P(v_j, C_i)$ is the proportion that the jth attribute has the value v_j in the class C_i, $P(v_j)$ is the proportion that the jth *n*-gram takes the value v_j in the training data, and $P(C_i)$ is the proportion of the training data belonging to the class C_i. This measure is also called *average mutual information* (Yang and Pederson, 1997).

Trend Micro, Inc.
EX1008-6

We then selected the top 500 *n*-grams, a quantity we determined through pilot studies (see Section 5.2), and applied several learning methods, all of which are implemented in the Wakaito Environment for Knowledge Acquisition (WEKA) (Witten and Frank, 2005): IB*k*, naive Bayes, a support vector machine (SVM), and a decision tree. We also "boosted" the last three of these learners, and we discuss each of these methods in the following sections. Since the task is to detect malicious executables, in subsequent discussion, we refer to the malicious class as the positive class and refer to the benign class as the negative class.

4.1 Instance-based Learner

One of the simplest learning methods is the instance-based (IB) learner (Aha et al., 1991). Its concept description is a collection of training examples or instances. Learning, therefore, is the addition of new examples to the collection. To classify an unknown instance, the performance element finds the example in the collection most similar to the unknown and returns the example's class label as its prediction for the unknown. For Boolean attributes, such as ours, a convenient measure of similarity is the number of values two instances have in common. Variants of this method, such as IB*k*, find the *k* most similar instances and return the majority vote of their class labels as the prediction. Values for *k* are typically odd to prevent ties. Such methods are also known as *nearest neighbor* and k-*nearest neighbors*.

One can estimate a probability distribution from the nearest neighbors and their distances. For ROC analysis, we used the probability of the negative class as a *case rating*, which indicates the degree to which an example is negative. Such ratings paired with the true labels of the test cases are sufficient for estimating an ROC curve (Swets and Pickett, 1982), a matter we discuss further in Section 5.1.

4.2 Naive Bayes

Naive Bayes is a probabilistic method that has a long history in information retrieval and text classification (Maron and Kuhns, 1960). It stores as its concept description the prior probability of each class, $P(C_i)$, and the conditional probability of each attribute value given the class, $P(v_j|C_i)$. It estimates these quantities by counting in training data the frequency of occurrence of the classes and of the attribute values for each class. Then, assuming conditional independence of the attributes, it uses Bayes' rule to compute the posterior probability of each class given an unknown instance, returning as its prediction the class with the highest such value:

$$C = \operatorname*{argmax}_{C_i} P(C_i) \prod_j P(v_j|C_i) \ .$$

For ROC analysis, we used the posterior probability of the negative class as the case rating.

4.3 Support Vector Machines

Support vector machines, or SVMs (Boser et al., 1992), have performed well on traditional text classification tasks (Dumais et al., 1998; Joachims, 1998; Sahami et al., 1998), and performed well on ours. The method produces a linear classifier, so its concept description is a vector of weights, $\vec{w}$, and an intercept or a threshold, b. However, unlike other linear classifiers, such as Fisher's (1936), SVMs use a kernel function to map training data into a higher-dimensioned space so that the problem is linearly separable. It then uses quadratic programming to set $\vec{w}$ and b such that the hyperplane's

Trend Micro, Inc.
EX1008-7

margin is optimal, meaning that the distance is maximal from the hyperplane to the closest examples of the positive and negative classes. During performance, the method predicts the positive class if $\langle \vec{w} \cdot \vec{x} \rangle - b > 0$ and predicts the negative class otherwise. Quadratic programming can be expensive for large problems, but sequential minimal optimization (SMO) is a fast, efficient algorithm for training SVMs (Platt, 1998) and is the one implemented in WEKA (Witten and Frank, 2005). During performance, this implementation computes the probability of each class (Platt, 2000), and for ROC analysis, we used probability of the negative class as the rating.

4.4 Decision Trees

A decision tree is a rooted tree with internal nodes corresponding to attributes and leaf nodes corresponding to class labels. For symbolic attributes, branches leading to children correspond to the attribute's values. The performance element uses the attributes and their values of an instance to traverse the tree from the root to a leaf. It predicts the class label of the leaf node. The learning element builds such a tree by selecting the attribute that best splits the training examples into their proper classes. It creates a node, branches, and children for the attribute and its values, removes the attribute from further consideration, and distributes the examples to the appropriate child node. This process repeats recursively until a node contains examples of the same class, at which point, it stores the class label. Most implementations use the *gain ratio* for attribute selection (Quinlan, 1993), a measure based on the information gain. In an effort to reduce overtraining, most implementations also prune induced decision trees by removing subtrees that are likely to perform poorly on test data. WEKA's J48 (Witten and Frank, 2005) is an implementation of the ubiquitous C4.5 (Quinlan, 1993). During performance, J48 assigns weights to each class, and we used the weight of the negative class as the case rating.

4.5 Boosted Classifiers

Boosting (Freund and Schapire, 1996) is a method for combining multiple classifiers. Researchers have shown that *ensemble methods* often improve performance over single classifiers (Dietterich, 2000; Opitz and Maclin, 1999). Boosting produces a set of weighted models by iteratively learning a model from a weighted data set, evaluating it, and reweighting the data set based on the model's performance. During performance, the method uses the set of models and their weights to predict the class with the highest weight. We used the AdaBoost.M1 algorithm (Freund and Schapire, 1996) implemented in WEKA (Witten and Frank, 2005) to boost SVMs, J48, and naive Bayes. As the case rating, we used the weight of the negative class. Note that we did not apply AdaBoost.M1 to IBk because of the high computational expense.

5. Detecting Malicious Executables

With our methodology defined, our first task was to examine how well the learning methods detected malicious executables. We did so by conducting three experimental studies using a standard experimental design. The first was a pilot study to determine the size of words and n-grams, and the number of n-grams relevant for prediction. With those values determined, the second experiment consisted of applying all of the classification methods to a small collection of executables. The third then involved applying the methodology to a larger collection of executables, mainly to investigate how the approach scales.

