

Learning to Detect Malicious Executables in the Wild

Jeremy Z. Kolter
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA
jzk@cs.georgetown.edu

Marcus A. Maloof
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA
maloo@cs.georgetown.edu

ABSTRACT

In this paper, we describe the development of a fielded application for detecting malicious executables in the wild. We gathered 1971 benign and 1651 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 also suggest that our methodology will scale to larger collections of executables. To the best of our knowledge, ours is the only fielded application for this task developed using techniques from machine learning and data mining.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Data Mining*; I.2.6 [Artificial Intelligence]: Learning—*Concept Learning*; K.6.5 [Management of Computing and Information Systems]: Security and Protection—*Invasive Software*

General Terms: Algorithms, Experimentation, Security

Keywords: Data Mining, Concept Learning, Security, Malicious 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” [27, 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 the system’s users. As 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* [3, p. 339]. Indeed, researchers have recently discovered that simple obfuscation techniques foil commercial programs for virus detection [7]. These challenges have prompted some researchers to investigate learning methods for detecting new or unknown viruses, and more generally, malicious code.

Our efforts to address this problem have resulted in a fielded application, built using techniques from machine learning [30] and data mining [17]. The Malicious Executable Classification System (MECS) currently detects unknown malicious executables “in the wild”, that is, without removing any obfuscation. To date, we have gathered 1971 system and non-system executables, which we will refer to as “benign” executables, and 1651 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*, TFIDF, 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 [40], 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. 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.

With this paper, we make three main contributions. We show how established methods for text classification apply to executables. We present empirical results from an extensive study of inductive methods for detecting malicious executables in the wild. We report on a fielded application developed using machine learning and data mining.

In the three sections that follow, we describe related work, our data collection, and the methods we applied. Then, in Section 6, we present empirical results, and in Section 7, we discuss these results and other approaches.

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. These have concentrated mostly on PC viruses, 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.

In an early attempt, Lo et al. [25] conducted an analysis of several programs—evidently by hand—and identified *tell-tale signs*, which they subsequently used to filter new programs. While we appreciate their attempt 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 [21] and have incorporated a similar approach for detecting boot-sector viruses into IBM’s Anti-Virus software [41].

More recently, instead of focusing on boot-sector viruses, Schultz et al. [37] 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) functions calls from the DLLs, and (3) the number of different system calls from within 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 [29], 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 [8] 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 Table 1. The authors also presented ROC curves [40], 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

Table 1: Results from the study conducted by Schultz et al. [37]

Method	TP Rate	FP Rate	Accuracy (%)
Signature + <code>hexdump</code>	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 + <code>strings</code>	0.97	0.04	97.11
Voting Naive Bayes + <code>hexdump</code>	0.98	0.06	96.88

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* [3, p. 370], which involves comparing current files and directories to past copies; one can also compare cryptographic hashes. One can also audit running programs [38] and statically analyze executables using pre-defined malicious patterns [7]. 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 [13], or who wrote the twelve disputed Federalist Papers, Hamilton or Madison [22]. Recently, similar techniques have been used in the relatively new field of *software forensics* to determine program authorship [39]. Gray et al. [15] wrote a position paper on the subject of authorship, whereas Krsul [23] 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 [2, 19], which we will not survey here.

Krsul [23] 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” [24, §5.1].

While interesting, it is unclear how much confidence we should have in these results. Krsul [23] 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 [18]. 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 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 1971 benign executables and 1651 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>).

We obtained viruses, 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 collection, a commercial product for detecting viruses failed to identify 18 of the 114 malicious executables. Note that for viruses, we examined only the loader programs; we did not include infected executables in our study.

We used the `hexdump` utility [29] 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 6.1. Using the n -grams from all of the executables, we applied techniques from information retrieval and text classification, which we discuss further in the next section.

