Detection of obfuscation in Java malware

Renuka Kumar, Anand Raj Essar, Vaishakh

Amrita Center for Cybersecurity Systems & Networks, Amritapuri, 690525, India

Abstract

Code obfuscation was introduced as a viable technique to prevent reverse engineering of software applications. Obfuscation protects an application's key algorithms and data structures from theft by hackers. However, malware authors use the same techniques to create a malware or insert malicious logic into a legitimate application. This paper proposes an analysis system to detect lexical and string obfuscation in Java malware. We identify a set of eleven features that characterize obfuscated code, and use it to train a machine learning classifier to distinguish between obfuscated and non-obfuscated malware. The features are extracted using a static analyzer that examines bytecode. Our experimental results based on a dataset of 375 malware samples containing 182927 strings and 12721 Java classes provide an accuracy of 99%. The proposed features are effective even when a dictionary is employed for lexical obfuscation. We evaluated the robustness of our features by calculating chi-squared statistic for each feature.

Keywords: Obfuscation; Machine learning; Java; Malware; Bytecode; Static analysis; Supervised learning, Cryptanalysis

1. Introduction

The number of incidents of targeted attacks using Java based malware affecting both enterprise and personal systems has increased in the last decade. According to the 2014 IBM X-Force Threat Intelligence report, more than half of all the exploits target Java\(^1\). The 2014 annual security report by Cisco states that Java comprises 91 percent of web exploits\(^2\). In 2013, Symantec reported that attackers spread Java-based malware through email attachments in the form of Jar and Class files under the assumption that millions of computers have Java Runtime Environment installed\(^3\). In 2012, Dr. Web, a Russian anti-virus vendor, reported a Java-based malware called Flashback that infected over half a million Mac OS X computers around the world\(^4\). A variant of the IceFog APT called JavaFog
targeted big corporations and had very low detection rates\textsuperscript{5}. Malware is often obfuscated to evade detection by antivirus scanners and static analysis tools.

Obfuscation is a program transformation technique that creates a modification of the original program with the intent of hiding its logic. The original program is transformed into a semantically equivalent form that could differ in its implementation. The transformed program is more difficult to understand and hence analyze\textsuperscript{6}. While obfuscation was devised to protect applications from getting reverse engineered, attackers use it as a tool to hide their malicious code. Malware authors also use obfuscation to morph malware. Such obfuscated malware can easily evade detection by analysis tools that employ pattern matching. In addition, the degree and type of obfuscation can be easily varied to create newer malware variants. Fig. 1 shows the difference between obfuscated and non-obfuscated strings in bytecode. While the differences are easy to spot by visual inspection, automating detection is nontrivial. Following are the different types of obfuscation identified\textsuperscript{12,13,14}.

- Lexical obfuscation: This is the most commonly employed obfuscation technique wherein the class, method and field names are renamed with meaningless words or words from a dictionary to conceal their actual purpose. When a dictionary is used to obfuscate, it is also called dictionary obfuscation. Lexical renaming makes it hard to guess the intended purpose of the identifier from its name.
- Control flow obfuscation: This type of obfuscation obscures the control flow of the program by inserting opaque predicates or dead code, inlining and outlining of methods and changing the order of executing statements\textsuperscript{15}.
- Data obfuscation: This technique alters how variables are stored and interpreted in a program. Variables are also split to create additional variables.
- String encryption: Strings used in a program are encrypted, and decrypted at runtime.

In this paper, we study lexical obfuscation, the most commonly employed obfuscation technique. Lexical obfuscation renames classes, methods and fields based on a certain pattern. An example pattern is: a*b*c*d* \ldots x*y*z. The star quantifier (*) indicates that the preceding letter can occur zero or more times. Hence a lexically obfuscated identifier can be renamed as a,b,c etc. or aa,ab,ac etc. Standard Java classes, however, are not obfuscated during lexical transformation. Fig. 2 shows lexical obfuscation of methods and fields in a sample class file.

