Static Analysis of Malicious Java Applets

Conference Paper · March 2016

DOI: 10.1145/2875475.2875477

CITATIONS
11

READS
442

5 authors, including:

Fabio Di Troia
San Jose State University
77 PUBLICATIONS **1,467** CITATIONS
SEE PROFILE

Corrado Aaron Visaggio
University of Sannio
156 PUBLICATIONS **5,047** CITATIONS
SEE PROFILE

Thomas H. Austin
San Jose State University
50 PUBLICATIONS **1,675** CITATIONS
SEE PROFILE

Mark Stamp
San Jose State University
258 PUBLICATIONS **5,259** CITATIONS
SEE PROFILE

Static Analysis of Malicious Java Applets

Nikitha Ganesh
Department of Computer Science
San Jose State University
San Jose, California
nikithaganesh@gmail.com

Fabio Di Troia
Department of Engineering
Università degli Studi del Sannio
Benevento, Italy
fabioditroia@msn.com

Visaggio Aaron Corrado
Department of Engineering
Università degli Studi del Sannio
Benevento, Italy
visaggio@unisannio.it

Thomas H. Austin
Department of Computer Science
San Jose State University
San Jose, California
taustin22@gmail.com

Mark Stamp
Department of Computer Science
San Jose State University
San Jose, California
mark.stamp@sjsu.edu

ABSTRACT

In this research we consider the problem of detecting malicious Java applets, based on static analysis. Dynamic analysis is often more informative, since it is immune to many common obfuscation techniques, while static analysis is often more efficient. Consequently, static analysis is generally preferred, provided we can obtain results comparable to those obtained using dynamic analysis. We conducted experiments using three techniques that have been employed in previous studies of metamorphic malware. We show that our static approach can detect malicious Java applets with greater accuracy than previously published research that relied on dynamic analysis.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive Software

Keywords

Malware; Applets; Hidden Markov models

1. INTRODUCTION

As with most Java applications, Java applets typically consist of a set of compiled Java classes, usually bundled in a jar archive. Applets are generally embedded within HTML pages, explicitly specifying the main class as well as additional parameters that should be provided to the applet. To protect the host from unrestricted access, Java applets are subject to a security manager. Malicious applets try to disable this security manager, allowing them to access restricted resources. For example, a malicious applet might attempt to download and execute a malicious binary.

Our goal here is to perform static analysis of Java applets using three previously-developed techniques. Specifically, we test the Hidden Markov Model method from [25], the Opcode Graph Similarity technique from [17], and the Simple Substitution Distance from [18]. For all of these scores, we use bytecodes extracted from applets. We compare our results to the dynamic analysis technique in [10], using the same data sets, and we show that our static approach yields better results than the more costly dynamic analysis.

The paper is organized as follows. Section 2 contains relevant background information. Section 3 gives an overview of static and dynamic analysis. We also include a discussion of previous work involving dynamic analysis to which we are comparing our results. Section 4 gives our experimental results. Finally, Section 5 contains our conclusions and a brief discussion of future work.

2. BACKGROUND

This section provides a summary of malware detection techniques, followed by a brief discussion of code obfuscation, and an overview of Java applets. Then we discuss the three static detection techniques that are analyzed in this paper, namely, Hidden Markov Models, Bytecode Graph Similarity, and Simple Substitution Distance. Finally, we consider ROC analysis, which we use to measure the success of our experiments and to compare our results to previous work.

2.1 Malware Detection

Signature detection is the most common method that antivirus software uses to identify malware. A signature is generally a string of bits found in a file, which might include wildcards. Signature scanning consists of pattern matching [1].

Signature detection is highly effective against common types of malware, provided that a signature has been extracted. Also this method has minimal overhead for users and administrators. However, a fundamental problem with signature detection is that we can only detect malware based on known signatures, and hence even a slight variant of a known virus might be missed.

Change detection consists of looking for changes in files as a way to detect potential malware infections [1]. One advantage of change detection is that it can detect new malware. However, the number of false positives is likely to be high, and hence this approach is often combined with other techniques, such as signature scanning.

Anomaly detection relies on finding malware-like activity or characteristics [1]. A major challenge with such an approach is to determine a baseline, i.e., to determine what is normal. In addition, we have to determine how far from normal we must be before classifying code as malware. Another difficulty is that the definition of normal will tend to change over time, which implies that the detection system must adapt to such changes. The main advantage of anomaly detection is that we can potentially detect previously unknown malware.

In the research literature, a large number of advanced malware detection techniques based on static analysis have been considered. Roughly speaking, these can be classified as statistical-based [18, 22, 25], graph-based [7, 15, 17], and structural-based [2, 13, 19], or some combination thereof. For the experiments reported here, we employ the Hidden Markov Model (HMM) based detection scheme in [25], the Bytecode Graph Similarity (BGS) method in [17], and the Simple Substitution Distance (SSD) in [18]. We provide additional details on these techniques and their use in malware detection in Sections 2.4, 2.5, and 2.6, respectively.

2.2 Code Obfuscation

According to [10], the Java applets that we analyze in Section 4 use a wide variety of obfuscation techniques. Here, we discuss a few representative examples of techniques that can be used to obfuscate malware. The goal of obfuscation is typically to evade signature-based detection, but in some cases it can also serve to evade more advanced detection strategies [14, 20].

At a high level, code obfuscation relies on some combination of insertion, deletion, substitution, and transposition. In practice, it seems that transposition and insertion are widely used, with substitution used less often, and deletion occasionally used, mostly to prevent the code from growing in size over multiple generations. Here, we take a brief look at insertion, substitution, and transposition.

