# Android Malware Classification through Analysis of String Literals

**Richard Killam, Paul Cook, Natalia Stakhanova**
Faculty of Computer Science
University of New Brunswick
richard.killam@unb.ca, paul.cook@unb.ca, natalia.stakhanova@unb.ca

**Abstract**

As the popularity of the Android platform grows, the number of malicious apps targeting this platform grows along with it. Accordingly, as the number of malicious apps increases, so too does the need for an automated system which can effectively detect and classify these apps and their families. This paper presents a new system for classifying malware by leveraging the text strings present in an app's binary files. This approach was tested using over 5,000 apps from 14 different malware families and was able to classify samples with over 99% accuracy while maintaining a false positive rate of 2.0%.

**Keywords:** Mobile Malware, Android, String analysis

## 1. Introduction

The rising popularity of the Android platform has led to an increase in observed malware, as much as 391% from 2013 to 2014 (Pulse Secure Mobile Threat Center, 2015). This unprecedented increase is often associated with the inherit openness of the Android platform, and the corresponding lack of security measures to prevent potential abuse.

Consequently, a significant amount of research attention has focused on exploring the nature of malware applications and developing techniques for their detection. A wide range of malware detection approaches have been introduced, from generic methods such as RiskRanker (Grace et al., 2012a), DroidRanger (Zhou et al., 2012b), and Drebin (Arp et al., 2014), to more specialized methods offering detection of repackaging (Crussell et al., 2013; Gonzalez et al., 2015; Zhou et al., 2013) and privacy violations (Enck et al., 2010; Zhang et al., 2013). Keeping up with the growing prevalence of Android malware, the existing studies focus on either a complex multi-feature analysis capable of providing insight into a malware sample (e.g., DroidSafe (Gordon et al., 2015)), or a large-scale binary classification identifying malicious and benign samples during the triage stage.

With the increasing amount and complexity of malware, in triaging it is beneficial to employ methods that do not rely on sophisticated analyses, and are capable of providing insight into the malicious functionality represented by a malware app.

Since core functionality is shared by samples of the same malware family, identification of the malware family that a sample belongs to is the first step in understanding and assessing malware impact and potential damage.

In this work, we propose a machine-learning based approach for the detection of Android malware through analysis of unreferenced, hidden strings present in Android executables. In general, strings have multiple uses in Android binaries, including referencing code components based on class or methods' names. Prior research has focused mostly on analysis of the code portions of Android executables (i.e., .dex files) referenced through corresponding strings. Since this is typical and therefore expected code invocation method, it is commonly employed in reverse engineering of Android applications (commonly referred to as apps). Leveraging this, malware authors often avoid detection by doing indirect code invocation by placing code in unreferenced strings. In this work, we propose to explore these components. Specifically, we investigate the role of strings found within an Android executable, but not referenced by the code.

We evaluate the proposed method with a dataset made available through The Android Malware Genome project (Zhou and Jiang, 2012), and a collection of over 4,000 benign apps retrieved from Google Play market. Our findings using this novel approach based on unreferenced strings show that this is an efficient alternative to prior approaches to malware classification. Our best approach achieves 99.3% accuracy, and a false positive rate of just 0.5%, for malware detection, and 99.2% accuracy with a false positive rate of 2.0% for malware family classification.

We further contrast our approach against the one using all strings. Our experiments show that using only a small set of unreferenced string features, we are able to effectively identify a suspicious app maintaining the same high accuracy as with all strings.

The rest of this paper is organized as follows: we present related work in Section 2. Our system design is described in Section 3. A detailed description of the features used for classification are outlined in Section 4. We describe our experimental setup in Section 5. We present results in Section 6. Finally, we conclude our work in Section 7.

## 2. Related Work

In recent years the area of mobile security has seen extensive growth and improvement. A broad overview of characteristics of mobile malware, and approaches to its detection, has been given by Zhou et al. (2012), Polla et al. (2013), and Alzahrani et al. (2014).

The resource-constrained environment of mobile platforms presents significant challenges to the detection of malware. As a result, features that detection techniques rely on play a critical role for the accuracy of malware detection. The features commonly used in existing classification studies can be grouped into two categories: dynamic features derived from an app's behaviour during runtime, and static features, extracted from an app itself.

Systems that use dynamic analysis, such as RiskRanker (Grace et al., 2012b), focus on an app's behaviour while running, typically monitoring system calls made by the app. CopperDroid (Tam et al., 2015) and DroidScope (Yan and Yin, 2012) are examples of techniques that utilize dynamic features at a host level (e.g., system calls). Arora et al. (2014) focused on network-level features extracted from app behaviour.