Trend Micro, Inc.
EX1008-8

5.1 Experimental Design

To evaluate the approach and methods, we used stratified ten-fold cross-validation. That is, we randomly partitioned the executables into ten disjoint sets of equal size, selected one as a testing set, and combined the remaining nine to form a training set. We conducted ten such runs using each partition as the testing set.

For each run, we extracted *n*-grams from the executables in the training and testing sets. We selected the most relevant features from the training data, applied each classification method, and used the resulting classifier to rate the examples in the test set.

To conduct ROC analysis (Swets and Pickett, 1982), for each method, we pooled the ratings from the iterations of cross-validation, and used `labroc4` (Metz et al., 2003) to produce an empirical ROC curve and to compute its area and the standard error of the area. With the standard error, we computed 95% confidence intervals (Swets and Pickett, 1982).

5.2 Pilot Studies

We conducted pilot studies to determine three parameters: the size of *n*-grams, the size of words, and the number of selected features. Unfortunately, due to computational requirements, we were unable to evaluate exhaustively all methods for all settings of these parameters, so we assumed that the number of features would most affect performance, and began our investigation accordingly.

Using the experimental methodology described previously, we extracted bytes from 476 malicious executables and 561 benign executables and produced *n*-grams, for $n = 4$. (This smaller set of executables constituted our initial collection, which we later supplemented.) Using information gain, we then selected the best 10, 20, ..., 100, 200, ..., 1,000, 2,000, ..., 10,000 *n*-grams, and evaluated the performance of naive Bayes, SVMs, boosted SVMs, J48, and boosted J48. Selecting 500 *n*-grams produced the best results.

We fixed the number of *n*-grams at 500, and varied *n*, the size of the *n*-grams. We evaluated the same methods for $n = 1, 2, \ldots, 10$, and $n = 4$ produced the best results. We also varied the size of the words (one byte, two bytes, etc.), and results suggested that single bytes produced better results than did multiple bytes.

And so by selecting the top 500 *n*-grams of size four produced from single bytes, we evaluated all of the classification methods on this small collection of executables. We describe the results of this experiment in the next section.

5.3 Experiment with a Small Collection

Processing the small collection of executables produced 68,744,909 distinct *n*-grams. Following our experimental methodology, we used stratified ten-fold cross-validation, selected the 500 best *n*-grams, and applied all of the classification methods. The ROC curves for these methods are in Figure 1, while the areas under these curves (AUC) with 95% confidence intervals are in Table 2.

As one can see, the boosted methods performed well, as did the instance-based learner and the support vector machine. Naive Bayes did not perform as well, and we discuss this further in Section 8.

Trend Micro, Inc.
EX1008-9

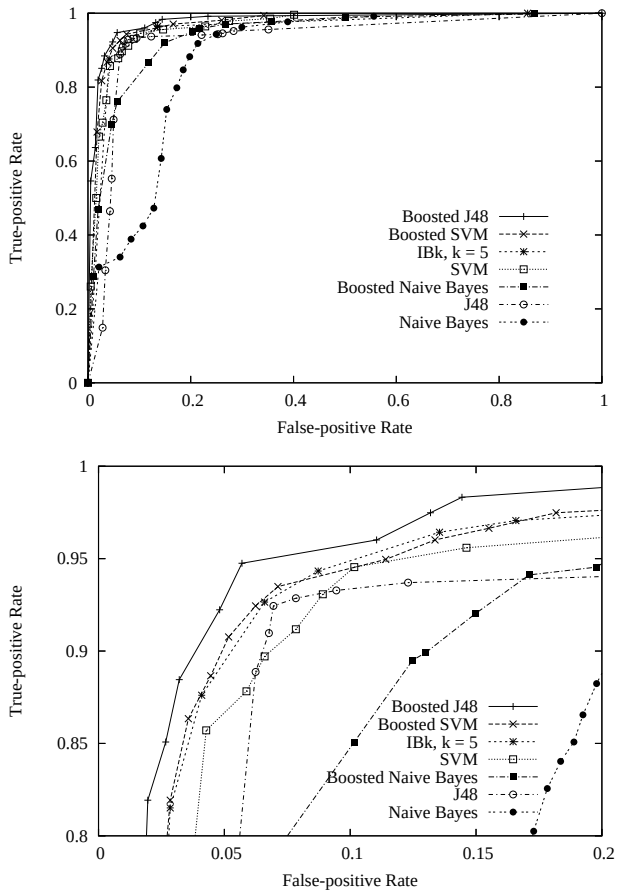

Figure 1: ROC curves for detecting malicious executables in the small collection. Top: The entire ROC graph. Bottom: A magnification.

Method	AUC
Boosted J48	0.9836±0.0095
Boosted SVM	0.9744±0.0118
IBk, $k = 5$	0.9695±0.0129
SVM	0.9671±0.0133
Boosted Naive Bayes	0.9461±0.0170
J48	0.9235±0.0204
Naive Bayes	0.8850±0.0247

Table 2: Results for detecting malicious executables in the small collection. Measures are area under the ROC curve (AUC) with a 95% confidence interval.

