Who Changed You? Obfuscator Identification for Android

Yan Wang and Atanas Rountev
The Ohio State University, Columbus, OH, USA
Email: {wang10,rountev}@cse.ohio-state.edu

***Abstract*—Android developers commonly use *app obfuscation* to secure their apps and intellectual property. Although obfuscation provides protection, it presents an obstacle for a number of legitimate program analyses such as detection of app cloning and repackaging, malware detection, identification of third-party libraries, provenance analysis for digital forensics, and reverse engineering for test generation and performance analysis. If the obfuscator used to create an app can be identified, and if some details of the obfuscation process can be inferred, subsequent analyses can exploit this knowledge. Thus, it is desirable to be able to automatically analyze a given app and determine (1) whether it was obfuscated, (2) which obfuscator was used, and (3) how the obfuscator was configured.**

We have developed novel techniques to identify the obfuscator of an Android app for several widely-used obfuscation tools and for a number of their configuration options. We define the obfuscator identification problem and propose a solution based on machine learning. To the best of our knowledge, this is the first work to formulate and solve this problem. We identify a feature vector that represents the characteristics of the obfuscated code. We then implement a tool that extracts this feature vector from Dalvik bytecode and uses it to identify the obfuscator provenance information. We evaluate the proposed approach on real-world Android apps obfuscated with different obfuscators, under several configurations. Our experiments indicate that the approach identifies the obfuscator with about 97% accuracy and recognizes the configuration with more than 90% accuracy.

***Keywords*-Android, App analysis, Obfuscation**

I. Introduction

The explosive growth in the use of mobile devices such as smartphones and tablets has led to substantial changes in the computing industry. Android is one of the major platform for such devices [1]. Because of commercial interests and security concerns, many companies and individual developers are reluctant to make public the Android app source code. But even without source code, these apps can be easily decompiled and pirated. Developers have to use additional protections to secure their apps and intellectual property.

One widely used approach for app protection is *obfuscation*. Obfuscation hides informative data in the software and makes it hard to understand for both humans and decompilation tools. Obfuscation is easy to use: a number of obfuscators have been developed and their deployment is fairly simple. For example, the Android IDE provided for free by Google integrates the ProGuard [2] obfuscation tool. Due to this ease of use, obfuscation is becoming more popular. For example, according to a study published in 2014 [3], about 15% of the apps in the Google Play app store are obfuscated, for both free and paid apps. In our studies we observed a percentage that is substantially higher.

Although obfuscation provides desirable protection, it presents an obstacle for legitimate program analyses. Consider the *detection of app cloning and repackaging*. At least 25% of apps in Google Play are clones of other apps [3] and almost 80% of the top 50 free apps have fake versions because of repackaging [4]. Obfuscation may impair clone and repackage detection tools because an obfuscated clone app may appear to be different from the original app [5], [6]. Another consideration is *malware detection* using static analysis. Analysis of obfuscated malicious code presents a number of challenges [7], [8]. As yet another example, identifying the *third-party libraries* included in an Android app is a well-known problem important for general static analysis, security analysis, testing, and detection of performance bugs. Obfuscation hinders such library identification [9], [10], [11].

In all these examples, one possible approach is to develop *obfuscator-tailored* techniques. If the specific obfuscator used to create the final app code can be identified, and if some details of the obfuscation process can be inferred, the subsequent analyses can exploit this knowledge. However, a key prerequisite is to be able to analyze a given app and determine (1) whether it was obfuscated, (2) which obfuscator was used, and (3) how the obfuscator was configured.

These questions are also important for *provenance analysis*. The app is the result of a process that starts with the source code and produces the final distributed APK file. The provenance details of this process and the toolchain that implements it are important for a number of reasons. For example, such details are useful for digital forensics [12] and for reverse engineering of models used for test generation (e.g., [13], [14]) and code instrumentation for performance analysis (e.g., [15]). Knowledge of the obfuscator and the configuration options used by it reveal aspects of the app toolchain provenance, similarly to provenance analysis for binary code [12].

We have developed novel techniques to identify the obfuscator of an Android app for several widely-used obfuscation tools and for a number of their configuration options. The approach relies only on the Dalvik bytecode in the app APK. The identification problem is formulated as a machine learning task, based on a model of various properties of the bytecode (e.g., different categories of strings). This model reflects characteristics of the code that may be modified by the obfuscators—in particular, names of program entities as well

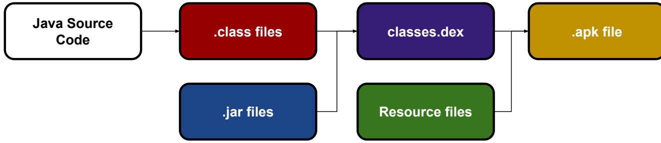


Fig. 1: Workflow of Android compilation.

```
1.new-instance v4,Landroid/content/ComponentName; // type@0033
2.move-result-object v1
3.if-nez v1,000d // +0004
4.const/4 v2,#int 0 // #0
5.return-object v2
6.invoke-static {v3},Landroid/support/v4/content/IntentCompat;
.makeMainActivity:(Landroid/content/ComponentName;)
Landroid/content/Intent; // method@0e03
7.invoke-direct {v4},Landroid/content/Intent;.<init>:()V
// method@013a
```

Fig. 2: Example of Dalvik bytecode.

as sequences of instructions. The main insight of our approach is that relatively simple code features are sufficient to infer precise provenance information. The specific contributions of this work are:

- We define the obfuscator identification problem for Android and propose a solution based on machine learning techniques. To the best of our knowledge, this is the first work to formulate and solve this problem.
- We identify a feature vector that represents the characteristics of the obfuscated code. We then implement a tool that extracts this feature vector from Dalvik bytecode and uses it to identify the obfuscator provenance information.
- We evaluate the proposed approach on Android apps obfuscated with different obfuscators, under several configurations. Our experiments indicate that the approach identifies the obfuscator with 97% accuracy and recognizes the configuration with more than 90% accuracy.