Techniques that employ static analysis target the files that are packaged with an application. This eliminates the need to execute the app in order to detect any malicious intent. The majority of systems that utilize static analysis have focused on two types of packaged files: AndroidManifest.xml — which holds the app's metadata — and executable files. App permissions contained in AndroidManifest.xml have been explored by several studies, including DroidRanger (Zhou et al., 2012b), Drebin (Arp et al., 2014), and Auditor (Talha et al., 2015).

Static features extracted from executables often require additional preprocessing, and are commonly used in studies in the form of opcode or bytecode $n$-grams. For instance, Wolfe et al. (2014) analyze the bytecode $n$-grams extracted from Android binaries. Juxtapp (Hanna et al., 2013) evaluates code similarity based on opcode $n$-grams extracted from selected packages of the disassembled .dex file. Similarly, DroidMOSS (Zhou et al., 2012a) evaluates app similarity using fuzzy hashes constructed based on a series of opcodes. DroidKin (Gonzalez et al., 2014) detects unique apps by analyzing the similarity of opcode $n$-grams and metadata of apps.

In contrast to these previous studies, we propose a novel form of static analysis based specifically on features derived from the unreferenced strings contained within an executable.

## 3. Android String Analysis

An Android application package (APK) file, is a compressed folder that contains a variety of files including an executable .dex file; a manifest file (AndroidManifest.xml) that describes the content of the package; and resources files (e.g., image and sound files). The Dalvik executable file, or simply .dex file, is a binary that results from compiling the app's Java source code. As illustrated by Figure 1, this file is partitioned into a number of sections that describe aspects of the structure of the file.

Among these sections are several identifier lists that contain offsets pointing to the corresponding entries in the data section. As such, the string identifiers list (i.e., the string_ids section) provides offsets to all strings used by the .dex file. These strings are used for internal naming — e.g., class, field, or local variable names — and to reference constant objects specified in the source code (e.g., string literals). The other identifier sections — type_ids, proto_ids, field_ids, method_ids, and class_defs — may also contain references to the string identifiers list. For example, a class named `myclass` will have a corresponding entry in the string identifier section pointing to the actual string *myclass*.

This structure is defined in a formal specification of the layout of .dex executable files guiding the development of

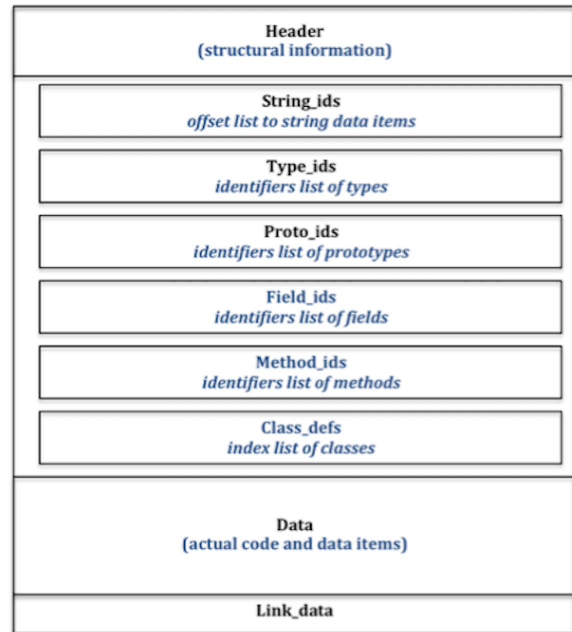


Figure 1: The structure of a DEX file.

Android apps.[1] Although app development must follow the suggested guidelines, there are a number of techniques that enable parts of code to be hidden in an Android executable (Apvrille, 2012; Apvrille, 2013). The main premise of these techniques is to place the code inside the data section while avoiding it being referenced by class and method index lists. Since these lists are typically used to invoke the methods and classes, this prevents reverse engineering of an app and allows malware apps to bypass anti-malware scans.

The majority of prior research has focused on the code section of the officially documented .dex structure. In contrast, we explore the text components. As such we differentiate between strings that are pointed to by any of the other identifier sections, denoted as "referenced strings", and all other strings referred to as "unreferenced strings". Referenced strings can be viewed as a part of functional app code. They are linked to other identifier lists and thus, for example, can represent classes, methods, or data structures. On the other hand, the data section also contains "unreferenced" strings that are only referenced by the string offset list, and thus carry textual information, e.g., string literals. In prior work on malware analysis this part of the code has not carried the same weight as the functional portion. However, these unreferenced strings often carry hidden or interesting information. For example, hardcoded URLs and email addresses (represented as strings) are common in malware apps, and would occur amongst the unreferenced strings. As such, analysis of unreferenced strings can potentially indicate malicious activity embedded in Android apps.