Inserted code can be in the form of dead code (i.e., code that never executes) or garbage code (i.e., code that executes, but has no effect). It is relatively easy to insert dead code and doing so can break signatures. In addition, if the dead code is selected from benign files, the statistical profile of the malware will more closely resemble that of the benign code, making statistical-based detection more difficult [14].

Of course, there are innumerable ways to write code that performs the same function. And since code substitutions can serve to break signatures and change the statistical profile, substitution is a potentially powerful obfuscation technique. However, it is not trivial to implement code substitution in a thorough and systematic way.

Transposition is a powerful obfuscation strategy, at least as a means of evading signature detection. In [4], it is proved that transposition alone is sufficient to defeat signature detection, and the proof is constructive, with the resulting obfuscation method being practical. However, in [23], it is shown that the transposition method from [4] is ineffective against the HMM-based technique developed in [25]. That is, a straightforward transposition technique has relatively little effect on the statistical properties of the code.

2.3 Overview of Java Applets

Java programs comes in two forms, namely, standard applications and applets [3]. Standard applications run on their own, whereas applets run inside a web page. An applet is generally a small application and it is delivered to users in the form of bytecode.

The user launches a Java applet from a web page, and the applet is then executed within a Java Virtual Machine (JVM) in a process that is separate from the web browser itself. The executable code of the applet is downloaded along with the text of the page. Once the code arrives from the server, it is executed by the browser on the local machine [26]. This allows the developer to incorporate dynamic features in Web pages.

The infrastructure required to support applets is built into the Web browser. The environment provided by the browser insulates the applet from the underlying operating system and allows it to function on any platform on which a Java-enabled browser is available. Applet code is inherently compact. Standard libraries—such as libraries for graphical user interfaces and network access—are provided on the client platform. Hence, only the code that is unique to the applet needs to be transferred across the network. A simple example of an applet is given in Figure 1.

```
import java.awt.*;
import javax.swing.*;
public class JavaApplet extends JApplet {
    public void paint(Graphics g) {
      super.paint(g);
        g.drawString("The Java Applet",
             30,20);
    }
}
```

Figure 1: Example of an Applet

There are three main parts to the applet in Figure 1, namely, the import statements, the class declaration, and the custom method (in this example, the `paint` method). Import statements allow the programmer to reuse code that has already been written. For example, if we want to display text on the screen, then we can use existing code to draw things on the screen by simply referencing the code; see, for example, the `Graphics` class in Figure 1.

Each class that we create needs a class declaration. In the example in Figure 1, the name of the class is `JavaApplet`. Because we are developing applets, we have to specify that our class is an applet. To do so, we say that our class `extends` the `JApplet` class. Then this new class inherits the functionality of an applet. It is important that the applet

has `public` access specified to ensure that we can run the program directly, since applets do not have a main function.

As previously mentioned, applets are intended to be embedded within web pages. The embedding HTML file needs to be in a separate file from the applet source code. Figure 2 gives HTML code to run the applet in Figure 1.

```
<html>
   <body>
      <applet code="JavaApplet.class"
           width=400 height=500>
      </applet>
   </body>
</html>
```

Figure 2: HTML to Display Applet

2.4 Hidden Markov Models

As the name suggests, a Hidden Markov Model (HMM) consists of a Markov process, where the underlying states are hidden. In an HMM, we have a series of observations that are related to the hidden states by fixed probability distributions. An HMM can be viewed as machine learning technique and an HMM can also be viewed as a discrete hill technique, since hill climbing is used in the training phase where we determine the model parameters, based on the observations.

The notation in Table 1 is used to describe an HMM [21]. We denote an HMM as $\lambda = (A, B, \pi)$. The matrices A, B, and π are row-stochastic, that is, each row satisfies the conditions of a discrete probability distribution.

Table 1: Hidden Markov Model Notation

Notation	Explanation
T	length of the observation sequence
N	number of states in the model
M	number of observation symbols
Q	states of Markov process, $q_0, q_1, \ldots, q_{N-1}$
V	possible observations, $0, 1, \ldots, M-1$
X	hidden state sequence, $X_0, X_1, \ldots, X_{T-1}$
A	state transition probabilities
B	observation probability matrix
π	initial state distribution
$\mathcal{O}$	observation sequence, $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}$

A generic view of an HMM is given in Figure 3. Note that the A matrix drives the Markov process, while B contains the probability distributions that relate the observations to the underlying (and hidden) Markov process.

HMMs are of great practical utility primarily because there are efficient algorithms to solve each of the following problems [21].

- Problem 1: Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, compute $P(\mathcal{O}|\lambda)$. Here, given the model, we want to determine the likelihood of the observed sequence $\mathcal{O}$. We can interpret the result as a score that measures how well $\mathcal{O}$ fits the given model.

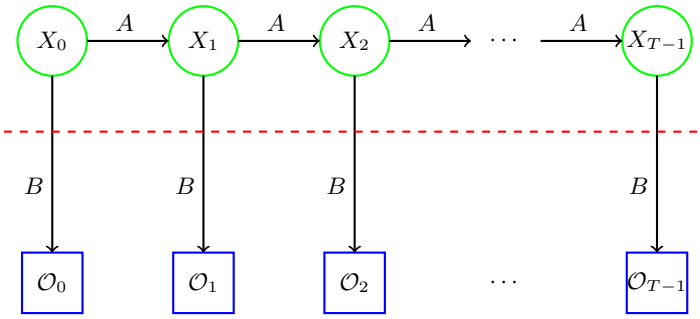

Figure 3: Illustration of HMM

- Problem 2: Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, determine an optimal state sequence for the Markov model. That is, the most likely hidden state sequence can be uncovered.

- Problem 3: Given an observation sequence $\mathcal{O}$ and the dimensions N and M, determine a model λ that maximizes the probability of $\mathcal{O}$. That is, we can train a model to fit an observation sequence.