Trend Micro, Inc.
EX1008-10

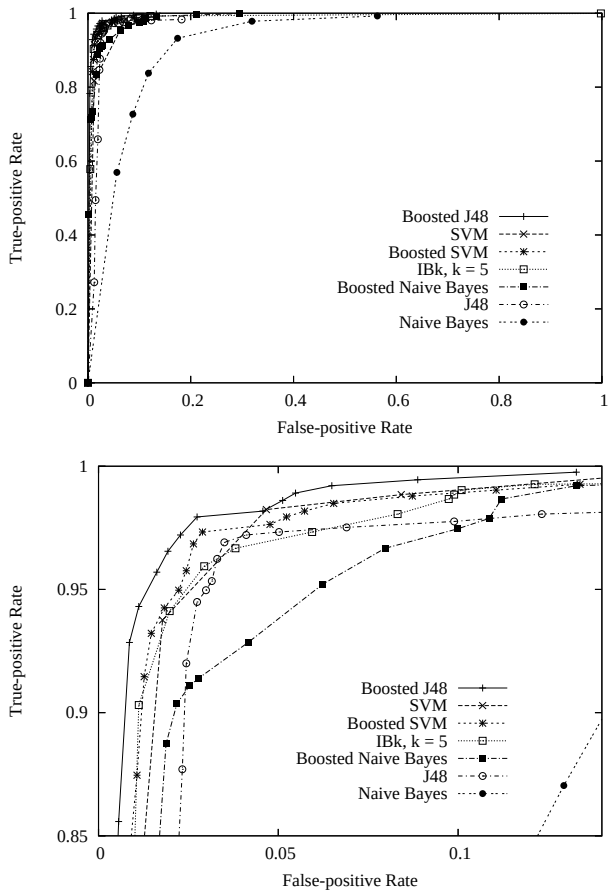

Figure 2: ROC curves for detecting malicious executables in the larger collection. Top: The entire ROC graph. Bottom: A magnification.

Method	AUC
Boosted J48	0.9958±0.0024
SVM	0.9925±0.0033
Boosted SVM	0.9903±0.0038
IBk, $k = 5$	0.9899±0.0038
Boosted Naive Bayes	0.9887±0.0042
J48	0.9712±0.0067
Naive Bayes	0.9366±0.0099

Table 3: Results for detecting malicious executables in the larger collection. Measures are area under the ROC curve (AUC) with a 95% confidence interval.

Trend Micro, Inc.
EX1008-11

5.4 Experiment with a Larger Collection

With success on a small collection, we turned our attention to evaluating the methodology on a larger collection of executables. As mentioned previously, this collection consisted of $1,971$ benign executables and $1,651$ malicious executables, while processing resulted in over 255 million distinct *n*-grams of size four. We followed the same experimental methodology—selecting the 500 top *n*-grams for each run of stratified ten-fold cross-validation, applying the classification methods, and plotting ROC curves.

Figure 2 shows the ROC curves for the various methods, while Table 3 presents the areas under these curves with 95% confidence intervals. As one can see, boosted J48 outperformed all other methods. Other methods, such as IB*k* and boosted SVMs, performed comparably, but the ROC curve for boosted J48 dominated all others.

6. Classifying Executables by Payload Function

We next attempted to classify malicious executables based on the function of their payload. That is, rather than detect malicious executables, we investigated the extent to which classification methods could determine whether a given malicious executable opened a backdoor, mass-mailed, or was an executable virus. We see this aspect of our work most useful for experts in computer forensics. A tool performing this task reliably could reduce the amount of effort to characterize previously undiscovered malicious executables.

Our first challenge was to identify and enumerate the functions of payloads of malicious executables. For this, we consulted VX Heavens and Symantec's Web site. Obviously, the information on these Web sites was not designed to support data-mining experiments, so we had to translate text descriptions into a more structured representation.

However, a greater problem was that we could not find information for all of the malicious executables in our collection. Indeed, we found information for only 525 of the $1,651$ malicious executables. As a result, for most categories, we had too few executables to build classifiers and conduct experiments.

A second challenge was that many executables fell into multiple categories. That is, many were so-called *multi-class examples*, a problem common in bioinformatics and document classification. For instance, a malicious executable might open a backdoor and log keystrokes, so it would be in both the backdoor and keylogger classes.

One approach is to create compound classes, such as backdoor+keylogger, in addition to the simple classes (e.g., backdoor and keylogger). One immediate problem was that we had too few examples to support this approach. We had a number of backdoors, a number of keyloggers, but had few executables that were both backdoors and keyloggers.

As a result, we chose to use *one-versus-all classification*. This involves grouping all of, say, the executables with backdoor capabilities into the backdoor class, regardless of their other capabilities (e.g., key logging), and placing all other executables into a non-backdoor class. One then builds a detector for the backdoor class, and does the same for all other classes.

To make a decision, one applies all of the detectors and reports the predictions of the individual classifiers as the overall prediction of the executable. For example, if the detectors for backdoor and for keylogger report hits, then the overall prediction for the executable is backdoor+keylogger.

Trend Micro, Inc.
EX1008-12

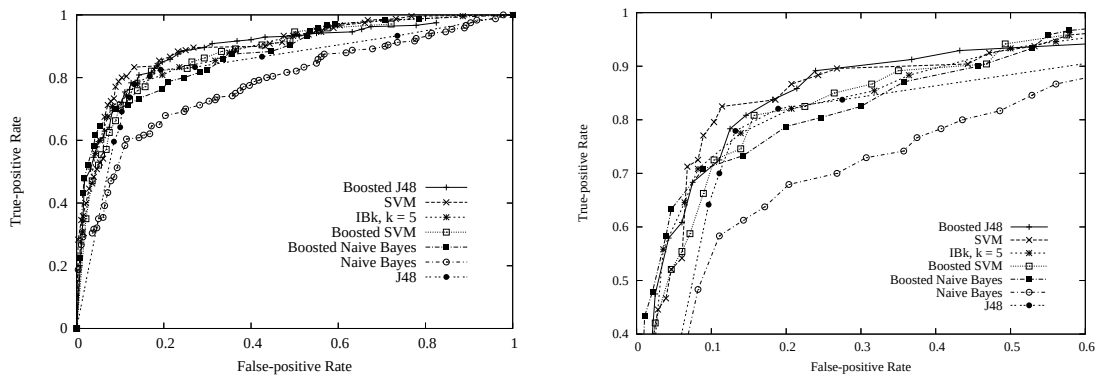

Figure 3: ROC curves for detecting malicious executables with mass-mailing capabilities. Left: The entire ROC graph. Right: A magnification.