In this paper we investigate the impact of unreferenced strings found within the Android executable on the tasks of identifying malicious Android apps, and classifying Android apps with respect to malware family.

[1] `https://source.android.com/devices/tech/dalvik/dex-format.html`. Accessed: 2016-02-11

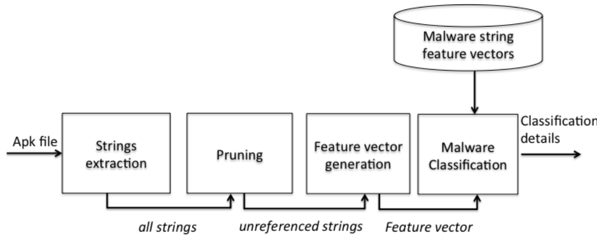


Figure 2: The sequence of analysis in the proposed approach.

### 3.1. String analysis

To investigate the role of unreferenced strings in malware analysis, the proposed system encompasses several steps: string extraction, pruning, feature vector generation, and finally assessment, as shown in Figure 2. The initial steps of string extraction and pruning primarily focus on preprocessing activities that are necessary to allow our system to carry out feature vector generation. These vectors then serve as the basis for the following classification and analysis.

In the *strings extraction* step, for each app under analysis, our system recursively unpacks and extracts all .dex files found in both the root APK, as well as within all sub-APKs. All of these .dex files contribute to the final string data associated with the root app. The strings of each .dex file are extracted by traversing the string_ids list and collecting all strings present in data section.

The *pruning* step then prunes these strings to only include the unreferenced strings. This is accomplished by traversing the 5 other identifier lists — type, proto, field, method, and class — and then removing strings that are referenced by these lists. The resulting unreferenced strings therefore consist solely of non-executable code, such as string literals and local variable names, as well as strings from hidden executable parts of code not referenced through identifier lists. These remaining strings then form the basis for feature vectors.

To generate feature vectors in the *feature vector generation* step, we consider word-level string $n$-gram features (described in the following section) derived from the unreferenced strings. As a point of comparison, we also consider feature vectors generated from all strings (i.e., referenced and unreferenced strings). To form these feature vectors, we simply omit the pruning step.

In contrast to our proposed approach, malware analysis is typically conducted at the bytecode or opcode level. Opcodes are generally beneficial in representing the low-level semantics of the code, while bytecode is seen as the complete representation of the code at a low-level. As a further important point of comparison, we therefore also experiment with opcode and bytecode $n$-grams (also described in the Section 4.). For these opcode and bytecode features, the strings extraction and pruning phases do not apply.

Feature vectors are generated for each app in a large collection of Android apps, each of which is known to be either a malicious app, or a benign (non-malware) app. In the case of malware, the malware family is also known. These feature vectors are used to train supervised classifiers to identify malicious Android apps, and classify Android apps according to their malware family. After training, in the *malware classification* step the malware status and family of a new app is assessed based on its classification under the previously-trained classifier model.

## 4. String features

To preserve the semantics of the original strings, the string features used in the proposed approach were extracted at the word level. They were then processed into $n$-grams of varying lengths. An $n$-gram is a contiguous sequence of $n$ items. In the case of word level $n$-grams these items are words in a sentence. For example, given the sentence *Good morning John Doe.*, three 2-gram tokens can be made:

1. *Good morning*
2. *morning John*
3. *John Doe.*

Note that the tokenization used here was to split a string based on whitespace; therefore in this case the period at the end of the sentence is a part of the word *Doe.*

In the feature vector generation process, each extracted string was written onto a separate line, so that these word level $n$-grams could be line bounded. This was done to ensure that $n$-grams did not contain words from different lines. Since the strings are held in lexicographical order, adjacent strings are not necessarily related. As such, $n$-grams spanning multiple lines would contain word sequences that are not associated, that is, words that are not a part of the same original string. These $n$-grams would contain meaningless relationships, and would thus add noise to the feature set, which could hinder classification.

### 4.1. String Feature Extraction

Each app was represented using both token ($n$-gram) frequencies and token frequency-inverse document frequency (tf-idf) weights. Frequency vectors contain the number of times a term (i.e., an $n$-gram) occurs in a given document (where in our case a "document" is the collection of strings extracted from a given app); tf-idf assigns lower weights to terms that occur in many documents, and is calculated as follows:

$$\text{tf-idf}_{i,d} = f_{i,d} * \log\left(\frac{N}{n_i}\right) \quad (1)$$

where $f_{i,d}$ is the frequency of term $i$ in document $d$, $N$ is the total number of documents, and $n_i$ is the number of documents that term $i$ occurs in at least once.