In this research, we use the solutions to Problems 1 and 3, as follows. First, we extract bytecode sequences from a collection of malicious applets that are known to belong to a given family. We append the sequences to yield one long bytecode sequence. Then we select N and train an HMM on this bytecode sequence (using the solution to Problem 3). Given an applet that we want to score, we extract its bytecode sequence and score it against this HMM model (using the solution to Problem 1). A high score indicates that the applet in question matches the family used to train the model, while a low score indicates that the applet is not "close" to the applets in the family. In this way, we can classify applets as being members of a specific family of malicious applets, or not.

For additional details on HMMs in general, see [16, 21]. For more information on HMMs as used in malware research, see [25] or the report [9].

2.5 Bytecode Graph Similarity

Bytecode Graph Similarity (BGS) is a graph-based scoring technique based on extracted bytecode sequences [17]. In this technique, a graph is constructed based on the digram frequency of bytecode sequences from malicious applets belonging to a specific family. We refer to this as the family graph.

A similar process is used to construct a graph for a given applet that we want to score. We refer to this as the applet graph. A score is generated by calculating the distance between the applet graph and the family graph.

Given an applet, the sequence of bytecodes is extracted and a weighted directed applet graph is constructed based on

digraph frequencies. Next, we give an example to illustrate this graph construction process.

First, we extract the bytecode sequence from the given applet. Suppose, for example, that we obtain the bytecode sequence in Table 2 from an applet.

Table 2: Bytecode Sequence

Number	Bytecode	Number	Bytecode
1	aload	11	iconst
2	iconst	12	ldc
3	ldc	13	sipush
4	istore	14	iconst
5	sipush	15	aload
6	aload	16	aload
7	ldc	17	aload
8	iconst	18	ldc
9	iconst	19	iconst
10	istore	20	istore

Table 3 contains a matrix derived from the digraph counts of the sequence in Table 2. For example, the value in row 1 and column 3 is 2, since the bytecode `ldc` is followed by `iconst` twice in Table 2.

Table 3: Bytecode Count

	ldc	aload	iconst	sipush	istore
ldc	0	0	2	1	1
aload	2	2	1	0	0
iconst	2	1	1	0	2
sipush	0	1	1	0	0
istore	0	0	1	1	0

Next, we convert the counts in Table 3 to probabilities by dividing each element by its corresponding row sum (provided the row sum is greater than zero). That is, we convert the count matrix into a row-stochastic probability matrix. In this example, the probability matrix corresponding to Table 3 is given in Table 4.

Table 4: Probability Matrix

	ldc	aload	iconst	sipush	istore
ldc	0	0	1/2	1/4	1/4
aload	2/5	2/5	1/5	0	0
iconst	1/3	1/6	1/6	0	1/3
sipush	0	1/2	1/2	0	0
istore	0	0	1/2	1/2	0

The probability matrix thus constructed can be viewed as a bytecode graph, where each node represents a unique bytecode. In this bytecode graph, a directed edge is drawn from one bytecode to another, provided there is a non-zero probability in the corresponding position of the probability matrix. Edge weights are assigned to the graph directly from the probability matrix. For example, the bytecode graph corresponding to the probability matrix in Table 4 is given in Figure 4.

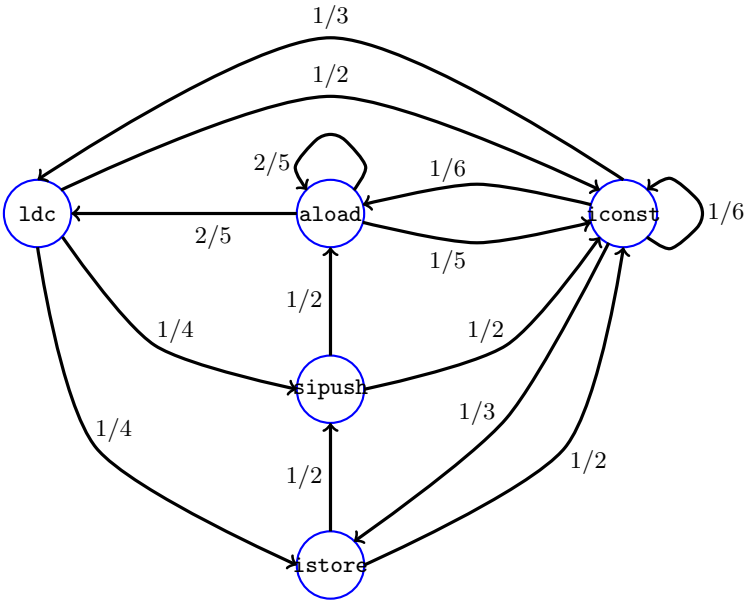

Figure 4: Bytecode Graph

To construct a family graph, the same process is used, except that bytecode digraph statistics for a set of applets belonging to the same family are used. The process of constructing the family graph can be viewed as the training phase. Given the extracted bytecode sequences, training is extremely efficient.

Scoring a given applet against a specific family is also very efficient. We construct the bytecode graph for the given applet, and we sum the absolute differences of its probability matrix with that of the family graph. The score is always non-negative, and an exact match yields a score of zero. Thus, the smaller the score, the better the applet matches the statistics of the family, and hence the more likely that it belongs to the same family.

Since the bytcode graph is constructed to be row stochastic, the BGS score weights all bytecodes equally. That is, the most common bytecode has no more influence on the bytecode graph score than a rare bytecode. This is in contrast to an HMM score, for example, where bytecodes are weighted according to their relative frequencies. This emphasis on rare features can be an effective means of detecting certain malware families [17].