II. Overview

This section presents an overview of Android obfuscation and a high-level description of our analysis approach.

A. Software Obfuscation

Software obfuscation is a well-known technique for protecting software from reverse engineering attacks [16]. Obfuscation transforms the input code to generate new target code. At a high level, we have $T = f(I)$ where I is the input code that needs obfuscation; it can be source code, bytecode, or machine code. T is the target code and f is an obfuscating transformation. I and T must have the same observable behavior: for the same valid input, both should terminate and generate the same output. Obfuscation cannot achieve a "virtual black box" [17]—in other words, perfect obfuscation is impossible. Nevertheless, in practice it is still an effective approach to protect the code from humans and from code analysis and reverse engineering tools [18].

B. Android Obfuscation

Obfuscation for Android is both easy to use (e.g., using the ProGuard tool available in Android Studio) and observed for

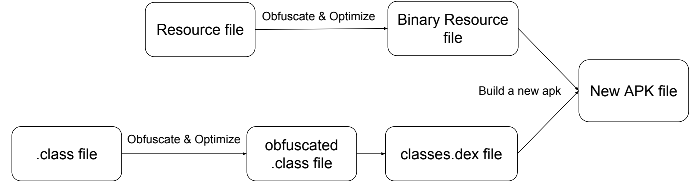


Fig. 3: Workflow of an obfuscator.

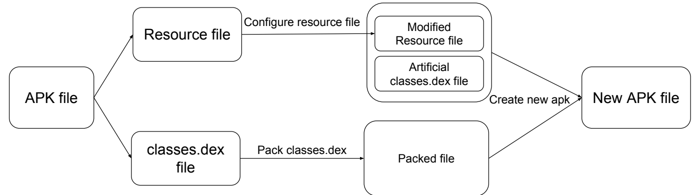


Fig. 4: Workflow of a packer.

many applications in the Google Play app store [3]. Android apps are released as APK ("Application PacKage") files. The process of obtaining an APK file is shown in Figure 1. The developer builds the app using Java. The Java source code is compiled into *.class* files using some Java compiler. The *.class* files contain standard Java bytecode. Android uses its own format of bytecode called Dalvik bytecode. Figure 2 is a snippet of such bytecode. The Java bytecode is compiled into a file *classes.dex* ("Dalvik Executable"), which contains all the Dalvik bytecode together with other *.jar* files that may come from third-party libraries. In most cases, there is only one *classes.dex* file, although a large app can be split into multiple dex files. In addition to the dex file(s), the APK also contains resource files (e.g., images). At installation time, the Android runtime will compile *classes.dex* using the *dex2oat* tool. The output will be executable code for target device, which achieves better performance than the Dalvik bytecode.

In Android, obfuscation happens before releasing the app. Without obfuscation, the Dalvik bytecode is relatively easy to reverse engineer, using tools such as Soot [19], Androguard [20], Baksmali [21], Apktool [22], Dex2jar [23], Dexdump [24], etc. If the app is obfuscated, even just using ProGuard (which only provides basic obfuscation), the changes will affect these tools and hinder reverse engineering [25].

Although obfuscation may protect intellectual property, it may also affect other legitimate uses of reverse engineering and app analysis. As discussed earlier, one such example is the detection of cloning and repackaging. Several studies have shown the negative influence of obfuscation on such detection [5], [6]. Obfuscation is also an obstacle for some anti-malware tools [7], [8] because the pattern matching used by these tools requires static analysis that is not resilient to obfuscation changes. Obfuscation also causes problems for the identification of third-party libraries because it may change the content of library classes [9], [10], [11].

```
1  public static void initSetting(Editor editor){
2    editor.remove("deviceSalt");
3    editor.remove("encryptedKeyPass");
4    editor.commit();
5  }
6  private void goNextStage(){
7    gotoStage(this.stage + 1);
8  }
```

(a) Original classes.dex.

```
1  public static void a(Editor arg0){
2    arg0.remove(R.a("o\u0018}\u0014h\u0018X\u001cg\t"));
3    arg0.remove(android.support.v7.appcompat.R.a(
                                       "u,s0i2d't\tu;@#c1"));
4    arg0.commit();
5  }
6  private void a(){
7    a(this.l + 1);
8  }
```

(b) Obfuscated classes.dex.

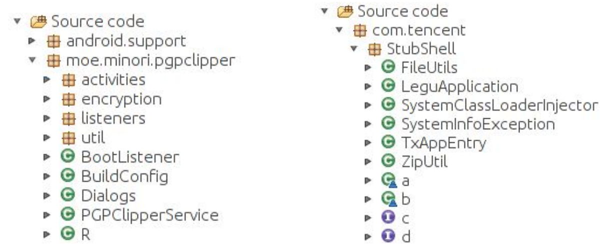
Fig. 5: Example of obfuscation performed by Allatori.

C. Two Categories of Obfuscators

Various obfuscators are available to Android developers. These tools can be broadly classified in two categories. An obfuscator in the first category performs code transformations. In most cases it will transform Java bytecode, and has almost the same behavior as a Java obfuscator except for considerations related to Android features. Examples of tools from this category are ProGuard, Allatori, DashO, DexGuard, and Shield4J. Figure 3 illustrates the workflow of this type of obfuscator. One typical technique is *name obfuscation*, which replaces the names of packages, classes, methods, and fields with meaningless sequences of characters. Sometimes the package structure is also modified, which further obscures the names of packages and classes. Other techniques include *flow obfuscation*, which modifies code order or the control-flow graph, and *string encryption*, which encrypts the constant strings in the code. Some tools may go further and obfuscate the XML files in the resource part of the APK.