These tokens were extracted by grouping the strings on each line into word level $n$-grams, with $n$ ranging from 1 to 4. Any lines that had fewer than $n$ words were ignored for that feature set. The analysis of string sizes across our dataset showed that only a few strings had more than 4 words (Figure 3). As such in this work we capped $n$-gram length at 4.

If $n > 1$ then a unique line boundary token was added to the start and end of each line. This ensured that all lines

Table 1: Examples of $n$-gram features. <LB> is the line boundary token.

| Line Text | 1-grams | 2-grams | 3-grams | 4-grams |
|---|---|---|---|---|
| mThread= | mThread= | <LB> mThread=<br>mThread= <LB> | <LB> mThread= <LB> | |
| wrote final 256; | wrote<br>final<br>256; | <LB> wrote<br>wrote final<br>final 256;<br>256; <LB> | <LB> wrote final<br>wrote final 256;<br>final 256; <LB> | <LB> wrote final 256;<br>wrote final 256; <LB> |

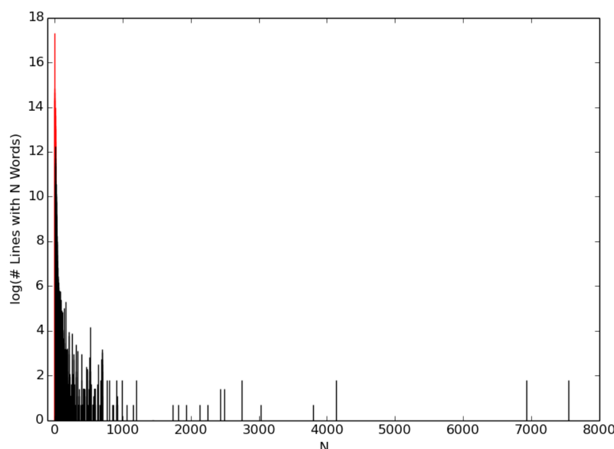


Figure 3: The distribution of string lengths in the dataset.

contained at least 3 words (including the line boundary tokens), thereby reducing the number of lines that were ignored for having fewer than $n$ words, and therefore making more information available in the feature vectors. Adding line boundary tokens also serves to identify the words and $n$-grams that occur at the beginning and end of strings. This could be potentially important information in that a string that begins with, for example, *warning* or *error* could mean something very different than a string that simply contains these words.

Line boundary tokens were not added to the 1-gram features. The frequency of these special line boundary tokens in this case would be equivalent to counting the number of strings in the app. As this information is not directly encoded by the other $n$-gram features, it was also avoided here.

An example of the proposed feature extraction method is shown in Table 1. Note the following important properties:

- There are no <LB> tokens for the 1-gram feature sets.
- There were no 4-gram tokens for the first string, "mThread=". This is because, even with the line boundary tokens, the string only contained 3 words. As such, it was ignored for not having at least $n = 4$ words in it.
- The words in the 2 and 3-gram tokens were in the same order in which they appeared in the original string, and the words were always adjacent. That is, the 2-gram feature set did not contain tokens such as "final wrote" or "wrote 256;".

Table 2: Distribution of Android APK files across malware families

| | Malware Family | File Count |
|---|---|---|
| 1 | ADRD | 22 |
| 2 | AnserverBot | 187 |
| 3 | BaseBridge | 122 |
| 4 | DroidDreamLight | 46 |
| 5 | DroidKungFu1 | 34 |
| 6 | DroidKungFu2 | 30 |
| 7 | DroidKungFu3 | 309 |
| 8 | DroidKungFu4 | 96 |
| 9 | Geinimi | 69 |
| 10 | GoldDream | 47 |
| 11 | KMin | 52 |
| 12 | Pjapps | 58 |
| 13 | YZHC | 22 |
| 14 | Other | 166 |
| 15 | Benign GooglePlay apps | 4,574 |
| | Total | 5,834 |

# 5. Experimental setup

## 5.1. Data

To evaluate the performance of the proposed malware analysis approach we employed a standard benchmark dataset produced by The Android Malware Genome project (Zhou and Jiang, 2012). This dataset consists of 1,260 malware apps. We also required a collection of benign (i.e., non-malware) apps for evaluation purposes. We performed a large-scale study of the top Android applications retrieved from Google Play market between September 2014 and January 2015. These apps were inspected by ESET antivirus scanner to detect the presence of malware. All malicious apps were removed from the dataset. This resulted in 4,574 benign (i.e., non-malware) GooglePlay apps which were used in this study.

The apps retrieved from Google play market and The Android Malware Genome project were used for classification experiments. This dataset of 5,834 apps is detailed in Table 2.