Recent research studies the effect of string obfuscation on JavaScript based webpages and Java applets\textsuperscript{8,9,18}. They show results based on n-gram, a text-based feature, to detect abnormal strings in JavaScript webpages. Such
webpages could bypass signature-based detection. NOFUS is a static classifier that uses abstract syntax trees to classify obfuscated versus non-obfuscated JavaScript code. Arora, et.al discusses a static heuristics based technique to detect packing in PE format files. Research has also proven the feasibility of using machine learning techniques to classify malware and detect obfuscation. We use ProGuard, Jshrink and Zelix KlassMaster, three commonly used commercial and open source obfuscators, to arrive at a generic logic for obfuscation detection.

In this paper, we propose a novel analysis system that statically extracts lexical features from obfuscated Java malware, and supplies it into a machine learning classifier in order to detect lexical and string obfuscation. We enhance existing literature in string obfuscation detection by including metrics that have been traditionally applied in cryptanalysis and text categorization, and propose additional metrics to improve classification. Our analysis system is agnostic to the obfuscation tool used and can also detect dictionary obfuscation. We tested our classifier on 375 samples containing 182927 strings and 12721 classfiles. Our evaluation showed that the detection tool works with high accuracy and yields low false negatives.

The remainder of this paper is as organized as follows. Section 2 of this paper describes related work in this area. Section 3 gives an overview of the proposed system including the feature set to the machine learning classifier. We then give an evaluation of our work in Section 4. We conclude our work in section 5.

2. Related work

Choi proposes three metrics: n-gram, entropy and word size for statically identifying string obfuscation in JavaScript web pages. They performed tests on 33 malicious web pages. Visaggio improves and proves that a weighted combination of these three metrics produces better results than individual metrics. They used four code obfuscators and extracted 1000 JavaScript strings from 1000 websites. In our proposed system, we study the effectiveness of these metrics specifically for Java malware. It has been found that the threshold values vary substantially for the metrics entropy and word size when applied on Java malware. Research literature shows very little work on automated detection for lexical and string obfuscation in Java-based malware.

Java applets are a common attack vector to deliver drive-by downloads. The Jarhead tool uses machine-learning algorithms for detecting obfuscation in such malicious applets. They extracted three features – total number of instructions, total number of classes and the number of functions per class – from strings in the applet. They also use cyclomatic complexity as a feature to detect control flow obfuscation. We restrict our detection to lexical and string obfuscation and utilize a different set of features to do so.

Machine learning has also been used as a technique to detect phishing URLs by studying the structure and lexical features of a malicious URL. They achieved an accuracy of 0.81 and an F1 score of 0.74. They proposed a classifier using SVM. Huang also showcases a similar work of detecting phishing URL by using SVM-Based technique.

M.S. Mayzner and M.E. Tresselt contributed to anagram research by providing a frequency table for letter combinations based on a sample of 20,000 English words. Peter Norvig enhanced their work by using a larger data set. We brought existing cryptanalysis techniques such as unigram frequencies, bigram frequencies and distribution of vowels in strings, into static analysis of strings found in Java programs. Metrics such as N-gram has been effectively applied in plagiarism detection. Sanaz Jabbari suggested a method to detect word obfuscation by analyzing the semantic relatedness of a word and its context. Honglak Lee proposed Hidden Markov Model for deobfuscating spam emails by analyzing probable misspellings, incorrect segmentations, and substitutions or insertions of non-alphabetic characters. The studies are done on textual data by evaluating its context and similarity. We do not examine the context, as it is not viable for Java programs.

3. Detection

This section introduces our analysis system, which is a combination of a static analyzer that extracts features indicating obfuscation, and a machine learning classifier that uses supervised learning to distinguish between obfuscated and non-obfuscated Java malware. The tool is designed to be generic and is agnostic to the obfuscation tool used to obfuscate malware. Fig. 3 shows the architecture of our obfuscation detection system.
3.1 Static Analyzer

The static analyzer tool operates at the level of bytecode to extract lexical information from it. Java bytecode is the instruction set for the Java Virtual Machine (JVM)\textsuperscript{31}. Java classfile contains the precisely compiled form of Java code. It is a stream of bytes (containing Java bytecode) that is executed on the JVM. Java malware is packaged as a Java Archive (Jar) containing classfiles. The input to the static analyzer is the malware Jar file. The contents of the Jar file are analyzed and the features extracted from the classes.

The symbol table information for each class is stored in a section of the classfile called the constant pool. The constant pool stores literal constant values such as numbers, strings, identifiers names, class names, method names etc. Since lexical obfuscation renames identifiers, the analyzer parses through the constant pool to extract strings, class names, field names and method names. They form the basis of our feature set.