Example: Figure 5b illustrates obfuscation with Allatori, one of the tools used in our experiments. Figure 5a shows the original code. For illustration purposes, the figures show the equivalent Java source code rather than the actual bytecode. Names are obfuscated and the constant strings are encrypted. For example, methods `initSetting` and `goNextStage` are renamed to `a`. The encrypted strings are decrypted using methods `R.a` and `android.support.v7.appcompat.R.a`. These methods are created by the obfuscator in the corresponding `R` resource classes. The tool uses multiple decryption methods because it employs several different encryption options and seems to arbitrarily pick one to encode a string.

The second category of tools contains *packers*. A packer encrypts the original *classes.dex* file, then decrypts this file in memory at run time and executes it via reflection using `DexClassLoader`. Tools such as APKProtect, Bangcle, and Legu belong to this category. They will not perform any bytecode modifications—rather, they hide the entire dex file. Figure 4 shows the corresponding process. The original



(a) Original class structure. (b) Legu class structure.

Fig. 6: Example of obfuscation performed by Legu.

classes.dex will be packed and an artificial *classes.dex* file will be generated. The new file loads the original file in memory. The packed *classes.dex* is encrypted. Figure 6b shows the package and class structure after obfuscation with Legu. The original structure is shown in Figure 6a. New helper classes are created and the original classes are hidden.

A tool may provide choices to its users. For example, both ProGuard and Allatori have a configuration to specify whether to perform different levels of obfuscation, and/or to provide a user-defined renaming style to replace the default one.

D. Machine Learning Model

The goal of our work is to identify the specific obfuscator that was used to create a given APK file. This information may enable the creation of obfuscation-tailored analysis, testing, and instrumentation, as well as provenance analysis for digital forensics. While the internal details of various obfuscation tools are not public, these tools are developed by unrelated development teams and are likely to differ substantially in low-level details. Our premise is that these differences manifest themselves in the obfuscated code. Thus, we aim to select suitable code features and to apply machine learning techniques based on them. In particular, we use labeled data to train a *classifier*. The classifier will map a vector of code features to a label representing one of several known obfuscation tools. Details of this approach are presented in Section III.

E. Obfuscators Used in Our Study

There are many obfuscators and most of them are commercial. The cost of these tools is relatively high. As a result, in this study we selected free tools or free versions of paid tools. We used five different tools. ProGuard is provided by Google, Allatori is developed by a Russian company, and DashO is a product from an U.S. company. Legu and Bangcle are packers from China. ProGuard is integrated with Android Studio, the official IDE from Google. Allatori is a commercial tool, but it has a free educational version. Amazon, Fujitsu, Motorola, and hundreds of other companies worldwide have used Allatori to protect their software products from being reverse-engineered by business rivals [26]. The company developing DashO [27] has over $5,000$ corporate clients in over 100 countries. Legu is developed by Tencent, one of the largest IT companies in China. Currently there are more than 5 million developers using this company's platform and apps released by them

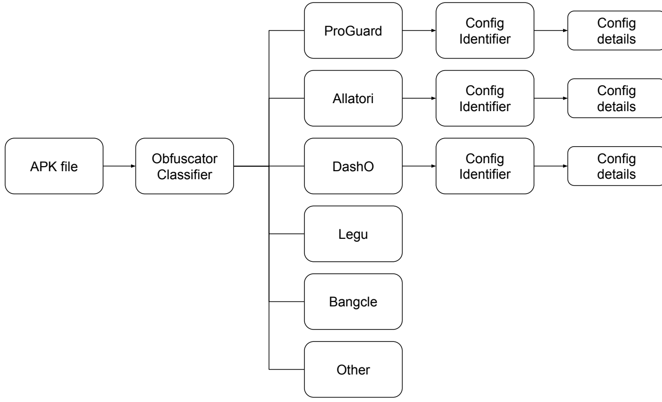


Fig. 7: Workflow for obfuscator identification.

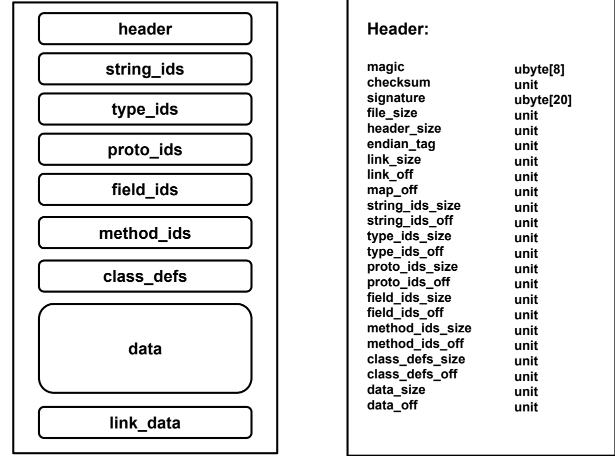
will be protected by Legu [28]. Bangcle is developed by a company with the same name that provides security services to individuals, enterprises, and governments. The tool has served seventy thousand companies and has protected more than seven hundred thousand apps, which are installed on about seven hundred million devices [29].

III. Machine Learning for Obfuscator Identification

We use supervised learning in which a training set is used to create several classifiers. Later, a given APK (with unknown obfuscator provenance) is classified using these classifiers. The processing of an unknown APK is shown in Figure 7. In the first stage, the APK is classified in one of six categories based on obfuscator type. For the three obfuscators that allow customized configurations (ProGuard, Allatori, and DashO), a second stage gathers information about the configuration under which the APK was obfuscated. This section describes how a classifier is generated for the first stage. The next section describes additional classifiers used in the second stage.