Some families in this dataset ended up with only a few samples, making classification and validation of results difficult. As such we grouped low frequency families into an "Other" class. The app distribution among the families had a convenient gap between families with 16 and 22 samples. As such, all families that had fewer than 17 files were considered to be low frequency, and placed in the 'Other' class. Thereby resulting in 15 distinct malware families, including the benign GooglePlay apps.

### 5.2. Feature extraction

Opcodes for each DEX file were extracted using dexdump, a disassembler tool that is a part of the Android SDK. Opcode operands were discarded as their variability typically results in noise and excessively sparse feature vectors which makes classification more difficult.

Each app was represented using opcode $n$-grams, where $n$ ranged from 1 to 4, the same range of $n$ considered for string $n$-grams. Opcode $n$-grams were DEX file bounded (as opposed to line bounded in the case of string $n$-grams). A special DEX file boundary token was included for the opcode $n$-grams for $n > 1$, for similar reasons as the inclusion of the line boundary token in the case of string $n$-grams.

The bytecode features were obtained by reading each byte of each DEX file. Since DEX files start with the magic number "dex",[2] thus acting as built-in boundary tokens, additional boundary tokens were not used. Each app was represented using bytecode $n$-grams, where $n$ ranged from 1 to 2.

### 5.3. Classification and Baseline

We implemented the proposed approach in Python, using the scikit-learn library (Pedregosa et al., 2011). In preliminary experiments we considered a variety of classifiers including $k$-nearest neighbors, multinomial naive Bayes, logistic regression, and a linear support vector machine. The linear SVM (Fan et al., 2008) had the best performance, or close to it, in terms of both accuracy and false positive rate on malware detection and family classification. We therefore focus on, and report results for, the linear SVM classifier for the remainder of the paper.

In the case of the datasets used in this study, and in most application marketplaces, benign applications outnumber malicious applications. In a case such as this where the classes (benign and malicious) are imbalanced, a very naive strategy of simply selecting the most frequent class can perform very well, particularly in terms of accuracy. We therefore also report a majority class baseline — referred to as "Baseline" — that classifies all samples as the most common class in the training data. In our experiments this strategy will classify all samples as benign. This baseline provides a reference against which we can interpret the results of other methods. Crucially, it is not the only approach against which we compare our string-based classification approaches; we also consider approaches based on opcodes.

## 6. Experimental Results

In our experimental study we focused on analysis of:

1. Binary classification: classification of a sample as either a malicious or benign app. This type of classification is also referred to as malware detection.

2. Malware family classification: where the samples were classified as one of 15 different classes, 14 of which are malware families and 1 class representing benign apps. These families, and the number of files in them, are listed in Table 2.

3. Unreferenced strings vs all strings: The classification performance using unreferenced strings was contrasted against that using all strings (i.e., referenced and unreferenced).

4. Classification with selected string features: classification of apps based on selected, highly-informative features.

All classification experiments were repeated 5 times, each using stratified 5-fold cross validation.

### 6.1. Unreferenced strings vs all strings

We compared the performance of the linear SVM when using all of the strings versus using only the unreferenced strings. These results are shown in Table 3. The results are, overall, quite similar. For example, the best accuracy for malware detection ("Binary") is 99.5% for all strings, and 99.3% for unreferenced strings. For malware family classification there is a similarly small difference in terms of the best accuracy for the all strings and unreferenced strings approaches. That the all strings approach does slightly better does not come as a surprise; retaining all strings provides more information to the classifier. However, with unreferenced strings, we are able to drastically cut the total number of strings that are examined (from roughly 127 million to 46 million) while maintaining a similar accuracy.

The performance results of the linear SVM classifier using only the unreferenced strings for malware detection and malware family classification are shown in Table 4. Unsurprisingly, all $n$-gram classification strategies performed significantly better than the baseline in all cases, with the exception of the false positive rate for malware detection (Binary).

For both malware detection and family classification, we see an increase in performance for all metrics when increasing the size of the word grams from 1 to 2, and again when increasing from 2 to 3. This is likely caused by the increased contextual information that is carried by higher order $n$-grams. However, there is a clear drop in performance for 4-grams. Two factors could contribute to this. Data sparsity could be an important factor, because many 4-grams will be unique. Furthermore, the system ignores strings that contain fewer than $n$ words after padding with line boundary tokens. The 4-gram strategy ignored over 30M lines throughout the dataset. One possibility to overcome this would be to pad lines with additional word boundary tokens; however, this would substantially increase the number of features in this model, making classification much more computationally intensive.

The best strategy for both malware detection and family classification was to use 3-gram word counts. This strategy was able to achieve near perfect ($> 99\%$) accuracy for both malware detection and malware family classification while maintaining a low false positive rate. Moreover, the accuracy, precision, and recall of this method is better than that for any of the classifiers using the more-conventional bytecode or opcode $n$-gram features.