3.2 Classifier

The classifier has two phases – a training phase (where statistical information about obfuscated and non-obfuscated malware is gathered) and the test/classification phase (where the classifier classifies new malware samples as obfuscated or non-obfuscated). The malware Jar samples are labeled as obfuscated or non-obfuscated to train the classifier. Research claims that SVM is the most appropriate classifier for text-based pattern recognition and yields the smallest error rates even with a limited training set\textsuperscript{20,38}. The training set data is labeled ‘True’ when the malware sample is obfuscated and ‘False’ to indicate non-obfuscated sample. We calculate the accuracy of results obtained from the classifier based on Precision, Recall and F-score\textsuperscript{18}.

Precision: Precision decreases with increase in false positives. It is computed as:

\[
\text{Precision} = \frac{\text{correctly labeled obfuscated samples}}{\text{total # of all samples labeled as obfuscated}}.
\]

Recall: Recall decreases with increase in false negatives. It is computed as:

\[
\text{Recall} = \frac{\text{correctly labeled obfuscated samples}}{\text{total # of all obfuscated samples}}.
\]

F-score: F-score indicates the overall classification accuracy. F-score varies from 0 to 1, with best value at 1.

\[
\text{F-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.
\]

3.3 Feature Extraction

Based on our study, we identified eleven text-based features to detect lexically obfuscated and string encrypted malware. The feature set is summarized in Table 1 and the rationale for the feature set is explained below.

Method, Class & Field Name Length: Lexical obfuscation uses shorter name patterns when a dictionary is not employed. The intuition behind this metric is that very short identifier names are uncommon in Java programs.

Word Count: In dictionary obfuscation, words of the attacker’s choice are applied to the identifiers. The metric word count measures the frequency of occurrence of words while renaming Java identifiers. It indicates the number of such repetitions of a word in a malware sample.

The remaining features identify string obfuscation. Bigrams are pairs of consecutive letters in strings. Common bigrams in strings are th, he, in, er e.t.c. Bigram frequency has been used as a stimulus measure in word perception experiments\textsuperscript{21}. 
**Frequency of Rare Bigrams:** Non-obfuscated strings seldom contain rare bigrams. A list of rarely occurring bigrams is taken from and other publically available corpora. Examples of rare bigrams are xx, yy, qq, xz, qt etc.

**Frequency Distribution and Vowel Count:** Vowels contribute to a major share of alphabets in a string. Prior research has computed the frequency distribution of vowels in the English language. Obfuscated strings show a drastic variation in the frequency of occurrence of vowels, as well as their total share among the whole characters that make up the strings in a classfile. It is proven that vowels constitute around one-third of the total number of characters in making meaningful sentences.

**Frequency of non-readable characters:** Obfuscated strings usually contain many unreadable characters. Such malformed strings occur as a result of applying shift operators on characters.

The basis for the following metrics is that obfuscated and encoded strings excessively contain special characters.

**N-gram:** N-gram is the frequency of occurrence of special characters of length equal to N. We set the value of N=1.

**Word Size:** Word size indicates the average length of words in a malware sample. This is matched with the average size of a word in the English language as the threshold.

**Entropy:** Entropy is a measure of distribution of bytes in the string and is computed as follows:

\[
E(B) = -\sum_{i=1}^{N} \left( \frac{B_i}{T} \right) \log\left( \frac{B_i}{T} \right)
\]  

where \( B_i \) is the count of each byte, while \( T \) is the total count of bytes within the string. The value \( E \) tends to 0, when string has less number of bytes. The max value of \( E \) is \( \log N \), meaning that the bytes are widely distributed around this set of bytes. \( N \) is set to 256 since we are considering all the ASCII characters. Entropy is found to be increasing with diverse characters within a string. The analysis tool calculates entropy of long strings. For example, in a string ‘aab’, the two alphabets ‘a’ and ‘b’ occur with frequencies 0.667 and 0.333 respectively. The entropy is calculated as 0.9183, approximately equal to 1, using the equation (1). This value indicates that each symbol is encoded by 1 bit and 3 bits are required to encode the string optimally.
Table 1. Features, description and thresholds