A. Training Set

For training, we obtained open-source apps from the F-Droid repository [30] and attempted to build them with Android Studio, which uses the Gradle build tool. A total of 282 apps were successfully built with Gradle. These apps were then obfuscated by us using various obfuscators and configurations. The source code was required because some obfuscators work on *.class* files. For each app, we created 6 *baseline APKs*: (1) without obfuscation, (2) obfuscated with ProGuard's default configuration, (3) obfuscated with Allatori's default configuration, (4) obfuscated with DashO's default configuration, (5) obfuscated with Legu, and (6) obfuscated with Bangcle. Since ProgGuard, Allatori, and DashO allow customization, we also created *customized APKs*. Using different configurations of ProGuard, 6 customized APKs were obtained per app. Similarly, 3 customized APKs per app were obtained using configurations of Allatori, and another 3 customized APKs were built using DashO. Details of the configurations used for training are presented in the next section.



(a) Dex file structure. (b) Header structure.

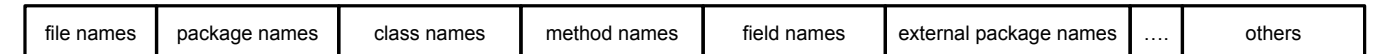


Fig. 8: Structure of feature dictionary.

Each APK is labeled with a label from set $Labels = \{ProGuard, Allatori, DashO, Legu, Bangcle, Other\}$ where the last label was used for the unobfuscated APKs.[1] The features of each APK (described below) together with its label are used as input to the training phase.

B. Feature Dictionary

Features are obtained from the bytecode in *classes.dex*. The structure of this file is shown in Figure 8a. Many parts of the file are irrelevant for our purposes. For example, the structure of the header is shown Figure 8b; it contains the magic number, checksum number, file size, etc. This information varies across apps regardless of obfuscation.

In our approach we focus on the data section. We choose the *strings* in this section as candidates for features. These strings have different roles. For example, some are names of program entities such as classes and methods. For the code shown in Figure 5b, the set of class names is {`"Editor"`} and the set of method names is {`"a"`, `"remove"`, `"commit"`}. These distinctions between strings are informative.

The feature dictionary is a collection of 10 sets of strings, as shown in Figure 8. All strings except file names are from the data section in the dex file. Four sets correspond to names of packages, classes, methods, and fields, respectively. These are program entities that are defined by the APK code. We also consider four corresponding sets, but for external entities defined by code outside of the dex file (e.g., by the Android framework). The reason for this separation is that an obfuscator may treat internal and external names differently. Finally, any other string occurring in the code is considered to be in the set of "other" strings.

[1]If the classifier produces *Other* (Figure 7), this means that the unknown APK was either unobfuscated or was obfuscated by an unknown tool.

For the code from Figure 5b, this "other" set is {"this", "arg0", "o\u0018}\u0014h\u0018X\u001cg\t", "u,s0i2d't\tu;@#c1"}.

We developed a dex parser to collect this information from a given APK, based on the mapping relationship between ids and the strings in the dex file. The location and size of the ids can be obtained from the header. The file names are obtained from the structure of the APK archive, which can be examined using the standard `aapt` tool ("Android Asset Packaging Tool", used for Zip-compatible archives such as APK files). This information may be useful when the obfuscator adds its own helper library files.

For each obfuscated APK A_i from the training set, 10 sets $S_{i,j}$ of strings are extracted ($1 \leq j \leq 10$) based on the categories described above (illustrated in Figure 8). The following processing of these sets is performed to obtain the final feature dictionary. First, if several APKs $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ are obtained from the same original app a using the same obfuscator l, their corresponding string sets are merged using set intersection. The result is a collection of 10 sets $S_j^{a,l}$ for $1 \leq j \leq 10$, where $S_j^{a,l} = \cap_{p=1}^{k} S_{i_p,j}$. The intent is to use strings that are configuration-independent. After this step, for each pair (a, l) where a is an app and l is an obfuscator, there is a collection of 10 string sets $S_j^{a,l}$.

The next step is to trim these string sets as follows. For each obfuscator l and each string category j, we consider the union of all $S_j^{a,l}$. The resulting set F_j^l can be thought of as a feature dictionary specific to this obfuscator l and this string category j (e.g., class names). Each such F_j^l is trimmed by removing rarely-occurring strings. Specifically, if the percentage of sets $S_j^{a,l}$ containing a string s is less than a particular threshold, s is removed from F_j^l. Our experiments indicated that a threshold of 15% produces substantial reduction in the size of the feature vector without significant reduction in prediction accuracy.

To further reduce dimensionality, features that are covered by other ones will be ignored. Specifically, consider two string $s, s' \in F_j^l$. If the set of apps $\{a \,|\, s \in S_j^{a,l}\}$ is a proper subset of $\{a \,|\, s' \in S_j^{a,l}\}$, then s is less informative than s' and will be removed from F_j^l.

After these trimming techniques are applied, the feature dictionaries for all obfuscators are reduced from thousands of elements to hundreds of elements. Trimming also reduces the training time for the classifier by about a factor of two.

The final feature dictionary is a collection of 10 string sets $\hat{F}_j$ for $1 \leq j \leq 10$. Each set is computed as

$$\hat{F}_j = \bigcup_l F_j^l - \bigcap_l F_j^l$$

The second term removes strings that always appear regardless of the obfuscator l, since such strings are not useful for distinguishing among obfuscators.