2.6 Simple Substitution Distance

In [18] a novel malware score based on simple substitution cryptanalysis is developed and analyzed. When using this score, we consider the file to be scored as an encrypted version of a given malware family. We then try to decrypt the file, and the better the resulting decryption, the higher the score. Note that the file is not actually encrypted, but a member of a given malware family can be viewed as on obfuscated element of the family. And, simple substitution encryption is more akin to obfuscation than strong encryption. Consequently, the decryption process is a form of de-obfuscation, and the better we can de-obfuscate, relative to

a given family, the more likely that the file is a member of the family.

The Simple Substitution Distance (SSD) method in [18] uses a fast hill climbing technique based on Jakobsen's algorithm [12]. Next, we outline Jakobsen's algorithm, in the context of a simple substitution cipher on English text. Then we provide a simple example to illustrate the scoring process in the context of bytecodes.

Suppose that we are given a ciphertext message and we now that the plaintext is English, where only the 26 letters appear. We also assume that we know the message was encrypted using a simple substitution. Then there are 26 ciphertext symbols, and each symbol maps to one of the 26 letters in English.

To begin, we select an initial putative key of the form

$$K = k_1, k_2, k_3, \ \ldots \ , k_{26}. \tag{1}$$

At each step, Jakobsen's algorithm modifies the key by swapping elements according to the "swap schedule",

$$\begin{array}{lccccccc}
\text{row 1:} & k_1|k_2 & k_2|k_3 & k_3|k_4 & \ldots & k_{23}|k_{24} & k_{24}|k_{25} & k_{25}|k_{26} \\
\text{row 2:} & k_1|k_3 & k_2|k_4 & k_3|k_5 & \ldots & k_{23}|k_{25} & k_{24}|k_{26} & \\
\text{row 3:} & k_1|k_4 & k_2|k_5 & k_3|k_6 & \ldots & k_{23}|k_{26} & & \\
\vdots & \vdots & & & \iddots & & & \\
\text{row 23:} & k_1|k_{24} & k_2|k_{25} & k_3|k_{26} & & & & \\
\text{row 24:} & k_1|k_{25} & k_2|k_{26} & & & & & \\
\text{row 25:} & k_1|k_{26} & & & & & &
\end{array} \tag{2}$$

where $k_i|k_j$ indicates that we swap elements i and j of the current putative key K.

Jakobsen's algorithm is a hill climb. Consequently, we compute a score after each swap, and we only modify the putative key in the case where the score improves. Next, we specify the scoring function.

Let $E = \{e_{ij}\}$ be a 26×26 matrix containing digraph frequencies obtained from a large sample of English text. For example, $e_{1,1}$ contains the frequency of the digraph `AA`, while $e_{20,8}$ is the frequency of the digraph `TH`. Let $D = \{d_{ij}\}$ be a 26×26 matrix containing the digraph statistics of the putative plaintext, i.e., the "plaintext" obtained when we decrypt the ciphertext using the initial putative key K in (1). Then in Jakobsen's algorithm, the score of the putative key K is computed as

$$\texttt{score}(D, E) = \sum_{i,j} |d_{ij} - e_{ij}| \tag{3}$$

Note that $\texttt{score}(D, E) \geq 0$ and a perfect match yields $\texttt{score}(D, E) = 0$. It follows that the lower the score, the better.

Pseudo-code for Jakobsen's algorithm is given in Table 5. Note that whenever the score improves, we update the key and we restart the swapping schedule in (2) from the beginning.

To compute a Simple Substitution Distance score for an applet, we first extract bytecode sequences from a large number of members of a given malicious applet family. We use these

Table 5: Jakobsen's Algorithm

```
//
// Given: Matrix E of digraph stats and ciphertext C
//
let K = (k_1, k_2, ..., k_26) = initial key obtained from C
decrypt C with K and compute digraph matrix D
let s = score(D, E) // score is defined in (3)
let a = 1
let b = 1
while b < 26 do
    i = a
    j = a + b
    K' = swap_K(k_i, k_j)
    D' = swap_D(i, j)
    s' = score(D', E)
    if s' < s then // improved score
        s = s' // update
        K = K' // update
        D = D' // update
        a = 1 // reset swapping
        b = 1 // reset swapping
    else // get next key swap
        a = a + 1
        if a + b > 26 then
            a = 1
            b = b + 1 // next row of (2)
        end if
    end if
end while
return K
```

bytecodes to construct the analog of the E matrix in Jakobsen's algorithm, that is, we construct the bytecode digraph distribution matrix. Then, given an applet that we want to classify, we extract its bytecode sequence and generate the analog of the D matrix. For both the E and D matrices, we restrict to the most frequent n bytecodes, with symbol $n+1$ used to denote any other bytecode [18]. In Section 4, we experiment with different values of n.

We choose the initial K to best match the monograph bytecode statistics of the family applets. That is, we assume that the most frequent bytecode in the family applets maps to the most frequent bytecode in the suspect applet, the second most frequent bytecode in the family maps to the second most frequent bytecode in the suspect code, and so on. We determine the initial matrix D based on "decrypting" with this initial putative key K. We normalize both the E and D matrices so that our scores are independent of the length of the bytecode sequence.