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Threshold for detection of malware (per single Jar file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name length</td>
<td>Number of characters in method name</td>
<td>length &lt; 3</td>
</tr>
<tr>
<td>Class name length</td>
<td>Number of characters in class name</td>
<td>length &lt; 3</td>
</tr>
<tr>
<td>Field name length</td>
<td>Number of characters in class name</td>
<td>length &lt; 3</td>
</tr>
<tr>
<td>Word count</td>
<td>Repetitions of a word</td>
<td>% of repeated words &gt; 33</td>
</tr>
<tr>
<td>Rare bigrams</td>
<td>Number of bigrams in a string that are</td>
<td>% of rare bigrams &gt; 1.5</td>
</tr>
<tr>
<td></td>
<td>not commonly found</td>
<td></td>
</tr>
<tr>
<td>Vowel frequency</td>
<td>The frequency of vowels in strings</td>
<td>If frequency is not in order e</td>
</tr>
<tr>
<td>Vowel count</td>
<td>Number of vowels in strings</td>
<td>% of vowels &lt; 25</td>
</tr>
<tr>
<td>Word size</td>
<td>The length of a string</td>
<td>length &gt; 9</td>
</tr>
<tr>
<td>N-gram</td>
<td>Occurrences of special characters within a string N=1</td>
<td>% of special characters &gt; 9</td>
</tr>
<tr>
<td>Non-readable</td>
<td>Number of strings containing non-</td>
<td>% words containing non-readable</td>
</tr>
<tr>
<td></td>
<td>readable characters</td>
<td>characters &gt; 5</td>
</tr>
<tr>
<td>Entropy</td>
<td>Distribution of bytes within a string</td>
<td>word length &gt; 8 and metric entropy &lt; 0.3</td>
</tr>
</tbody>
</table>

4. Experiment & Results

The static analyzer that contains the feature extractor is implemented using Java ASM 25, which is a bytecode manipulation framework in Java. A total of 182927 strings and 12721 class names were extracted from 375 Jar files of which 125 were obfuscated and 250, non-obfuscated. We used Weka 26 library for classification and tested four classifiers - Support Vector Machines (SVM), Naive Bayes, C4.5 Decision tree and Random forest. In our experiments, we used the 10-fold cross validation to train and evaluate the classifier results.

The test samples were categorized into two classes. The first class contained non-obfuscated samples and the second contained obfuscated samples. Moskovitch research 23 was used for estimating the ratio of obfuscated vs. non-obfuscated files in the training set. 33% total samples were obfuscated samples.

The threshold values are calculated based on previous research findings as well as our own. For example, the length of majority of words in English has been determined to fall in the range 4-7. For our experiment, the threshold was set to 9, since obfuscated and encoded strings tend to run long. The threshold value was arrived at by empirically measuring the longest word in non-obfuscated jars, which was obtained to be of length 20. The reason for such an unusually high value is the presence of URLs and encoded strings in non-obfuscated samples. However, a threshold value that is within the range 4-7 will not change our results in any way. The order of vowels in decreasing order of frequency is obtained from Dushatsky’s work 22. The unigram and bigram frequencies are assigned based on Solo’s research 29. On continuous observation, we found that lexical obfuscation that does not use a dictionary, can be detected by setting the threshold of the length of identifier names to 2. Since there are 26 * 26 possible combinations for a 2-letter name, we assume that a single package will never contain such a large number of classes. For the metric entropy, we arrived at a threshold value 0.3 after analyzing all the extracted strings. We evaluated the robustness of the features by calculating chi-squared statistic for each feature. The five most robust features are: vowel frequency, vowel count, method name length, word size and n-gram. Table 1 shows the summary of threshold values.
4.1. Results

Fig. 4 shows the percentage contributions of each of the ten features (excluding vowel frequency since vowel count includes the total frequency of each vowel) when 250 non-obfuscated malware and 125 obfuscated malware samples were statically analyzed.