The feature vector used for training and subsequent classification has 10 sub-vectors, each corresponding to one set $\hat{F}_j$. If a string $s \in \hat{F}_j$ occurs in category j inside an APK, the corresponding element of the j-th sub-vector of the feature vector is 1; otherwise the element is 0.

C. Training a Classifier

After constructing the feature vector for each APK in the training set, we build a corresponding classifier. In our scenario, we have a multi-class classification problem— that is, APKs are classified into one of several (more than two) classes. The classes are defined by set $Labels$ described earlier. We do not consider the case when more than one obfuscator is used for the same APK, because typically in practice developers will adopt one obfuscator tool. Therefore, a *one-vs-rest* classifier is appropriate for our purposes. This strategy trains a single classifier per class. Each class is fitted against all other classes. The concrete type of classifier we use is linear Support Vector Machine (SVM). A SVM finds a weight vector w that defines a decision boundary in the feature space which best separates two different classes. The distance from a particular example to that boundary is the *margin* and is defined as $w^T x$, where x is the feature vector. In such a binary classifier, each instance is assigned to class $+1$ or -1 depending on the sign of the margin.

For one-vs-rest classification, a binary classifier can be extended to n classes. This standard approach determines n weight vectors $\{w_1, \ldots, w_n\}$ by partitioning the data into two groups, one for the current class and the other for everything else. Given these vectors and a new instance with feature vector x, we choose the class that maximizes the margin:

$$\mathrm{argmax}_{1 \leq k \leq n} \, w_k^T x$$

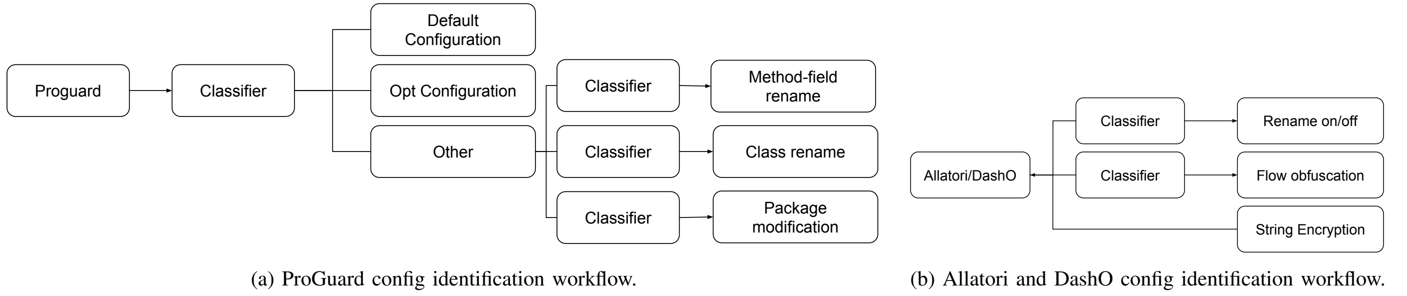
In addition to linear SVM, we also experimented with other kinds of classifiers; details are presented in the evaluation section. Our experience is that linear SVM provides a good balance between accuracy and running time.

IV. Obfuscator Configuration

After obtaining the type of obfuscator, we determine characteristics of the obfuscator configuration, if applicable. Because the configuration may significantly change the behavior of an obfuscator, this knowledge may be useful for APK analysis and reverse engineering. In our case, ProGuard, Allatori, and DashO provide customization: the obfuscator user can define an XML file to modify the default tool behavior.

A. ProGuard Configuration

The configuration analysis for ProGuard is shown in Figure 9a. In the first step, three types of configurations are differentiated. The *default* one and the *default-opt* one are provided directly in Android Studio. The difference is that the opt configuration performs aggressive optimizations of the APK to reduce code size and to improve run-time performance. The third option *other* means the developer customized the configuration. There are four kinds of possible modifications: (1) customizing field and method names, (2) changing class names, (3) package flattening, which moves all packages into a single user-specified parent package, and (4) repackaging classes, which moves all classes into a single user-specified package. The last two transformations will change the structure of the package hierarchy and we treat them as one category in



(a) ProGuard config identification workflow.

(b) Allatori and DashO config identification workflow.

Fig. 9: Workflow for configuration characterization.

the classification process. All modifications depend on user-defined values. For example, the tool user can provide a list of arbitrary strings as candidates for class names, or can specify a custom package name.

During training we can only observe specific instances of these user-defined names. The resulting classifiers can subsequently determine, for a given (never-seen-before) ProgGuard-obfuscated APK, which settings have been customized. As shown in Figure 9a, the approach first identifies whether the configuration is one of the two default ones or is user-customized. In the third case, we determine which specific settings have been changed.

The training process for both stages is similar to what was described in the previous section, but with different feature vectors. For each app a, we constructed 6 APKs using ProgGuard: default, default-opt, customized field/method names, customized class names, package flattening, and repackaging. The labels used to label these APKs are from set $\{default$, $default\text{-}opt$, $field\&method$, $class$, $package\}$. The last label is used for both package-related transformations. In our training set, APKs with the last 3 labels are constructed with arbitrary new names defined by us. The resulting classifiers are used to process future APKs in which the user-specified names are different from the ones used during training.

We consider a feature dictionary which is a combination of the features of APKs with labels $default$ and $default\text{-}opt$. The remaining APKs are not used because they are dependent on customizable user-defined names. For defining the first classifier in Figure 9a, the APK labels are mapped to the simplified label set $\{default$, $default\text{-}opt$, $other\}$. For the remaining three classifiers shown in Figure 9a, the APK labels are mapped to the two labels $true$ and $false$, based on whether the particular setting is the default one or not. As before, linear SVMs are used for the classifiers.