4. CLASSIFICATION METHODOLOGY

Our overall approach drew techniques from information retrieval (e.g., [16]) and from text classification (e.g., [12, 36]). We used the n -grams extracted from the executables to form training examples by viewing each n -gram as a binary attribute that is either present in (i.e., 1) or absent from (i.e., 0) the executable. We selected the most relevant attributes (i.e., n -grams) by computing the *information gain* (IG) for each:

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

where C is the class, v_j is the value of the j th attribute, $P(v_j, C)$ is the proportion that the j th attribute has the value v_j in the class C_i , $P(v_j)$ is the proportion that the j th n -gram takes the value v_j in the training data, $P(C)$ is the proportion of the training data belonging to the class C . This measure is also called *average mutual information* [43].

We then selected the top 500 n -grams, a quantity we determined through pilot studies (see Section 6.1), and applied several learning methods, most of which are implemented in WEKA [42]: *IBk*, TFIDF, 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.

4.1 Instance-based Learner

One of the simplest learning methods is the instance-based (IB) learner [1]. 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 binary 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 *IBk*, 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*.

4.2 The TFIDF Classifier

For the TFIDF classifier, we followed a classical approach from information retrieval [16]. We used the *vector space model*, which entails assigning to each executable (i.e., document) a vector of size equal to the total number of distinct n -grams (i.e., terms) in the collection. The components of each vector were weights of the top n -grams present in the executable. For the j th n -gram of the i th executable, the method computes the weight w_{ij} , defined as

$$w_{ij} = tf_{ij} \times idf_j,$$

where tf_{ij} (i.e., term frequency) is the number of times the i th n -gram appears in the j th executable and $idf_j = \log \frac{d}{df_j}$ (i.e., the inverse document frequency), where d is the total number of executables and df_j is the number of executables that contain the j th n -gram. It is important to note that this classifier was the only one that used continuous attribute values; all others used binary attribute values.

To classify an unknown instance, the method uses the top n -grams from the executable, as described previously, to form a vector, \vec{u} , the components of which are each n -gram’s inverse document frequency (i.e., $u_j = idf_j$).

Once formed, the classifier computes a similarity coefficient (SC) between the vector for the unknown executable and each vector for the executables in the collection using the *cosine similarity measure*:

$$SC(\vec{u}, \vec{w}_i) = \frac{\sum_{j=1}^k u_j w_{ij}}{\sqrt{\sum_{j=1}^k u_j^2 \cdot \sum_{j=1}^k w_{ij}^2}},$$

where \vec{u} is the vector for the unknown executable, \vec{w}_i is the vector for the i th executable, and k is the number of distinct n -grams in the collection.

After selecting the top five closest matches to the unknown, the method takes a weighted majority vote of the executable labels, and returns the class with the least weight as the prediction. It uses the cosine measure as the weight. Since we evaluated the methods using ROC analysis [40], which requires *case ratings*, we summed the cosine measures of the negative executables in the top five, subtracted the sum of the cosine measures of the positive executables, and used the resulting value as the rating. In the following discussion, we will refer to this method as the TFIDF classifier.

4.3 Naive Bayes

Naive Bayes is a probabilistic method that has a long history in information retrieval and text classification [26]. 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.

4.4 Support Vector Machines

Support vector machines (SVMs) [5] have performed well on traditional text classification tasks [12, 20, 36], 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, SVMs use a kernel function to map training data into a higher dimensional space so that the problem is linearly separable. It then uses quadratic programming to set \vec{w} and b such that the hyperplane’s 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 [32] and is the one implemented in WEKA [42]. During performance, this implementation computes the probability of each class [33], and for ROC analysis, we used probability of the negative class as the rating.

4.5 Decision Trees

A decision tree is a 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 [35], 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 [42] is an implementation of the ubiquitous C4.5 [35]. During performance, J48 assigns weights to each class, and we used the weight of the negative class as the case rating.

4.6 Boosted Classifiers

Boosting [14] is a method for combining multiple classifiers. Researchers have shown that *ensemble methods* often improve performance over single classifiers [9, 31]. 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 [14] implemented in WEKA [42] 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. EXPERIMENTAL DESIGN

To evaluate the approaches 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 [40], for each method, we pooled the ratings from the iterations of cross-validation, and used `labroc4` [28] 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 [40]. We present and discuss these results in the next section.