For example, the plot of the metric word size shows that 15.43 % of the total words in non-obfuscated samples and 50.35 % in obfuscated samples have a word length greater than the threshold we set. URLs and encoded strings account for the longer word length in non-obfuscated samples. The graphs also show a higher percentage of identifiers (class, method and field names) renamed in obfuscated malware. The frequency of vowels does not follow the standard frequency distribution of English words for obfuscated samples. This indicates that strings are being transformed into meaningless words by some form of encryption technique. There are also more occurrences of rare bigrams, special characters and non-readable characters in obfuscated strings. An increase in the randomness
of characters in a string results in a corresponding increase in the entropy of the string. We calculated entropy for longer strings and divided the value by string length to obtain the metric entropy.

The evaluation of the performance of the four machine learning classifiers is shown in Table 2. Case I comprise of total number of obfuscated samples classified as non-obfuscated and case II comprises of total number of non-obfuscated samples classified as obfuscated. Research claims that SVM is the most appropriate classifier for text-based pattern recognition and yields the smallest error rates even with a limited training set. Unlike what we expected, the Random Forest algorithm returned the best overall results in classification. The classifier correctly classified 99% of all instances with an F-score of 0.99. While SVM did not classify even a single obfuscated malware as non-obfuscated, it classified a large number of non-obfuscated samples as obfuscated. C4.5 and Random forest algorithms performed better than SVM and produced lesser number of false negatives.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Precision I</th>
<th>Recall I</th>
<th>F-score I</th>
<th>Total # false Negatives</th>
<th>% of correctly classified instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>0.962</td>
<td>0.957</td>
<td>0.957</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.978</td>
<td>0.977</td>
<td>0.977</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>C4.5</td>
<td>0.984</td>
<td>0.983</td>
<td>0.983</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Random forest</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

5. Conclusion

Malware authors use obfuscation as a technique to conceal their code, and to evade static analysis tools. As obfuscation tools are also increasingly becoming sophisticated, it is crucial to be able to detect obfuscation prior to reverse engineering the malware. Our experiments prove that obfuscation in Java malware can be detected by extracting statically significant features. Automated detection is possible by using the features to train a machine learning classifier. The metrics used in detecting JavaScript obfuscation can be applied for Java malware by varying its threshold values. The newly proposed metrics – word count, identifier length, and frequency of rare bigrams and non-readable characters, have also been found to be effective when used along with the other existing metrics for obfuscation detection. A novel feature of our work was that we were able to bring techniques traditionally used for cryptanalysis such as frequency distribution of vowel count, frequency distribution of rare bigrams etc., to detect string obfuscation. The metric word count was also effective to detect dictionary obfuscation. Random forest classifier predicted output for test samples with an F-score of 0.99 by correctly classifying 99% of total instances.

References


15. Christian Collberg, Clark Thomborson and Douglas Low, ”Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, The University of Auckland


17. Christophe Chong [Stanford], Daniel Liu [Stanford], and Wonhong Lee [Neustar], ”Malicious URL Detection”

18. Peter Likarish, Eunjin (EJ) Jung and Insooon Jo, ”Obfuscated Malicious Javascript Detection using Classification Techniques”, Malicious and Unwanted Software (MALWARE), 2009 4th International Conference


21. Glenn A. Rice and David Owen Robinson, ”The role of bigram frequency in the perception of words and nonwords”, Memory & Cognition, 1975


23. Robert Moskovitch, Dima Stolpe, Clint Feher, Nir Nissim, Nathalie Japkowicz and Yuval Elovici, ”Unknown Malcode Detection and the Imbalance Problem”

24. Pedro Domingos, ”A Few Useful Things to Know about Machine Learning”, University of Washington

25. ASM Java bytecode manipulation and analysis framework, Available at http://asm.ow2.org/


27. Mayzner MS and Tresselt ME, ” Tables of single-letter and di-gram frequency counts for various word-length and letter-position combinations”, Psychonomic Monograph Supplements, 1965


30. V.V. Bochkarev, A.V. Shevlyakova and V.D. Soloyev, ”Average word length dynamics as indicator of cultural changes in society”, Kazan Federal University, Kazan, Russia


33. Sanaz Jabbari, Ben Allison and Louise Guthrie, ”Using A Probabilistic Model Of Context To Detect Word Obfuscation”, University of Sheffield

34. Honglak Lee and Andrew Y. Ng, ”Spam Deobfuscation using a Hidden Markov Model”, Stanford University


37. ProGuard Java code obfuscation and optimization tool available at http://sourceforge.net/projects/proguard/?source=directory