B. Allatori/DashO Configuration

The analyses of Allatori and DashO configurations are shown in Figure 9b. The two tools are very similar in terms of configuration options and are analyzed using the same approach; below we describe only Allatori processing.

We identified three categories of customization settings. The *rename* switch controls whether the tool will obfuscate the names of packages, classes, etc. This setting is different from

Algorithm 1: FeatureDictionaryForAllatoriOrDashO

Input: $APKs = \{apk_i\}$: APKs obfuscated by Allatori/DashO with and without flow obfuscation
Input: N

```
1 feature_InstSequences ← ∅
2 foreach apk ∈ APKs do
3     foreach class ∈ apk do
4         foreach method ∈ class do
5             feature_InstSequences ← feature_InstSequences ∪
              all instruction sequences of length N
```

the one used in ProGuard as it does not require used-defined names. If turned on, the identifiers will be obfuscated using logic internal to the tool and the package hierarchy will also be changed. The process of identifying this property is the same as with ProGuard. *String encryption* is a setting to determine whether the constant strings in the code are encrypted. If turned on, the tool will identify all string data and encode it as illustrated in the earlier example from Figure 5b. The tool also adds code to decode the strings at run time. Our learning cannot handle this case well, because the approach does not model the “naturalness” of constant strings. One simple approach to detect this setting could be to examine the code for calls to tool-specific string decoding methods.

Another customization category is related to *flow obfuscation*. If set, the tool will change control-flow features of loops and branches. Here we need a different feature dictionary, constructed as described in Algorithm 1. The process is similar to considering N-grams in natural language processing.

For each obfuscated APK we compute the set of consecutive instruction sequences with length N. Since the obfuscation may affect the names of instruction operands, only the operator is considered. The sequences are obtained using the Androguard tool. Consider an app a, its baseline obfuscated APK, and its 3 APKs obtained from customized configurations. Each APK defines a set of sequences. We consider the union of these 4 sets and remove from it the intersection of the 4 sets. The result is a set S^a of configuration-dependent sequences. Next, rarely-occuring sequences are filtered. If, over the entire set of apps a, the percentage of sets S^a containing a sequence s is less than the threshold from Section III-A, s is removed from all S^a. The union of the final S^a is the feature dictionary.

V. Evaluation

We evaluated our approach on a number of Android apps. The evaluation considers the following questions: (1) Do the selected features represent the characteristics of these four obfuscators so that the classifiers can predict the correct label with high accuracy? (2) Are linear SVMs suitable in terms of accuracy and time cost, compared to other possible classifiers?

A. Data Set

We obtained 282 apps from F-Droid [30] and created several APKs for each app, as described in Section III-A. In total, we successfully generated about 2600 APK files, because in some cases the obfuscator(s) failed.

B. Methodology

The data set has to be split into training set and testing set. The training set is used to build the parameters of the model, while the testing set is chosen for evaluating the performance. Using standard 10-fold cross validation, we randomly divides the apps into 10 equally-sized subsets $Apps_i$ for $1 \leq i \leq 10$. The experiment is executed 10 times, once for each i. In each run, set $Apps_i$ is used as the testing set and the remaining apps are used as the training set. The predicted labels are then compared with the actual ones. The machine learning framework used in our evaluation is *scikit-learn* [31]. This toolkit provides various classifiers, including linear SVMs.

C. Obfuscator Identification

Table I shows the accuracy and F_1 score measurements obtained using 10-fold cross validation. The table shows the mean values and standard deviations of the 10 runs. The accuracy, for one run, is the ratio of number of testing APKs with correctly-predicted labels to the total number of testing APKs. For multi-class classification, two types of F_1 scores are typically computed: *micro* and *macro* [32]. The micro F_1 score considers the total number of true positives, false negatives, and false positives. The macro metric computes a value for each label and uses the unweighted mean. The corresponding equations are shown below.

$$precision = \frac{TP}{TP + FP} \quad recall = \frac{TP}{TP + FN}$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

$$F_{1macro} = \frac{\sum_{i=1}^{N} F_{1i}}{N}$$

$$precision_{micro} = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N} TP_i + \sum_{i=1}^{N} FP_i}$$

$$recall_{micro} = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N} TP_i + \sum_{i=1}^{N} FN_i}$$

$$F_{1micro} = 2 * \frac{precision_{micro} * recall_{micro}}{precision_{micro} + recall_{micro}}$$

where TP is the number of true positives, FP is the number of false positives, FN is the number of false negatives, and N is the number of labels. The results indicate that our approach has high accuracy and F_1 score.

	Accuracy	F_1 micro	F_1 macro
mean	0.975	0.975	0.968
SD	0.007	0.007	0.009

TABLE I: Accuracy and F_1 score for obfuscator identification.

Precision						
	ProGuard	Allatori	DashO	Legu	Bangcle	Other
mean	0.977	0.979	0.993	1.000	1.000	1.000
SD	0.005	0.003	0.008	0.000	0.000	0.000
Recall						
	ProGuard	Allatori	DashO	Legu	Bangcle	Other
mean	0.976	0.977	0.996	1.000	1.000	1.000
SD	0.005	0.004	0,007	0.000	0.000	0.000

TABLE II: Precision and recall for obfuscator identification.

To evaluate the performance for each individual obfuscator, we also collected data about precision and recall for each label. The results are shown in Table II and can be summarized as follows. Apps that are not obfuscated are detected correctly. Legu and Bangcle seem to have unique characteristics and our approach identifies them correctly. For ProGuard, Allatori, and DashO the model achieves very high accuracy. Since these tools have some similar obfuscation strategies, in rare cases the classifier may be unable to distinguish them correctly. Nevertheless, the overall measurements indicate that our model has a very high chance to correctly identify the type of obfuscator, for the five types covered by our implementation.