Method	Payload		
	Mass Mailer	Backdoor	Virus
Boosted J48	0.8888±0.0152	0.8704±0.0161	0.9114±0.0166
SVM	0.8986±0.0145	0.8508±0.0171	0.8999±0.0175
IBk, $k = 5$	0.8829±0.0155	0.8434±0.0174	0.8975±0.0177
Boosted SVM	0.8758±0.0160	0.8625±0.0165	0.8775±0.0192
Boosted Naive Bayes	0.8773±0.0159	0.8313±0.0180	0.8370±0.0216
J48	0.8315±0.0184	0.7612±0.0205	0.8295±0.0220
Naive Bayes	0.7820±0.0205	0.8190±0.0185	0.7574±0.0250

Table 4: Results for detecting payload function. Measures are area under the ROC curve (AUC) with a 95% confidence interval.

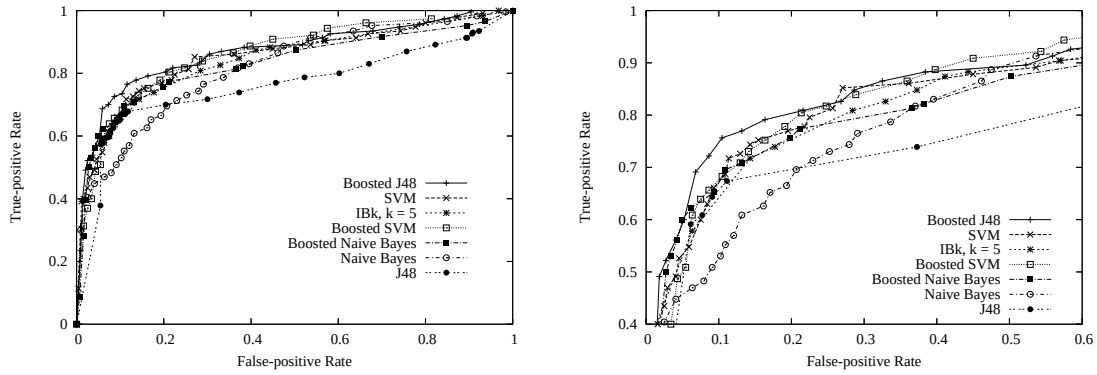

Figure 4: ROC curves for detecting malicious executables with backdoor capabilities. Left: The entire ROC graph. Right: A magnification.

Trend Micro, Inc.
EX1008-13

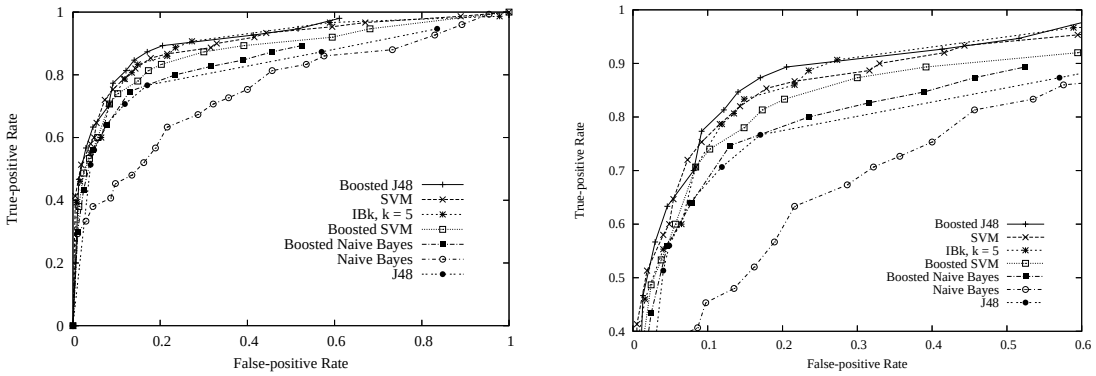

Figure 5: ROC curves for detecting executable viruses. Left: The entire ROC graph. Right: A magnification.

6.1 Experimental Design

We followed an experimental design similar to that described previously, but for each of the functional categories, we created a data set using only malicious executables. We divided it into two subsets, one containing executables that performed the function and one containing those that did not. We then proceeded as before, using stratified ten-fold cross-validation, applying and evaluating the methods, and constructing ROC curves.

6.2 Experimental Results

We present results for three functional categories: mass-mailer, backdoor, and executable virus. Figures 3–5 present the ROC curves for seven methods on the task of detecting executables that mass-mail, open a backdoor, or contain an executable virus. The areas under these ROC curves appear in Table 4. Overall, the results are not as good as those in the experiment that involved detecting malicious executables in our full collection of malicious executables, and we discuss possible causes in Section 8.

The relative performance of the methods on this task was roughly the same as in previous experiments. Naive Bayes generally did not perform as well as the other methods, and we discuss the reasons for this in Section 8. Boosted J48 and the SVM were again the best performing methods, although on this task, the SVM performed slightly better than on previous tasks.

7. Evaluating Real-world, Online Performance

Finally, to estimate what performance might be in an operational environment, we applied the methods to 291 malicious executables discovered after we gathered our original collection. In the previous sections, we generally followed a common experimental design in machine learning and data mining: We randomly partitioned our data into training and testing sets, applied algorithms to the training set, and evaluated the resulting detectors on the testing set. However, one problem with this design for this application is that learning methods were trained on recent malicious executables and tested on older ones. Crucially, this design does not reflect the manner in which a system based

Trend Micro, Inc.
EX1008-14

Method	Desired False-positive Rate					
	0.01		0.05		0.1	
	P	A	P	A	P	A
Boosted J48	0.94	0.86	0.99	0.98	1.00	1.00
SVM	0.82	0.41	0.98	0.90	0.99	0.93
Boosted SVM	0.86	0.56	0.98	0.89	0.99	0.92
IBk, $k = 5$	0.90	0.67	0.99	0.81	1.00	0.99
Boosted Naive Bayes	0.79	0.55	0.94	0.93	0.98	0.98
J48	0.20	0.34	0.97	0.94	0.98	0.95
Naive Bayes	0.48	0.28	0.57	0.72	0.81	0.83

Table 5: Results of a real-world, online evaluation. Predicted (P) versus actual (A) detect rates for three desired false-positive rates on 291 new, previously unseen malicious executables. Predicted detect rates are from Figure 2 and the experiment described in Section 5.4.

on our methodology would be used. In this section, we rectify this and describe an experiment designed to better evaluate the real-world, online performance of the detectors.