For example, suppose that the only bytecodes that appear in the family applets, arranged in descending order of frequency, are

$$\texttt{bipush}, \texttt{aload}, \texttt{ldc}, \texttt{pop}, \text{ and } \texttt{iload}$$

Next, assume that the extracted bytecode sequence from the applet we want to score is

$$\texttt{iconst}, \texttt{bipush}, \texttt{bipush}, \texttt{ldc}, \texttt{lstore}, \texttt{lstore}, \texttt{lstore} \tag{4}$$

The frequency counts for this bytecode sequence are

Bytecode	lstore	bipush	ldc	iconst
Frequency	3	2	1	1

Consequently, our initial guess for K is

Applet Family	bipush	aload	ldc	pop
File to Score	lstore	bipush	ldc	iconst

And, the result of "decrypting" the sequence (4) using this key is

$$\texttt{pop}, \texttt{aload}, \texttt{aload}, \texttt{ldc}, \texttt{bipush}, \texttt{bipush}, \texttt{bipush} \qquad (5)$$

From the sequence (5), we obtain the initial D matrix

	bipush	aload	ldc	pop	iload	other
bipush	2	0	0	0	0	0
aload	0	1	1	0	0	0
ldc	1	0	0	0	0	0
pop	0	1	0	0	0	0
iload	0	0	0	0	0	0
other	0	0	0	0	0	0

After scoring this D matrix versus the E matrix, the next step in Jakobsen's algorithm is to swap the first two bytecodes in the putative key, that is, we swap `bipush` and `aload`. The naïve algorithm would consist of again decrypting using this new key, and computing the new D matrix. However, according to Jakobsen's algorithm, the resulting D matrix can be obtained by simply swapping the first two rows and columns of the current D matrix. Thus, the updated D matrix is

	bipush	aload	ldc	pop	iload	other
bipush	1	0	1	0	0	0
aload	0	2	0	0	0	0
ldc	0	1	0	0	0	0
pop	1	0	0	0	0	0
iload	0	0	0	0	0	0
other	0	0	0	0	0	0

Therefore, in Jakobsen's algorithm, we "decrypt" the sequence once—all subsequent score computations only require elementary matrix manipulations. The score for the final D matrix gives the SSD score for the bytecode sequence.

2.7 ROC Curves

In this research, we use the area under the ROC curve (AUC) to quantify the success of experiments, and to compare our results to previous work. Given a scatterplot, an ROC curve is obtained by plotting the false positive rate (FPR) against the true positive rate (TPR) as the threshold varies through the range of data values. An AUC of 1.0 implies that there exists a threshold that results in no false positives or false negatives, which is the ideal case. In general, the AUC can be interpreted as the probability that a randomly selected positive instance scores higher than a randomly selected negative instance [5]. Therefore, an AUC of 0.5 means that the binary classifier is no better than flipping a coin. Also, an AUC of p will yield an AUC of $1 - p$ if we simply flip the classification criteria and, consequently, the AUC can never be less than 0.5.

An example of a scatterplot and the corresponding ROC curve is given in Figure 5. The red circles in the scatterplot represent positive instances, while the blue squares represent negative instances. In the context of malware classification, the circles are scores for malware files, while the squares are scores for benign files. Furthermore, we assume that higher scores are "better", that is, for this particular score, positive instances are supposed to score higher than negative instances.

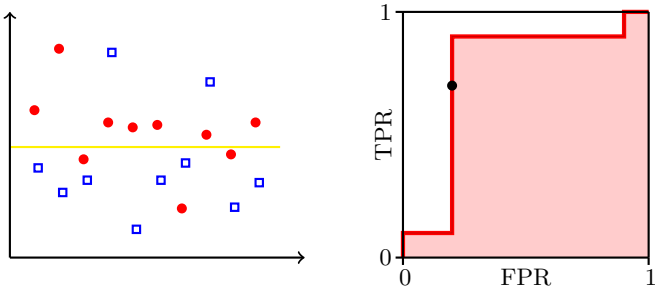

Figure 5: Scatterplot and ROC Curve

Note that if we place the threshold below the lowest point in the scatterplot in Figure 5, then

$$\text{TPR} = 1 \quad \text{and} \quad \text{FPR} = 1$$

On the other hand, if we place the threshold above the highest point, then

$$\text{TPR} = 0 \quad \text{and} \quad \text{FPR} = 0$$

Consequently, an ROC curve must always include the points $(0, 0)$ and $(1, 1)$. The intermediate points on the ROC curve are determined as the threshold passes through the range of values. For example, if we place the threshold at the yellow line in the scatterplot in Figure 5, the true positive rate is 0.7, since 7 of the 10 positive instances are classified correctly, while the false positive rate is 0.2, since 2 of the 10 negative cases lie on the wrong side of the threshold. This gives us the point $(0.2, 0.7)$ on the ROC curve, which is illustrated by the black circle on the ROC graph in Figure 5. The shaded region in Figure 5 represents the AUC which, in this example is 0.75.

3. PREVIOUS WORK

In this section, we first briefly discuss the tradeoffs between static analysis and dynamic analysis in the context of malware detection. Then we turn our attention to the main topic of the section, namely, HoneyAgent, a dynamic analysis tool that has previously been applied to the problem of detecting malicious Java applets [10]. In Section 4, we compare detection results obtained using static techniques to those obtained using HoneyAgent.

3.1 Static and Dynamic Analysis

Static analysis is the evaluation of code by means that do not require execution. For example, compiler optimizations rely on static analyses [8]. The static analysis techniques

that we consider in Section 4 rely entirely on information obtained from bytecode sequences extracted from applets.

In contrast to static analysis, dynamic analysis consists of testing and evaluation of an application at runtime. Dynamic analysis operates by executing (or emulating) a program and observing the results. If the code is executed (as opposed to being emulated), dynamic analysis is precise, in the sense that no approximation or abstraction needs to be done—we can examine actual run-time behavior of the program. However, it is unlikely that dynamic analysis can test all branches and hence some parts of the code will almost certainly remain out of view during such analysis.

In terms of malware detection, static analysis is generally more efficient than dynamic analysis. On the other hand, dynamic analysis defeats some types of obfuscation. For example, dead code will disappear when dynamic analysis is employed, but can be very difficult to automatically detect during static analysis. Due to its efficiency, static analysis is preferred, provided it is as effective as dynamic analysis.