D. ProGuard Configuration Identification

If the obfuscator is identified as ProGuard, the next step is to characterize its configuration as illustrated in Figure 9a. In the first stage, we determine how the configuration is mapped to a label from $\{default, default\text{-}opt, other\}$. For "other", a second stage determines a $true/false$ label in each of the following 3 categories: field&method, class, and package. Here $true$ means that the category was customized by the ProGuard user. The accuracy and F_1 scores for both stages are shown in Table III. Since there are three labels in the first stage, we consider both micro and macro F_1 score. The labels in the second stage are binary and there is only one F_1 score. The results shown in the table indicate that our approach characterizes the ProGuard configuration with high accuracy.

The precision and recall of each stage are shown in Table IV. The recall of *default-opt* is relatively low. By comparing the ProGuard XML configuration files of *default* and *default-opt*, we observed that the difference between them is not very significant. Configuration *default-opt* will perform further performance optimizations and some related small code changes. In some scenarios, the optimization may not be available or it cannot cause an apparent difference. This may be the reason why for some apps this configuration is not identified. In addition, some apps are very simple and have few classes. For correctness, ProGuard will preserve some classes. For example, the app entry class defined in the *manifest.xml* file will not be obfuscated. As another example, subclasses of

Precision						
	ProGuard stage I			ProGuard stage II		
	default	default-opt	other	field&method	class	package
mean	0.984	0.939	0.936	0.995	0.998	0.905
SD	0.048	0.069	0.028	0.014	0.015	0.032
Recall						
	ProGuard stage I			ProGuard stage II		
	default	default-opt	other	field&method	class	package
mean	0.940	0.783	0.979	0.942	0.935	0.923
SD	0.073	0.088	0.016	0.056	0.064	0.040

TABLE IV: Precision and recall for ProGuard configuration.

ProGuard stage I						
	Acc.		F_1 micro		F_1 macro	
mean	0.938		0.938		0.917	
SD	0.022		0.022		0.031	
ProGuard stage II						
	field&method		class		package	
	Acc.	F_1	Acc.	F_1	Acc.	F_1
mean	0.988	0.963	0.989	0.967	0.940	0.909
SD	0.012	0.039	0.006	0.020	0.030	0.045

TABLE III: Accuracy and F_1 for ProGuard configuration.

	field&method	class	package
mean	0.957	0.960	0.945
SD	0.006	0.002	0.005

TABLE V: Accuracy with never-seen custom names.

Allatori				
	Rename		Flow	
	Accuracy	F1	Accuracy	F1
mean	0.994	0.987	0.933	0.931
SD	0.007	0.014	0.003	0.033
DashO				
	Rename		Flow	
	Accuracy	F1	Accuracy	F1
mean	0.981	0.981	0.979	0.984
SD	0.014	0.013	0.015	0.015

TABLE VI: Accuracy and F_1 for Allatori/DashO config.

Allatori				
	Precision		Recall	
	Renam	Flow	Rename	Flow
mean	0.993	0.936	0.989	0.929
SD	0.015	0.053	0.017	0.035
DashO				
	Precision		Recall	
	Rename	Flow	Rename	Flow
mean	0.977	0.974	0.986	0.976
SD	0.023	0.034	0.021	0.024

TABLE VII: Precision and recall for Allatori/DashO config.

framework class `android.view.View` require some class members to be unchanged. If the app is simple, the differences between *default* and *default-opt* may be insignificant and unrecognizable, which may cause some misclassification.

ProGuard allows the user to specify custom strings for class names, field names, method names, or package names. In the training process, we use a specific set of random strings (defined by us) for these values. To ensure that the classification can be successfully applied to APKs in which different user-defined obfuscated names were used, we performed the following experiment. We created another set of random strings, different from the string set used for training. During each run of the cross validation, the model is also tested against APKs obfuscated with this second set of string (i.e., with strings that have not been observed during training). The mean and standard deviation of the accuracy are shown in Table V. The accuracy is high for all three categories of names, indicating that our model is able to detect the change of these configuration settings regardless of the specific string values being used.

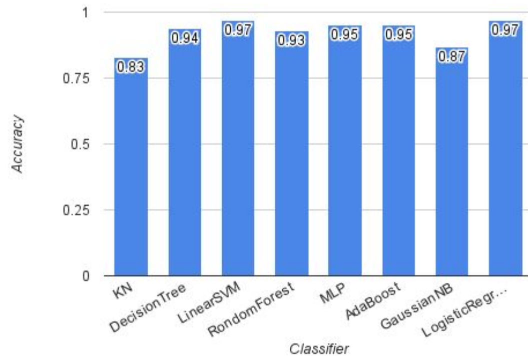
E. Allatori and DashO Configurations

The configuration characterization for Allatori and DashO has only one stage. As mentioned earlier, we do not consider string encryption, but focus on renaming and control-flow modifications. For both of these categories, a *true*/*false* label is determined to indicate that the configuration setting has been changed by the user. To compute the feature vector for the control-flow configuration, we define a feature dictionary containing all instruction subsequences of length 2, 3, and 4. The accuracy and F_1 scores are shown in Table VI. The results indicate that with high confidence we can identify whether these two Allatori and DashO configuration options have been modified.