[2] `https://source.android.com/devices/tech/dalvik/dex-format.html` Accessed: 2016-02-11

Table 3: Accuracy (Acc), macro-averaged precision (Prec) and recall (Rec), and false positive rate (FPR) for malware detection (Binary) and malware family classification (Family) using a linear SVM with features based on all strings (All) and just the unreferenced strings (Unref).

| | Strategy | Acc | | Prec | | Rec | | FPR | |
|---|---|---|---|---|---|---|---|---|---|
| | | All | Unref | All | Unref | All | Unref | All | Unref |
| Binary | 1-gram word freq. | 99.3% | 98.8% | 97.8% | 96.5% | 99.0% | 98.1% | 0.63% | 0.99% |
| | 2-gram word freq. | 99.5% | 99.2% | 98.6% | 98.0% | 99.1% | 98.5% | 0.40% | 0.56% |
| | 3-gram word freq. | 98.6% | 99.3% | 96.8% | 98.1% | 96.6% | 98.6% | 0.87% | 0.50% |
| | 4-gram word freq. | 98.5% | 98.6% | 96.7% | 96.9% | 96.5% | 96.7% | 0.89% | 0.85% |
| | 1-gram word tf-idf | 99.4% | 99.3% | 99.3% | 99.4% | 98.0% | 97.2% | 0.18% | 2.8% |
| | 2-gram word tf-idf | 99.3% | 99.2% | 99.3% | 99.3% | 97.3% | 97.2% | 0.19% | 2.8% |
| | 3-gram word tf-idf | 98.8% | 99.2% | 99.2% | 99.3% | 95.0% | 96.9% | 0.21% | 3.1% |
| | 4-gram word tf-idf | 98.6% | 98.6% | 99.1% | 99.1% | 94.5% | 94.5% | 0.25% | 0.24% |
| Family | 1-gram word freq. | 99.0% | 98.5% | 97.4% | 96.7% | 97.2% | 96.5% | 88.6% | 2.7% |
| | 2-gram word freq. | 99.5% | 99.0% | 98.4% | 98.3% | 98.4% | 97.5% | 94.9% | 1.6% |
| | 3-gram word freq. | 98.2% | 99.2% | 95.9% | 98.6% | 95.2% | 98.0% | 4.8% | 2.0% |
| | 4-gram word freq. | 98.2% | 98.4% | 94.9% | 96.2% | 94.9% | 95.5% | 5.1% | 4.5% |
| | 1-gram word tf-idf | 98.5% | 96.6% | 98.6% | 98.7% | 93.3% | 94.4% | 6.7% | 5.6% |
| | 2-gram word tf-idf | 98.3% | 98.7% | 98.0% | 98.6% | 92.4% | 94.9% | 7.6% | 5.1% |
| | 3-gram word tf-idf | 97.5% | 98.4% | 96.0% | 97.9% | 88.6% | 93.2% | 11.4% | 6.8% |
| | 4-gram word tf-idf | 97.1% | 97.1% | 93.5% | 93.5% | 85.0% | 85.0% | 15.0% | 15.0% |

Table 4: Accuracy (Acc), macro-averaged precision (Prec) and recall (Rec), and false positive rate (FPR) for malware detection (Binary) and malware family classification (Family) using a linear SVM with features based unreferenced strings, bytecodes, and opcodes.