As mentioned previously, we gathered our collection of executables in the summer of 2003. In August of 2004, we retrieved from VX Heavens all of the new malicious executables and selected those that were discovered after we gathered our original collection. This required retrieving the 3,082 new executables that were in the PE format and using commercial software to verify independently that each executable was indeed malicious. We then cross-referenced the names of the verified malicious executables with information on various Web sites to produce the subset of malicious executables discovered between July of 2003 and August of 2004. There were 291 such executables.

7.1 Experimental Design

To conduct this experiment, we built classifiers from all of the executables in our original collection, both malicious and benign. We then selected three desired false-positive rates, 0.01, 0.05, and 0.1. This, in turn, let us select three decision thresholds from each ROC curve for each method. Using these thresholds to parameterize specific classifiers, we applied them to each of the 291 new malicious executables in the order of their date of discovery.

7.2 Experimental Results

Rather than analyze all of these results, we will discuss the actual (A) detection rates for the desired false-positive rate of 0.05.[1] As one can see, boosted decision trees detected about 98% of the new malicious executables, missing 6 of 291 malicious executables. For some applications, six may be too many, but if one is willing to tolerate a false-positive rate of 0.1, then one can achieve a perfect detect rate, at least on these 291 malicious executables.

1. Our reasoning was that, for most operational scenarios, a desired false-positive rate of 0.1 would be too high, and the detect rates achieved for a desired false-positive rate of 0.01 were too low. Knowledge of a given operational environment would presumably help us choose a more appropriate decision threshold.

Trend Micro, Inc.
EX1008-15

However, it is also important to compare the actual detection rates to the predicted rates from the experiment using our larger collection of executables, discussed in Section 5.4. As one can see in Table 5 by comparing the predicted (P) and actual (A) detection rates for a desired false-positive rate of 0.05, four methods (SVM, boosted SVM, IB*k*, and J48) performed worse on the new malicious executables, two methods (boosted J48 and boosted naive Bayes) performed about as well under both conditions, and one method (naive Bayes) performed much better. Nonetheless, we determined that boosted decision trees achieved the best performance overall, not only in terms of the best actual performance on the new malicious executables, but also in terms of matching the predicted performance from the experiment involving the larger collection of executables.

8. Discussion

To date, our results suggest that methods of machine learning, data mining, and text classification are appropriate and useful for detecting malicious executables in the wild. Boosted classifiers, IB*k*, and a support vector machine performed exceptionally well given our current data collection. That the boosted classifiers generally outperformed single classifiers echos the conclusion of several empirical studies of boosting (Bauer and Kohavi, 1999; Breiman, 1998; Dietterich, 2000; Freund and Schapire, 1996), which suggest that boosting improves the performance of unstable classifiers, such as J48, by reducing their bias and variance (Bauer and Kohavi, 1999; Breiman, 1998). Boosting can adversely affect stable classifiers (Bauer and Kohavi, 1999), such as naive Bayes, although in our study, boosting naive Bayes improved performance. Stability may also explain why the benefit of boosting SVMs was inconclusive in our study (Breiman, 1998).

Our experimental results suggest that the methodology will scale to larger collections of executables. The larger collection in our study contained more than three times the number of executables in the smaller collection. Yet, as one can see in Tables 2 and 3, the absolute performance of all of the methods was better for the larger collection than for the smaller one. The relative performance of the methods changed somewhat. For example, the SVM moved from fourth to second, displacing the boosted SVMs and IB*k*.

Regarding our results for classifying executables by function, we suspect the methods did not perform as well as they did on the detection task for two reasons. First, with the classification task, the algorithms must make finer distinctions between malicious and benign executables. For example, a malicious executable that mass-mails will be similar in some respects to a legitimate e-mail client. Such similarity could account for the lower performance.

Indeed, in pilot studies, we attempted to use the methods to distinguish between benign and malicious executables that edited the registry. Performance on this task was lower than on the others, and we suspect this is because editing the registry is a function common to many executables, malicious and benign. Such similarity could have accounted for the lower performance.

Second, we suspect that, on the classification task, performance suffered because the algorithms built classifiers from fewer examples. Performance on the detection task improved when we added additional examples, and we suspect that, likewise, with additional examples, we will obtain similar improvements in accuracy on the classification task.

Regarding our online evaluation of the methods, we believe the experimental design represents how such methods would be used in a commercial or operational system. We did not conduct this experiment from the outset (Kolter and Maloof, 2004) because it was impossible to determine the date of discovery of all of the malicious executables in our collection. Moreover, to conduct the ideal

Trend Micro, Inc.
EX1008-16

```
030b0105 = T
|   0b010219 = T: malicious (2.0)
|   0b010219 = F
|   |   0000000a = T
|   |   |   0001ff25 = T
|   |   |   |   0c100001 = T
|   |   |   |   |   0000c700 = T: benign (6.0)
|   |   |   |   |   0000c700 = F: malicious (2.0)
|   |   |   |   0c100001 = F: malicious (2.0)
|   |   |   0001ff25 = F: benign (10.0)
|   |   0000000a = F: benign (253.0)
030b0105 = F
...
```

Figure 6: Portion of a decision tree built from benign and malicious executables.

experiment, we would also need to collect different versions of benign executables and when they were released. It was easier to take one "snapshot" of existing malicious and benign executables, conduct traditional experiments, and then, at a later date, retrieve any new malicious executables for the online experiment.

During the processing of the 291 new malicious executables, we did not update the classifiers when there was a mistake or a so-called "near miss". Clearly, in an operational setting, if the system were to make a mistake or to detect a malicious executable with low certainty, then, ideally, one would add it to the collection and reprocess everything. (One would also have to do the same for benign executables.) However, because of the computational overhead, we did not do this, and as a consequence, our results are pessimistic. Presumably, all of the methods would perform better with the benefit of additional training data. Nonetheless, for some methods, the results are quite promising, as shown in Table 5.