3.2 HoneyAgent

The developers of HoneyAgent [10] suggest that reliable detection of obfuscated Java applets requires dynamic analysis. This seems plausible, since dynamic analysis can, in effect, strip away one layer of obfuscation. However, a wide variety of static techniques have shown strong results when applied to highly obfuscated metamorphic malware [2, 13, 15, 17, 18, 19, 22, 25]. Our goal in this research is to compare the static techniques in [17, 18, 25] to the dynamic detection results obtained using HoneyAgent in [10]. The HMM-based technique in [25] has served as a benchmark in many other studies of detection techniques, while the techniques in [17] and [18] are superior in some cases.

HoneyAgent [10] is designed to analyze the runtime behavior of Java applets. It observes the interaction between a Java applet and the default Java Runtime Environment to detect actions such as file downloads, changes to the file system or process creation. By intercepting the respective function calls, HoneyAgent prevents changes to its host system while remaining invisible to the applet being analyzed. The tool also simulates the effect of successful exploitation of some common vulnerabilities, allowing for a more comprehensive analysis of the malicious applet. By applying a small set of heuristics to observed runtime behavior, HoneyAgent can identify malicious Java applets without human intervention.

At runtime, an applet interacts with the runtime environment. Thus, in any dynamic analysis system, it is necessary to provide an environment resembling the one an applet would expect on a vulnerable system, as well as to hide the code used for analysis as far as possible. This is the major challenge in dynamic analysis of Java applets. In contrast, no such environment is necessary in the case of static analysis, which makes for a much simpler process.

HoneyAgent is implemented as a Java agent [10]. Thus it is able to perform all interactions required to trace the behavior of an examined Java applet. The agent library cannot be accessed from within the Java Virtual Machine (JVM), which hinders malicious Java applets from detecting the presence of HoneyAgent.

HoneyAgent includes a variety of advanced features. For example, it intercepts all calls to API methods performed by the examined applet and reconstructs the parameters used by inspecting the current stack frame. This allows HoneyAgent to determine the behavior of the examined applet without requiring the massive runtime overhead introduced by breaking on every method call.

Java applets are executed within a sandbox, which thereby restricts the actions of the applet within the host. Malicious applets might try to interact with the host to, for example, install a malicious binary. Thus, HoneyAgent considers an applet malicious if it tries to perform such a prohibited activity.

In the HoneyAgent paper [10], only unsigned applets are considered for analysis. Applets signed with a valid and trusted certificate are executed without any security restrictions just like classic Java applications. Therefore, these applets would be allowed to perform functions that would otherwise be indicative of a malicious applet. Here, we use the same datasets as in [10].

3.3 Datasets

One set of malicious applets analyzed in [10] was obtained from VirusTotal [24]. The other dataset was taken from the Contagio malware dump [6]. For the research in this paper, we have used the same malicious datasets and the same benign dataset as in [10]. The number of files in each of these datasets is given in Table 6.

Table 6: Datasets

Dataset	Files	Bytecodes	
		Total	Per File
Contagio	105	1,639,346	15,612
VirusTotal	105	874,943	8332
Benign	98	1,082,612	11,047

According to [10], HoneyAgent was able to successfully detect malicious behavior in 94% of all samples from the Contagio dataset and 97% in the VirusTotal dataset, without generating any false positives for the benign applets. This yields in an overall detection rate of 96%. In the next section, we show that we can do somewhat better overall using an approach based only on features extracted via static analysis.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results for three static detection techniques applied to Java applets. Specifically, we consider the Hidden Markov Model, Bytecode Graph Similarity, and Simple Substitution Distance scores that were discussed in detail in Sections 2.4, 2.5, and 2.6, respectively. We compare our results to those obtained using the dynamic analysis tool, HoneyAgent [10]. Then we attempt to improve on these scoring results by reducing the number of bytecode categories based on edit distance. But first we briefly describe the process used to extract the bytecodes.

4.1 Extracting Bytecodes

Java applets consist of compiled Java classes which are usually bundled in a jar archive. The command

```
jar xf <jar-file-name>
```

extracts the contents of the specified `.jar` file. We then use `javap` to disassemble the Java `.class` files into bytecodes. The specific command used is

```
javap -c <class-file-name>.class
```

Figure 6 gives an example of the resulting output, from which the bytecodes (highlighted in blue) are easily extracted.

```
public DocFooter();
   code:
      0:  aload_0
      1:  invokespecial  #1   // java/applet/Applet ...
      4:  return

public void init();
   code:
      0:  aload_0
      1:  sipush         500
      4:  bipush         100
      6:  invokevirtual  #2   // Method resize:(II)V
      9:  aload_0
     10:  aload_0
     11:  ldc            #3   String LAST_UPDATED
     13:  invokevirtual  #4   Method getParameter ...
     16:  putfield       #5   date:Ljava/lang/String;
     19:  aload_0
     20:  aload_0
     21:  ldc            #6   String EMAIL
     23:  invokevirtual  #4   Method getParameter ...
     26:  putfield       #7   email:Ljava/lang/String;
     29:  return
```

Figure 6: Output from `javap -c JavaApp.class`

4.2 HMM Score

For these experiments, we use 5-fold cross validation. That is, we partition the dataset under consideration into five equal-sized subsets, say, S_1, S_2, S_3, S_4, and S_5. Then we train an HMM on sets S_1, S_2, S_3, and S_4. The resulting HMM is used to score the files in S_4 and all of the benign files. This process is then repeated four more times, with a different subset S_i reserved for testing in each "fold". The results of the five folds are combined into one scatterplot, an ROC curve is generated, and the AUC is computed (see Section 2.7 for a discussion of ROC analysis). Cross validation serves to smooth out any biases in the data, and also maximizes the number of score calculations for a given size of dataset.