6. EXPERIMENTAL RESULTS

We conducted three experimental studies using our data collection and experimental methodology, described previously. We first conducted pilot studies to determine the size of words and n -grams, and the number of n -grams relevant for prediction. Once determined, we applied all of the classification methods to a small collection of executables. We then applied the methodology to a larger collection of executables, all of which we describe in the next three sections.

6.1 Pilot Studies

We conducted pilot studies to determine three quantities: the size of n -grams, the size of words, and the number of selected features. Unfortunately, due to computational overhead, 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.) We then selected the best 10, 20, \dots , 100, 200, \dots , 1000, 2000, \dots , 10,000 n -grams, and evaluated the performance of a SVM, boosted SVMs, naive Bayes, 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 $n = 1, 2, \dots, 10$, 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.

6.2 Experiment with a Small Collection

Processing the small collection of executables produced 68,744,909 distinct n -grams. Following our experimental methodology, we used 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 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 7.

6.3 Experiment with a Larger Collection

With success on a small collection, we turned our attention to evaluating the text-classification methods on a larger collection of executables. As mentioned previously, this collection consisted of 1971 benign executables and 1651 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 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 (AUC) with 95% confidence intervals. As one can see, boosted J48 outperformed all other methods. Other methods, such as *IBk* and boosted SVMs, performed comparably, but the ROC curve for boosted J48 dominated all others.

7. DISCUSSION

To date, our results suggest that methods of text classification are appropriate for detecting malicious executables in the wild. Boosted classifiers, *IBk*, and a support vector machine performed exceptionally well given our current data collection. That the boosted classifiers generally outperformed single classifiers echoes the conclusion of several empirical studies of boosting [4, 6, 9, 14], which suggest that boosting improves the performance of unstable classifiers, such as J48, by reducing their bias and variance

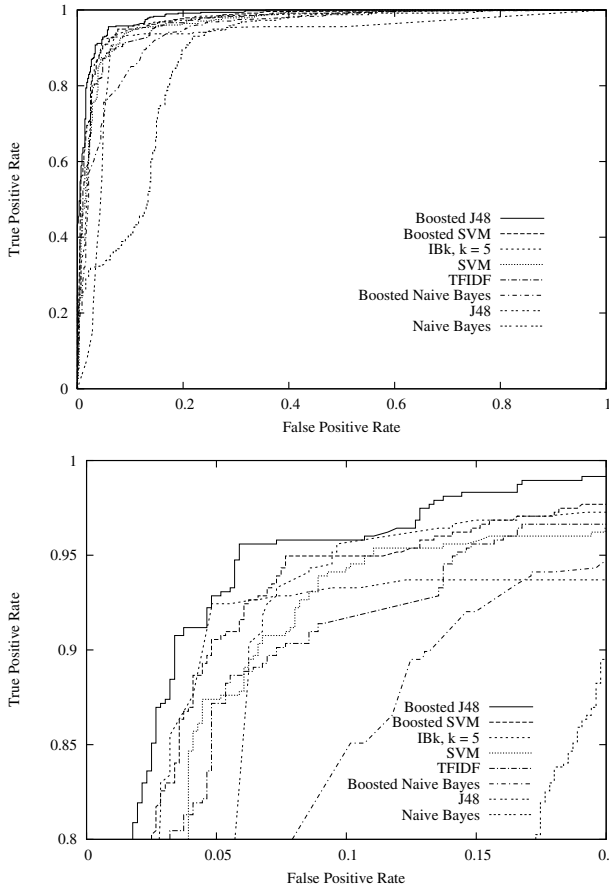


Figure 1: ROC curves for detecting malicious executables in the small collection. Top: The entire ROC graph. Bottom: A magnification.

Table 2: Results for detecting malicious executables in the small collection. Areas under the ROC curve (AUC) with 95% confidence intervals.