Visual inspection of the concept descriptions yielded interesting insights, but further work is required before these descriptions will be directly useful for computer-forensic experts. Figure 6 shows a portion of one decision tree built from benign and malicious executables.

As an example, the first branch of the tree indicates that if an executable contains the *n*-grams `030b0105` and `0b010219`, then it is malicious. After an analysis of our collection of malicious executables, we discovered that both *n*-grams were from the PE header, implying that a single file contained two such headers. More investigation revealed that two executables in our collection contained another executable, which explains the presence of two PE headers in a single file. This was an interesting find, but it represented an insignificantly small portion of the malicious programs.

Leaf nodes covering many executables were often at the end of long branches where one set of *n*-grams (i.e., byte codes) had to be present and another set had to be absent. Understanding why the absence of byte codes was important for an executable being malicious proved to be a difficult and often impossible task.

It was fairly easy to establish that some *n*-grams in the decision tree were from string sequences and that some were from code sequences, but some were incomprehensible. For example, the *n*-gram `0000000a` appeared in 75% of the malicious executables, but it was not part of the executable format, it was not a string sequence, and it was not a code sequence. We have yet to determine its purpose.

Trend Micro, Inc.
EX1008-17

Nonetheless, for the large collection of executables, the size of the decision trees averaged over 10 runs was about 90 nodes. No tree exceeded 103 nodes. The heights of the trees never exceeded 13 nodes, and subtrees of heights of 9 or less covered roughly 99.3% of the training examples. While these trees did not support a thorough forensic analysis, they did compactly encode a large number of benign and malicious executables.

Unfortunately, the best performing method did not always produce the most readable concept descriptions. Of the methods we considered, J48 is mostly likely to produce descriptions useful for computer-forensic experts. However, J48 was not the best performing method in any of our experiments. The best performing method was boosted J48. While it is true that this method also produces decision trees, it actually produces a set of weighted trees. We discussed the difficulties of analyzing a single tree, so it is unclear if analyzing an ensemble of weighted trees will be helpful for experts. And since J48 was not the best performing method, we may also have to question the utility of analyzing a single decision tree when its performance is subpar.

We estimated that about 20–25% of the malicious executables in our collection were obfuscated with either compression or encryption. To the best of our knowledge, none of the benign executables were obfuscated. Early in our investigation, we conjectured that obfuscation would likely interfere with classifying payload function, but that it would not do so with detecting whether the executable is malicious. Our results on these tasks seem to support our conjecture: We were able to detect malicious executables with high accuracy, so it is unlikely that obfuscation affected performance. On the other hand, we were not able to achieve the same high accuracy when classifying payload function, and the presence of obfuscation may have contributed to this result.

With the detection task, it is possible that the methods simply learned to detect certain forms of obfuscation, such as run-time compression, but this does not seem problematic as long as those forms are correlated with malicious executables. Based on our collection and our own investigation, this is presently the case.

To place our results in context with the study of Schultz et al. (2001), they reported that the best performing approaches were naive Bayes trained on the printable strings from the program and an ensemble of naive-Bayesian classifiers trained on byte sequences. They did not report areas under their ROC curves, but visual inspection of these curves suggests that with the exception of naive Bayes, all of our methods outperformed their ensemble of naive-Bayesian classifiers. It also appears that our best performing methods, such as boosted J48, outperformed their naive Bayesian classifier trained with strings.

These differences in performance could be due to several factors. We analyzed different types of executables: Their collection consisted mostly of viruses, whereas ours contained viruses loaders, worms, and Trojan horses. Ours consisted of executables in the Windows PE format; about 5.6% of theirs was in this format.

Our better results could be due to how we processed byte sequences. Schultz et al. (2001) used non-overlapping two-byte sequences, whereas we used overlapping sequences of four bytes. With their approach it is possible that a useful feature (i.e., a predictive sequence of bytes) would be split across a boundary. This could explain why in their study string features appeared to be better than byte sequences, since extracted strings would not be broken apart. Their approach produced much less training data than did ours, but our application of feature selection reduced the original set of more than 255 million *n*-grams to a manageable 500.

Our results for naive Bayes were poor in comparison to theirs. We again attribute this to the differences in data extraction methods. Naive Bayes is well known to be sensitive to conditionally

Trend Micro, Inc.
EX1008-18

dependent attributes (Domingos and Pazzani, 1997). We used overlapping byte sequences as attributes, so there were many that were conditionally dependent. Indeed, after analyzing decision trees produced by J48, we found evidence that overlapping sequences were important for detection. Specifically, some subpaths of these decision trees consisted of sequentially overlapping terms that together formed byte sequences relevant for prediction. Schultz et al.'s (2001) extraction methods would not have produced conditionally dependent attributes to the same degree, if at all, since they used strings and non-overlapping byte sequences.

Regarding our experimental design, we decided to pool a method's ratings and produce a single ROC curve (see Section 5.1) because `labroc4` (Metz et al., 2003) occasionally could not fit an ROC curve to a method's ratings from a single fold of cross-validation (i.e., the ratings were degenerate). We also considered producing ROC convex hulls (Provost and Fawcett, 2001) and cost curves (Drummond and Holte, 2000), but determined that traditional ROC analysis was appropriate for our results (e.g., the curve for boosted J48 dominated all other curves).

In our study, there was an issue of high computational overhead. Selecting features was expensive, and we had to resort to a disk-based implementation for computing information gain, which required a great deal of time and space to execute. However, once selected, WEKA's (Witten and Frank, 2005) Java implementations executed quickly on the training examples with their 500 Boolean attributes.

The greatest impediment to our investigation was the absence of detailed, structured information about the malicious executables in our collection. As mentioned previously, we had $1,651$ malicious executables, but found information for only 525 that was sufficient to support our experiment on function classification, described in Section 6.