For the HMM score, these initial experiments yield the AUC values in Table 7. From these results, we see that the HMM performs better than HoneyAgent on the Contagio data, but much worse on the VirusTotal data.

4.3 BGS Score

Next, we consider the Bytecode Graph Similarity score, as described in Section 2.5. The experimental process used here is similar to that for the HMM score, as discussed above. However, for the BGS score, recall that we need to select the parameter n, the number of highest-frequency bytecodes used to construct the graph (with all remaining bytecodes grouped together in the "other" category, giving $n+1$ categories). We experimented with various values of n to obtain the results in Table 8.

Table 7: Hidden Markov Model AUC

Family	AUC	
	HMM	HoneyAgent
Contagio	0.95	0.94
VirusTotal	0.77	0.97

Table 8: Bytecode Graph Similarity AUC

n	Contagio	VirusTotal
20	0.9843	0.9697
30	0.9895	0.9766
40	0.9886	0.9710
50	0.9682	0.9430
60	0.9171	0.9086
70	0.8509	0.8594

From Table 8, we see that the optimal result for the Contagio dataset in 0.9895, while the optimal result for VirusTotal is 0.9766. The comparable HoneyAgent numbers are given in Table 7. The BGS score performs better than HoneyAgent on both datasets.

4.4 SSD Score

For the Simple Substitution Distance score, the training and scoring process is similar to that used for the HMM and BGS scores. As in the BGS score, we optimize over the parameter n, which is the number of the most frequent bytecodes used for scoring. Again, all bytecode outside of these top n are lumped together in the "other" category. In this way, the resulting scoring matrices will be less sparse, and previous work has shown that this generally results in significantly stronger scores [18].

A summary of the AUC values for the SSD score over a range of choices of n is given in Table 9. In this case, the best result for Contagio is 0.8485, while the best result for VirusTotal is 0.9451. The comparable HoneyAgent numbers are given in Table 7. The SSD score performs significantly worse than HoneyAgent on the Contagio dataset, and slightly worse on the VirusTotal dataset.

Table 9: Simple Substitution Distance AUC

n	Contagio	VirusTotal
20	0.6784	0.8573
30	0.8485	0.8615
40	0.8288	0.9169
50	0.7686	0.9451
60	0.6916	0.9314
70	0.7782	0.9151
80	0.7068	0.8158

4.5 Improved Results Using Edit Distance

In an effort to improve further on the BGS and SSD scores, we experimented with grouping bytecodes together based on edit distance (also known as Levenshtein distance). The objective is to provide a simple means to group similar bytecodes, thereby reducing the dimension of the training problem and, hopefully, increasing the strength of the resulting models.

Edit distance is a numeric value that expresses the minimum number of transformation required to convert one specific string to another specific string. In our case, the allowed transformation are insertion, deletion, and substitution. For example, we can convert `books` to `broom` by the following series of allowed operations:

`books` → `brooks` (insert `r`)
`brooks` → `brooms` (substitute `k` for `m`)
`brooms` → `broom` (delete `s`)

This result shows that the edit distance between `books` to `broom` is at most 3 and, in fact, it is fairly easy to show that no fewer than three transformations will suffice. Consequently, the edit distance between `books` to `broom` is exactly 3. For additional information on this topic, including an efficient algorithm to compute edit distance, see [11].

In our experiments, an edit distance of k means that we group together any opcodes that differ—in the sense of insertions, deletions, and substitutions—by k or fewer edit operations. For example, the bytecodes `iconst_0`, `iconst_1`, and `iconst_2` all load values (0, 1, and 2, respectively) onto the stack, so it would seem plausible to group them together. And, these bytecodes are all close to each other, with respect to edit distance.

Results for our edit distance experiments with the BGS score are given in Table 10. We see that over the range of values tested, the best results for the BGS score are 0.9763 for Contagio and 0.9788 for VirusTotal. These results do not improve on those obtained without grouping bytecodes, as given in Table 8.

Table 10: AUC for BGS with Edit Distance

Bytecodes	Contagio	VirusTotal
20	0.9565	0.9621
25	0.9654	0.9695
30	0.9703	0.9733
35	0.9731	0.9758
40	0.9746	0.9771
45	0.9755	0.9781
50	0.9763	0.9788

Results for our edit distance experiments with the SSD score are given in Table 11. In these experiments, the best results are 0.9445 for Contagio and 0.9357 for VirusTotal. As can be seen by comparing to Table 9, the result for the Contagio dataset has improved from 0.8485 to 0.9445. This shows that in some cases, grouping similar opcodes can yield a large improvement in scoring effectiveness.

Table 11: AUC for SSD with Edit Distance

Bytecodes	Contagio	VirusTotal
20	0.8309	0.8655
30	0.8938	0.9311
31	0.9158	0.9284
32	0.9380	0.9301
33	0.9445	0.9355
34	0.9284	0.9328
35	0.9233	0.9357
36	0.9049	0.9338
40	0.9109	0.9338
45	0.8781	0.8992
50	0.8776	0.8983

4.6 Static vs Dynamic

A comparison of our best static results with those obtained with HoneyAgent is given in the form of a bar graph in Figure 7. Although the dynamic HoneyAgent technique achieves strong results, our experimental results clearly show that static techniques can perform as well, if not better.