Method	AUC
Naive Bayes	0.8850±0.0247
J48	0.9235±0.0204
Boosted Naive Bayes	0.9461±0.0170
TFIDF	0.9666±0.0133
SVM	0.9671±0.0133
IBk, $k = 5$	0.9695±0.0129
Boosted SVM	0.9744±0.0118
Boosted J48	0.9836±0.0095

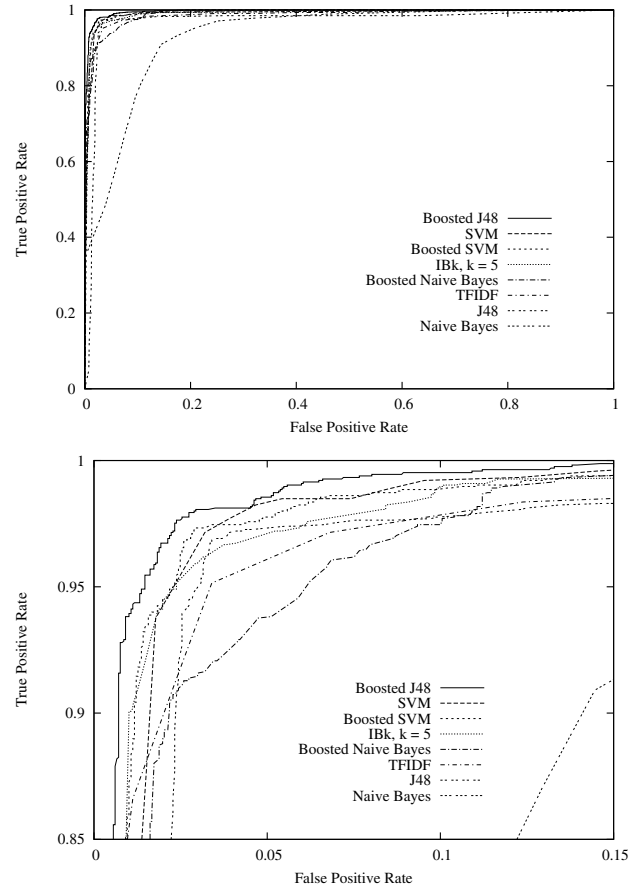


Figure 2: ROC curves for detecting malicious executables in the larger collection. Top: The entire ROC graph. Bottom: A magnification.

Table 3: Results for detecting malicious executables in the larger collection. Areas under the ROC curve (AUC) with 95% confidence intervals.

Method	AUC
Naive Bayes	0.9366±0.0099
J48	0.9712±0.0067
TFIDF	0.9868±0.0045
Boosted Naive Bayes	0.9887±0.0042
IBk, $k = 5$	0.9899±0.0038
Boosted SVM	0.9903±0.0038
SVM	0.9925±0.0033
Boosted J48	0.9958±0.0024

[4, 6]. Boosting can adversely affect stable classifiers [4], 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 [6].

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. The relative performance of the methods changed somewhat. For example, the SVM moved from fourth to second, displacing the boosted SVMs and *IBk*.

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. For instance, one short branch of a decision tree indicated that any executable with two PE headers is malicious. After analysis of our collection of malicious executables, we discovered two executables that contained another executable. While this was an interesting find, 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, one n -gram 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.

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.

To place our results in context with the study of Schultz et al. [37], 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, 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. [37] 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 dependent attributes [10]. 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 [37] 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) because `labroc4` [28] 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 [34] and cost curves [11], 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 [42] Java implementations executed quickly on the training examples with their 500 binary attributes.

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 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 completely execute 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.

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 in parallel. 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* or the *TFIDF* classifier 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. However, it remains an open issue whether an executable's statistical properties are predictive of its functional characteristics, an issue we are currently investigating and one we discuss briefly in the concluding section.

8. CONCLUDING REMARKS

We considered the application of techniques from information retrieval and text classification to the problem of detecting unknown malicious executables in the wild. After evaluating a variety of classification methods, results suggest that boosted J48 produced the best classifier with an area under the ROC curve of 0.996. Our methodology resulted in a fielded application called MECS, the Malicious Executable Classification System, which we have delivered to the MITRE Corporation.

In future work, we plan to investigate a classification task in which methods determine the *functional characteristics* of malicious executables. Detecting malicious executables is important, but after detection, computer-forensic experts must determine the program's functional characteristics: Does it mass-mail? Does it modify system files? Does it open a backdoor? This will entail removing obfuscation, such as compression, if possible. Furthermore, most malicious executables perform multiple functions, so each training example will have multiple class labels, a problem that arises in bioinformatics and in document classification.