We arduously gathered this information by reading it from Web pages. We contemplated implementing software to extract this information, but abandoned this idea because of the difficulty of processing such semi-structured information and because of that information's ad hoc nature. As an example, for one executable, the description that it opened a backdoor appeared in a section describing the executable's payload, whereas the same information for another executable appeared in a section describing how the malicious executable degrades performance. This suggests the need for a well-engineered database for storing information about malicious software.

As another example, we found incomplete information about the dates of discovery of many of the malicious executables. With this information, we could have evaluated our methods on the malicious executables in the order they were discovered. This would have been similar to the evaluation we conducted using the 291 previously unseen malicious executables, as described in Section 7, but having complete information for all of the executables would have resulted in a much stronger evaluation.

However, to conduct a proper evaluation, we would also needed a comparable collection of benign executables. It seems unlikely that we would be able to reconstruct realistic snapshots of complete Windows systems over a sufficient period of time. Snapshots of the system software might be possible, but creating a historical archive of application software and their different versions seems all but impossible. Such snapshots would be required for a commercial system, and creating such snapshots would be easier going forward.

In terms of our approach, it is important to note that we have investigated other methods of data extraction. For instance, we examined whether printable strings from the executable might be useful, but reasoned that subsets of *n*-grams would capture the same information. Indeed, after inspecting some of the decision trees that J48 produced, we found evidence suggesting that *n*-grams

Trend Micro, Inc.
EX1008-19

formed from strings were being used for detection. Nonetheless, if we later determine that explicitly representing printable strings is important, we can easily extend our representation to encode their presence or absence. On the other hand, as we stated previously, one must question the use of printable strings or DLL information since compression and other forms of obfuscation can mask this information.

We also considered using disassembled code as training data. For malicious executables using compression, being able to obtain a disassembly of critical sections of code may be a questionable assumption. Moreover, in pilot studies, a commercial product failed to disassemble some of our malicious executables.

We considered an approach that runs malicious executables in a "sandbox" and produces an audit of the machine instructions. Naturally, we would not be able to execute completely the program, but $10,000$ instructions may be sufficient to differentiate benign and malicious behavior. We have not pursued this idea because of a lack of auditing tools, the difficulty of handling large numbers of interactive programs, and the inability of detecting malicious behavior occurring near the end of sufficiently long programs. Moreover, some malicious programs can detect when they are being executed by a virtual machine and either terminate execution or avoid executing malicious sections of code.

There are at least two immediate commercial applications of our work. The first is a system, similar to MECS, for detecting malicious executables. Server software would need to store all known malicious executables and a comparably large set of benign executables. Due to the computational overhead of producing classifiers from such data, algorithms for computing information gain and for evaluating classification methods would have to be executed incrementally, in parallel, or both.

Client software would need to extract only the top n-grams from a given executable, apply a classifier, and predict. Updates to the classifier could be made remotely over the Internet. Since the best performing method may change with new training data, it will be critical for the server to evaluate a variety of methods and for the client to accommodate any of the potential classifiers. Used in conjunction with standard signature methods, these methods could provide better detection of malicious executables than is currently possible.

The second is a system oriented more toward computer-forensic experts. Even though work remains before decision trees could be used to analyze malicious executables, one could use IBk to retrieve known malicious executables similar to a newly discovered malicious executable. Based on the properties of the retrieved executables, such a system could give investigators insights into the new executable's function. While it remains an open issue whether an executable's statistical properties are predictive of its function, we have presented evidence suggesting it may be possible to achieve useful detection rates when predicting function.

9. Concluding Remarks

We considered the application of techniques from machine learning, data mining, and text classification to the problem of detecting and classifying unknown malicious executables in the wild. After evaluating a variety of inductive methods, results suggest that, for the task of detecting malicious executables, boosted J48 produced the best detector with an area under the ROC curve of 0.996.

We also investigated the ability of these methods to classify malicious executables based on their payload's function. For payloads that mass-mail, open a backdoor, and inject viral code, boosted J48 again produced the best detectors with areas under the ROC curve around 0.9. While overall

Trend Micro, Inc.
EX1008-20

the performance on this task was not as impressive as that on the detection task, we contend that performance will improve with the removal of obfuscation and with additional training examples.

Finally, boosted J48 also performed well on the task of detecting 291 malicious executables discovered after we gathered our original collection, an evaluation that best reflects how one might use the methodology in an operational environment. Indeed, our methodology resulted in a fielded prototype called MECS, the Malicious Executable Classification System, which we delivered to the MITRE Corporation.

In future work, we hope to remove obfuscation from our malicious executables and rerun the experiment on classifying payload function. Removing obfuscation and producing "clean" executables may prove challenging, but doing so would provide the best opportunity to evaluate whether obfuscation affected the performance of the classifiers.

We also plan to investigate the similarity of malicious executables and how such executables change over time. In this regard, we have not yet attempted to cluster our collection of executables, but doing so may yield two insights. First, if a new, unanalyzed malicious executable is similar to others that have been analyzed, it may help computer-forensic experts conduct a faster analysis of the new threat.

Second, if we add information about when the executables were discovered, we may be able to determine how malicious executables were derived from others. Although there is a weak analog between DNA sequences and byte codes from executables, we may be able to use a collection of malicious executables to build phylogenetic trees that may elucidate "evolutionary relationships" existing among them.

We anticipate that inductive approaches, such as ours, is but one process in an overall strategy for detecting and classifying "malware." When combined with approaches that use cryptographic hashes, search for known signatures, execute and analyze code in a virtual machine, we hope that such a strategy for detecting and classifying malicious executables will improve the security of computers. Indeed, the delivery of MECS to MITRE has provided computer-forensic experts with a valuable tool. We anticipate that continued investigation of inductive methods for detecting and classifying malicious executables will yield additional tools and more secure systems.