| Strategy | Binary | | | | Family | | | |
|---|---|---|---|---|---|---|---|---|
| | Acc | Prec | Rec | FPR | Acc | Prec | Rec | FPR |
| Baseline | 78.4% | | | 0.00% | 78.4% | | | 93.3% |
| 1-gram word freq. | 98.8% | 96.5% | 98.1% | 0.99% | 98.5% | 96.7% | 96.5% | 3.5% |
| 2-gram word freq. | 99.2% | 98.0% | 98.5% | 0.56% | 99.0% | 98.3% | 97.5% | 2.5% |
| 3-gram word freq. | 99.3% | 98.2% | 98.6% | 0.50% | 99.2% | 96.2% | 95.5% | 2.0% |
| 4-gram word freq. | 98.6% | 96.9% | 96.7% | 0.85% | 98.4% | 98.6% | 98.0% | 4.5% |
| 1-gram word tf-idf | 99.3% | 99.4% | 97.2% | 2.8% | 96.6% | 98.7% | 94.4% | 5.6% |
| 2-gram word tf-idf | 99.2% | 99.3% | 97.2% | 2.8% | 98.7% | 98.6% | 94.9% | 5.1% |
| 3-gram word tf-idf | 99.2% | 99.3% | 96.9% | 3.1% | 98.4% | 97.9% | 93.2% | 6.8% |
| 4-gram word tf-idf | 98.6% | 99.1% | 94.5% | 0.24% | 97.1% | 93.5% | 85.0% | 14.99% |
| 1-gram bytecode freq. | 85.6% | 70.1% | 70.7% | 10.29% | 74.8% | 24.7% | 21.4% | 78.58% |
| 2-gram bytecode freq. | 92.3% | 83.5% | 83.0% | 5.09% | 83.0% | 39.7% | 29.7% | 70.26% |
| 1-gram bytecode tf-idf | 79.0% | 93.0% | 3.14% | 0.06% | 78.4% | 7.0% | 6.8% | 93.23% |
| 2-gram bytecode tf-idf | 94.1% | 86.8% | 86.0% | 3.62% | 82.2% | 19.2% | 14.5% | 85.53% |
| 1-gram opcode freq. | 97.9% | 95.0% | 95.3% | 1.38% | 95.9% | 84.7% | 85.3% | 14.71% |
| 2-gram opcode freq. | 98.1% | 95.0% | 96.4% | 1.42% | 96.7% | 90.1% | 87.9% | 12.07% |
| 3-gram opcode freq. | 98.2% | 95.0% | 96.8% | 1.39% | 97.4% | 94.0% | 91.5% | 8.52% |
| 1-gram opcode tf-idf | 98.5% | 96.0% | 97.3% | 1.12% | 95.7% | 92.4% | 77.0% | 22.99% |
| 2-gram opcode tf-idf | 99.3% | 98.1% | 98.7% | 0.52% | 97.9% | 95.2% | 89.7% | 10.32% |
| 3-gram opcode tf-idf | 99.2% | 98.0% | 98.4% | 0.56% | 97.9% | 95.2% | 90.7% | 9.28% |

## 6.2. Feature Selection

Since anti-malware vendors are forced to maintain large and ever-growing numbers of signatures for malware detection, it is important that any malware detection and classification methods be able to use a minimal number of features. As such, we tested the proposed method using only the 50 highest ranked 3-gram tokens, extracted from the unreferenced strings, for each family.

These 3-grams were ranked by training a linear SVM using the one-vs-rest classification strategy. Features that have high coefficients have more of an impact on how the linear SVM classifies a given sample. Therefore, the 50 tokens that had the highest coefficient for each family, are equivalent to the 50 highest ranked tokens for each of those families. We present examples of the top-5 ranked 3-grams for selected families in Table 5. By using only these 50 strings per family, we were able to reduce the number of features to 728.[3] These features suggest that unreferenced strings

[3] Because some tokens were indicative of multiple families, the number is slightly less than the number of families $*$ 50.

Table 5: The 5 most important 3-grams when used to classify selected malware families. Note that <LB> is the line boundary token and that the whitespace between the words was removed to reduce memory consumption when training.

| Class | Rank | Token |
|---|---|---|
| Benign apps | 1 | <LB> this <LB> |
| | 2 | <LB> accessFlags <LB> |
| | 3 | <LB> android.intent.action.VIEW <LB> |
| | 4 | <LB> string <LB> |
| | 5 | <LB> com.android.vending <LB> |
| DroidKungFu2 | 1 | <LB> /system/etc/.rild_cfg <LB> |
| | 2 | <LB> /system/etc/.dhcpcd <LB> |
| | 3 | <LB> /WebView.db <LB> |
| | 4 | <LB> WebView.db.init <LB> |
| | 5 | <LB> /secbino <LB> |
| DroidKungFu3 | 1 | <LB> /system/etc/.rild_cfg <LB> |
| | 2 | <LB> /system/etc/.dhcpcd <LB> |
| | 3 | <LB> -1 <LB> |
| | 4 | <LB> sysName <LB> |
| | 5 | <LB> 4/system/bin/chmod <LB> |
| GoldDream | 1 | <LB> Content-Disposition:form-data; <LB> |
| | 2 | <LB> SmsDataType <LB> |
| | 3 | <LB> zjphonecall.txt <LB> |
| | 4 | <LB> lebar.gicp.net <LB> |
| | 5 | <LB> ws_v $< LB >$ |

that contain filenames, URLs, and source code are highly informative as to malware families.

Further manual analysis of selected string $n$-grams also confirmed our initial hypothesis that unreferenced strings present a narrow more focused view of malware behavior and bear interesting information. For example, GoldDream Trojan app uploaded stolen information to a remote server. The URL of this server 'lebar.gicp.net' became visible only through analysis of unreferenced strings (see Table 5). Our analysis also revealed that strings often carry code snippets that contain additional functionality. For example, javascript might be embedded as a string to be executed during the runtime of an app. The use of databases and consequently the presence of sql queries was also prevalent among malware apps.