We anticipate that MECS, the Malicious Executable Classification System, is but one step in an overall scheme for detecting and classifying "malware." When combined with approaches that search for known signatures, 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 pursuing

the classification of executables into functional categories will provide another.

9. 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. The authors also thank the anonymous reviewers for their time and useful comments, Ophir Frieder of IIT for help with the vector space model, Abdur Chowdhury of AOL 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 Georgetown for helpful comments on 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.

10. REFERENCES

- [1] D. Aha, D. Kibler, and M. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.
- [2] 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.
- [3] Anonymous. *Maximum Security*. Sams Publishing, Indianapolis, IN, 4th edition, 2003.
- [4] B. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36:105–139, 1999.
- [5] B. 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.
- [6] L. Breiman. Arcing classifiers. *The Annals of Statistics*, 26:801–849, 1998.
- [7] 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.
- [8] 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.
- [9] T. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40:139–158, 2000.
- [10] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.
- [11] C. Drummond and R. 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.
- [12] 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.
- [13] E. Durning-Lawrence. *Bacon is Shake-speare*. The John McBride Company, New York, NY, 1910.
- [14] Y. Freund and R. 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.
- [15] A. Gray, P. Sallis, and S. 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.
- [16] D. Grossman and O. Frieder. *Information retrieval: Algorithms and heuristics*. Kluwer Academic Publishers, Boston, MA, 1998.
- [17] D. Hand, H. Mannila, and P. Smyth. *Principles of data mining*. MIT Press, Cambridge, MA, 2001.
- [18] A. Jain, R. Duin, and J. Mao. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:4–37, 2000.
- [19] H. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31:1–8, 1988.
- [20] 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-Verlag.
- [21] J. Kephart, G. Sorkin, W. Arnold, D. Chess, G. Tesauro, and S. 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.
- [22] B. Kjell, W. Woods., and O. Frieder. Discrimination of authorship using visualization. *Information Processing and Management*, 30:141–150, 1994.
- [23] I. Krsul. Authorship analysis: Identifying the author of a program. Master’s thesis, Purdue University, West Lafayette, IN, 1994.
- [24] 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.
- [25] R. Lo, K. Levitt, and R. Olsson. MCF: A malicious code filter. *Computers & Security*, 14:541–566, 1995.
- [26] M. Maron and J. Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM*, 7:216–244, 1960.
- [27] G. McGraw and G. Morisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, pages 33–41, September/October 2000.
- [28] C. Metz, Y. Jiang, H. MacMahon, R. Nishikawa, and X. Pan. ROC software. Web page, Kurt Rossmann Laboratories for Radiologic Image Research, University of Chicago, Chicago, IL, 2003.
- [29] P. Miller. hexdump 1.4. Software, <http://gd.tuwien.ac.at/softeng/Aegis/hexdump.html>, 1999.
- [30] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, 1997.
- [31] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.
- [32] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and S. Mika, editors, *Advances in Kernel Methods—Support Vector Learning*. MIT Press, Cambridge, MA, 1998.
- [33] J. Platt. Probabilities for SV machines. In P. Bartlett, B. Schölkopf, D. Schuurmans, and A. Smola, editors, *Advances in Large-Margin Classifiers*, pages 61–74. MIT Press, Cambridge, MA, 2000.
- [34] F. Provost and T. Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42:203–231, 2001.
- [35] J. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Francisco, CA, 1993.
- [36] 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.
- [37] M. Schultz, E. Eskin, E. Zadok, and S. 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.
- [38] 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.
- [39] E. Spafford and S. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12:585–595, 1993.
- [40] J. Swets and R. Pickett. *Evaluation of diagnostic systems: Methods from signal detection theory*. Academic Press, New York, NY, 1982.
- [41] G. Tesauro, J. Kephart, and G. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11:5–6, August 1996.
- [42] I. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, San Francisco, CA, 2000.
- [43] Y. Yang and J. 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.