Acknowledgments

The authors first and foremost thank William Asmond and Thomas Ervin of the MITRE Corporation for providing their expertise, advice, and collection of malicious executables. We thank Nancy Houdek of Georgetown for help gathering information about the functional characteristics of the malicious executables in our collection. The authors thank the anonymous reviewers for their time and helpful comments, Ophir Frieder of IIT for help with the vector space model, Abdur Chowdhury of IIT for advice on the scalability of the vector space model, Bob Wagner of the FDA for assistance with ROC analysis, Eric Bloedorn of MITRE for general guidance on our approach, and Matthew Krause of Yale University for reviewing an earlier draft of the paper. Finally, we thank Richard Squier of Georgetown for supplying much of the additional computational resources needed for this study through Grant No. DAAD19-00-1-0165 from the U.S. Army Research Office. This research was conducted in the Department of Computer Science at Georgetown University. Our work was supported by the MITRE Corporation under contract 53271. The authors are listed in alphabetical order.

Trend Micro, Inc.
EX1008-21

References

D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine Learning*, 6: 37–66, 1991.

A. Aiken. MOSS: A system for detecting software plagiarism. Software, Department of Computer Science, University of California, Berkeley, http://www.cs.berkeley.edu/ aiken/moss.html, 1994.

Anonymous. *Maximum Security*. Sams Publishing, Indianapolis, IN, 4th edition, 2003.

B. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1–2):105–139, 1999.

B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fourth Workshop on Computational Learning Theory*, pages 144–152, New York, NY, 1992. ACM Press.

L. Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the Twelfth USENIX Security Symposium*, Berkeley, CA, 2003. Advanced Computing Systems Association.

W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, San Francisco, CA, 1995. Morgan Kaufmann.

T. G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–158, 2000.

P. Domingos and M. J. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

C. Drummond and R. C. Holte. Explicitly representing expected cost: An alternative to ROC representation. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 198–207, New York, NY, 2000. ACM Press.

S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 148–155, New York, NY, 1998. ACM Press.

E. Durning-Lawrence. *Bacon is Shake-speare*. The John McBride Company, New York, NY, 1910.

R. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156, San Francisco, CA, 1996. Morgan Kaufmann.

Trend Micro, Inc.
EX1008-22

A. R. Gray, P. J. Sallis, and S. G. MacDonell. Software forensics: Extending authorship analysis techniques to computer programs. In *Proceedings of the Third Biannual Conference of the International Association of Forensic Linguists*, pages 1–8, Birmingham, UK, 1997. International Association of Forensic Linguists. URL `citeseer.nj.nec.com/gray97software.html`.

D. Hand, H. Mannila, and P. Smyth. *Principles of data mining*. MIT Press, Cambridge, MA, 2001.

A. K. Jain, R. P. W. Duin, and J. Mao. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, 2000.

H. T. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31(1):1–8, 1988.

T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 487–494, Berlin, 1998. Springer.

J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White. Biologically inspired defenses against computer viruses. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 985–996, San Francisco, CA, 1995. Morgan Kaufmann.

B. Kjell, W. A. Woods, and O. Frieder. Discrimination of authorship using visualization. *Information Processing and Management*, 30(1):141–150, 1994.

J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 470–478, New York, NY, 2004. ACM Press.

I. Krsul. Authorship analysis: Identifying the author of a program. Master's thesis, Purdue University, West Lafayette, IN, 1994.

I. Krsul and E. Spafford. Authorship analysis: Identifying the authors of a program. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 514–524, Gaithersburg, MD, 1995. National Institute of Standards and Technology.

R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: A malicious code filter. *Computers & Security*, 14 (6):541–566, 1995. http://www.cs.columbia.edu/ids/mef/llo95.ps.

M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the Association of Computing Machinery*, 7(3):216–244, 1960.

G. McGraw and G. Morisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, pages 33–41, September/October 2000. http://www.cigital.com/ gem/malcode.pdf.

C. E. Metz, Y. Jiang, H. MacMahon, R. M. Nishikawa, and X. Pan. ROC software. Web page, Kurt Rossmann Laboratories for Radiologic Image Research, University of Chicago, Chicago, IL, 2003. URL `http://www-radiology.uchicago.edu/krl/roc\_soft.htm`.

Trend Micro, Inc.
EX1008-23

P. Miller. hexdump 1.4. Software, http://gd.tuwien.ac.at/softeng/Aegis/hexdump.html, 1999.

T. M. Mitchell. *Machine learning*. McGraw-Hill, New York, NY, 1997.

D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999. URL `http://www.jair.org`.

J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. J. C. Burges, and S. Mika, editors, *Advances in Kernel Methods—Support Vector Learning*. MIT Press, Cambridge, MA, 1998.

J. Platt. Probabilities for SV machines. In P. J. Bartlett, B. Schölkopf, D Schuurmans, and A. J. Smola, editors, *Advances in Large-Margin Classifiers*, pages 61–74. MIT Press, Cambridge, MA, 2000.

F. Provost and T. Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42: 203–231, 2001.

J. R. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Francisco, CA, 1993.

M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 AAAI Workshop*, Menlo Park, CA, 1998. AAAI Press. Technical Report WS-98-05.

M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–49, Los Alamitos, CA, 2001. IEEE Press. URL `http://www.cs.columbia.edu/~ezk/research`.

S. Soman, C. Krintz, and G. Vigna. Detecting malicious Java code using virtual machine auditing. In *Proceedings of the Twelfth USENIX Security Symposium*, Berkeley, CA, 2003. Advanced Computing Systems Association.

E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12:585–595, 1993.

J. A. Swets and R. M. Pickett. *Evaluation of diagnostic systems: Methods from signal detection theory*. Academic Press, New York, NY, 1982.

G. Tesauro, J. O. Kephart, and G. B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, August 1996.

I. H. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 2005. http://www.cs.waikato.ac.nz/ml/weka/index.html.

Y. Yang and J. O. Pederson. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 412–420, San Francisco, CA, 1997. Morgan Kaufmann.

Trend Micro, Inc.
EX1008-24