In preliminary experiments considering this feature selection strategy, the accuracy obtained by using only these 728 features was 96.2%, which is only 3 percentage points lower than using all of the unreferenced strings. However, the false positive rate went up to 13.9%, an increase of 11.9 percentage points. Nevertheless, this feature selection strategy managed to reduce the length of the feature vectors from over 8 million to just 728, while still maintaining relatively high accuracy.

### 6.3. Discussion

The proposed method of classifying Android malware is rather promising due to several reasons. As opposed to the majority of existing malware classification techniques that rely on a large set of features, the proposed approach requires only 50 features per malware family. This is especially appealing to anti-malware vendors that are constantly forced to keep up with an ever-growing number of signatures for malware detection.

The proposed method exhibits very good filtering capability overall. Our results are especially attractive since it seems to be possible to extract a small set of string features that can effectively identify a suspicious app and further characterize the majority of malware samples within a family. This approach can complement the existing techniques and significantly simplify the triaging stage in automated analysis tools.

## 7. Conclusion

The amount of malware targeting the Android mobile platform is on the rise, as such, the need for an effective automated system to detect and classify malware and the different malware families is crucial. This study presented such a solution, through the use of text strings extracted from the DEX files of an android application. The proposed system of using a linear SVM with line-bounded word-level 3-grams was able to classify malware families with an accuracy of 99.2% while maintaining a false positive rate of just 2.0%.

There are multiple opportunities for future work. First, we will test the feature selection strategy on a dataset of obfuscated apps, as well as attempt to improve the string extraction process by using various natural language processing techniques such as stemming and case folding. Finally, we plan to use the datasets from this study to develop an ensemble classification system which utilizes the string, bytecode, and opcode features.

## 8. Bibliographical References


Alzahrani, A. J., Stakhanova, N., Gonzalez, H., and Ghorbani, A. (2014). Characterizing evaluation practices of intrusion detection methods for smartphones. *Journal of Cyber Security and Mobility*, 3:89–132.

Apvrille, A. (2012). Guns and smoke to defeat mobile malware. In *Hashdays Conference*.

Apvrille, A. (2013). Playing hide and seek with dalvik executables. In *Hacktivity*.

Arora, A., Garg, S., and Peddoju, S. (2014). Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual NDSS*.

Crussell, J., Gibler, C., and Chen, H. (2013). Scalable semantics-based detection of similar android applications. In *18th ESORICS*, Egham, U.K.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on OSDI'10*, pages 1–6, Berkeley, CA, USA. USENIX Association.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Gonzalez, H., Stakhanova, N., and Ghorbani, A. (2014). Droidkin: Lightweight detection of android apps similarity. In *Proceedings of International Conference on Security and Privacy in Communication Networks (SecureComm 2014)*.

Gonzalez, H., A.Kadir, A., Alzahrani, A., Stakhanova, N., and Ghorbani, A. A. (2015). Exploring reverse engineering symptoms in android apps. In *European Workshop on Systems Security (EuroSec)*. IEEE.

Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. (2015). Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*.

Grace, M. C., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012a). Riskranker: scalable and accurate zero-day android malware detection. In *The 10th MobiSys*, pages 281–294.

Grace, M. C., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012b). Riskranker: scalable and accurate zero-day android malware detection. In *The 10th MobiSys*, pages 281–294.

Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. (2013). Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th DIMVA'12*, pages 62–81, Berlin, Heidelberg. Springer-Verlag.

La Polla, M., Martinelli, F., and Sgandurra, D. (2013). A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pulse Secure Mobile Threat Center. (2015). Mobile threat report. Technical report, Pulse Secure Mobile Threat Center.

Talha, K. A., Alper, D. I., and Aydin, C. (2015). {APK} auditor: Permission-based android malware detection system. *Digital Investigation*, 13:1 – 14.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*.

Wolfe, B., Elish, K., and Yao, D. (2014). High precision screening for android malware with dimensionality reduction. In *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*, pages 21–28.

Yan, L. K. and Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA. USENIX Association.

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC*, CCS '13, pages 611–622, New York, NY, USA. ACM.

Zhou, Y. and Jiang, X. (2012). Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE.

Zhou, W., Zhou, Y., Jiang, X., and Ning, P. (2012a). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM CODASPY '12*, pages 317–326, New York, NY, USA. ACM.

Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. (2012b). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual NDSS*.

Zhou, W., Zhou, Y., Grace, M., Jiang, X., and Zou, S. (2013). Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the CODASPY '13*, pages 185–196, New York, NY, USA. ACM.