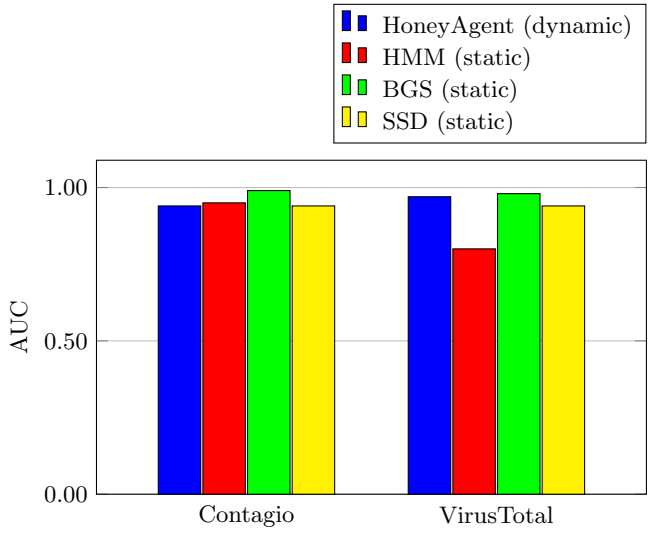

Figure 7: Static Analysis vs Dynamic Analysis of Malicious Java Applets

5. CONCLUSION AND FUTURE WORK

In this paper, we compared static malware detection techniques based on Hidden Markov Models [25], Bytecode Graph Similarity [17], and Simple Substitution Distance [18], to a dynamic scoring technique known as HoneyAgent [10]. These static scoring techniques were tested on the same malicious Java applets as in the HoneyAgent paper. Our results show that the static scores produce better results than those obtained using the dynamic HoneyAgent technique. The Btyecode Graph Similarity score is particularly impressive in our tests. These results indicate that the additional cost of dynamic analysis is unlikely to be warranted, at least for these particular datasets. Of course, there may be more challenging datasets for which the dynamic score yields significantly better results than a static score. In any case, there is a fairly substantial body of research indicating

that various static scoring techniques perform well against advanced metamorphic malware [2, 7, 13, 15, 17, 18, 19, 22, 25], so it is not too surprising that these techniques give strong results when applied to obfuscated Java applets.

For future work, it would be worthwhile to experiment with combinations of the HMM, BGS, and SSD scores—such combinations could likely perform significantly better than the individual scores. Support Vector Machines (SVM) would provide a straightforward technique for generating optimized score combinations. In addition, some of the other static scores cited in the previous paragraph could be analyzed in the context of Java applets. Some of these scores have been shown to do better in certain cases than the scores considered in this paper.

It would be very interesting to find a malware dataset for which a dynamic technique such as HoneyAgent clearly outperforms the static approaches analyzed in this paper. And, if such a real-world malware dataset cannot be found, it would be worthwhile to construct such a dataset, since this exercise would likely provide a better understanding of the relative strengths and weaknesses of static and dynamic analysis.

6. REFERENCES

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

[2] D. Baysa, R. M. Low, and M. Stamp, Structural entropy and metamorphic malware, *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, November 2013

[3] E. S. Boese, *An Introduction to Programming with Java Applets*, third edition, Jones & Bartlett, 2009

[4] J. Borello and L. Me, Code obfuscation techniques for metamorphic viruses, *Journal in Computer Virology*, 4(3):211–220, 2008

[5] A. P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms, *Pattern Recognition*, 30(7):1145–1159, 1997

[6] Contagio malware dump, `http://contagiodump.blogspot.com/`

[7] P. Deshpande, Metamorphic detection using function call graph analysis, submitted

[8] M. D. Ernst, Static and dynamic analysis: Synergy and duality, 2003, `https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf`

[9] N. Ganesh, Static analysis of malicious Java applets, Master's thesis, Department of Computer Science, San Jose State University, 2015

[10] J. Gassen and J. P. Chapman, HoneyAgent: Detecting malicious Java applets by using dynamic analysis, *Proceeding of the 9th International Conference on Malicious and Unwanted Software* (MALCON 2014), Fajardo, Puerto Rico, October 28–30, 2014

[11] M. Islita, Levenshtein edit distance, 2006, `http://www.miislita.com/searchito/levenshtein-edit-distance.html`

[12] T. Jakobsen, A fast method for the cryptanalysis of substitution ciphers, *Cryptologia*, 19:265–274, 1995

[13] J. Lee, T. H. Austin, and M. Stamp, Compression-based analysis of metamorphic malware, to appear in *International Journal of Security and Networks*

[14] D. Lin and M. Stamp, Hunting for undetectable metamorphic viruses, *Journal in Computer Virology*, 7(3):201–214, August 2011

[15] Y. Park, D. S. Reeves and M. Stamp, Deriving common malware behavior through graph clustering, *Computers & Security*, 39(B):419–430, November 2013

[16] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, 77(2):257–286, 1989

[17] N. Runwal, R. M. Low, and M. Stamp, Opcode graph similarity and metamorphic detection, *Journal in Computer Virology*, 8(1-2):37–52, May 2012

[18] G. Shanmugam, R. M. Low, and M. Stamp, Simple substitution distance and metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, August 2013

[19] I. Sorokin, Comparing files using structural entropy, *Journal in Computer Virology* 7(4):259–265, 2011

[20] S. M. Sridhara and M. Stamp, Metamorphic worm that carries its own morphing engine, *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, May 2013

[21] M. Stamp, A revealing introduction to hidden Markov models, 2012, `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf`

[22] A. H. Toderici and M. Stamp, Chi-squared distance and metamorphic virus detection, *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, February 2013

[23] S. Venkatachalam and M. Stamp, Detecting undetectable metamorphic viruses, *Proceedings of 2011 International Conference on Security & Management* (SAM '11), pp. 340–345

[24] VirusTotal, `https://www.virustotal.com/`

[25] W. Wong and M. Stamp, Hunting of metamorphic engines, *Journal in Computer Virology*, 2(3):211–229, 2006

[26] R. M. Yorston, Presenting archaeological information with Java applets, *Proceeding of the 25th Anniversary Conference*, University of Birmingham, April 1997