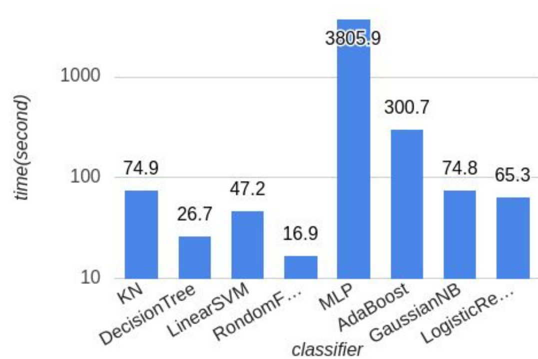
The precision and recall are shown in Table VII. The high values indicate that model typically correctly identifies the Allatori and DashO configuration settings. Misclassifications have the same root causes as with ProGuard. For example, for correctness, the tool has to preserve some elements of the original code. As another example, the tool will change the order of instructions inside a loop; if there are no loops in the code, no control-flow changes will be observed.

F. Comparison of Classifiers

Linear SVMs are only one of many available approaches that can be used for classification. To compare different classifiers, we considered 8 popular options: k-nearest neighbors, decision trees, linear SVMs, random forests, multi-layer perception, AdaBoost, Gaussian naive Bayes, and logistic regression. All are available as part of the *scikit-learn* machine learning toolkit used in our experiments. The experimental machine uses Ubuntu 16.04 with Intel Core i7-4770 CPU 3.40GHz and 16GB RAM. The accuracy for different clas-



(a) Accuracy of classifiers.



(b) Running time of classifiers.

Fig. 10: Performance of different classifiers.

sifiers is shown in Figure 10a. The y-axis is the mean of accuracy for each stage including both obfuscator and configuration identification. The x-axis is the type of classifier. The results indicate that the proposed feature vector works well for most classifiers. Among them, linear SVMs and logistic regression have the best performance. Figure 10b illustrates the total running time of 10-fold cross validation for both obfuscator and configuration identification. This includes the time to create the classifier (in training) and to apply it (in testing), given the feature vectors. The y-axis is the time in seconds. Based on these measurements, linear SVMs has the best accuracy while still exhibiting good efficiency. The time to extract the feature dictionary and feature vectors from the APKs (total for all 10 runs in 10-fold cross validation) is about 12 hours.

G. Case Study: Google Play Apps

We performed a case study of applying our classifiers to a set of popular apps from the Google Play store, which is the largest and most influential app store. We do not have ground truth to measure the quality of our classification, since we do not know the obfuscator provenance of apps from the store. Nevertheless, under the assumption that the classification has high accuracy, this study presents insights into the use of obfuscation techniques in widely-used apps. The apps were crawled from the Play store from May 2016 through August 2016. We collected the top 100 free apps for each store category (e.g., education, finance, etc.) and obtained a total of 2389 APKs; some apps appear in multiple categories.

The analysis of these APKs is summarized in Table VIII. The number in each column is the number of apps classified as being obfuscated by the corresponding tool. More than 55% of the APKs (1330) are identified as being obfuscated by one of the tools we consider. The remaining 1059 APKs may be obfuscated by other obfuscators or not obfuscated at all. The number of obfuscated apps is relatively high. One reason may be that most of the top popular apps are developed by companies instead of individuals. Companies pay more attention to security and intellectual property. For the relative usage of obfuscators, ProGuard is the most widely used tool (about 88% of the APKs). This is not surprising, since ProGuard is already integrated with Android Studio and requires little extra effort to configure and use. Moreover, Google provides an official tutorial on ProGuard and recommends the developers to use it before releasing their apps. The two Chinese tools Legu and Bangcle are rarely used in the Google Play store. The reason may be that most of their users are from China since both tools only have interfaces written in Chinese. The audiences of these Chinese developers are mainly from China. However, app users in China cannot access Google Play.

	ProGuard	Allatori	DashO	Legu	Bangcle	Other
Num.	1170	145	0	15	0	1059

TABLE VIII: Obfuscators for Google Play apps.

For apps using ProGuard and Allatori, we further analyzed their configurations. The results for ProGuard are shown in Table IX. About 70% of the apps use the default or default-opt configuration. The default configuration is the most frequently used. This may be because the default-opt setting is well known to be unsafe due to code optimizations that are not correct in all situations. Table IX also shows the number of apps that customize the corresponding settings. About 65% of apps that do not use default/default-opt employ package modifications. This setting is the most convenient one to use, because it does not require a full list of candidate values, but only a customized package name. The other two settings are not frequently customized. From the 145 apps obfuscated with Allatori, 26 change the renaming and 18 use the flow option.

H. Limitations

The generality of the experimental observations should be interpreted in the context of several limitations of the proposed techniques. First, we only consider five obfuscators. There are other tools that may use different obfuscation techniques. Second, our training data may have limited generality that does

ProGuard stage I			
	default	default-opt	other
Num.	555	236	379
ProGuard stage II			
	package	class	field&method
Num.	246	41	54

TABLE IX: Apps obfuscated with ProGuard.

not allow some characteristics of the obfuscators to emerge. Although the open-source F-Droid apps used for training cover a wide range of target domains and developers, they may differ from closed-source apps in some characteristics that affect the accuracy of the approach. Finally, because we focus on features from the string tables of the dex files, other relevant program properties are not represented. For example, obfuscation features such as function merging and string encryption cannot be captured by the proposed approach. Despite these limitations, we consider these experimental results to be promising initial evidence that automated and precise obfuscator provenance analysis for Android is feasible and efficient.

VI. Related Work

As far as we know, there is no prior work on techniques to identify the provenance of obfuscated Android apps. Studies have been performed on the general topic of software obfuscation, and some work focuses on obfuscation for Android.

General software obfuscation. Researchers have explored various questions related to obfuscation. Schrittwieser et al. [33] measured the performance of different program analyses against obfuscation. Barak et al. [17] theoretically proved that a perfect "virtual black box" obfuscation is impossible. Collberg et al. [16] summarized the existing popular techniques used in obfuscation. The follow-up work of Collberg et al. [18] detailed how obfuscation is used to protect software programs, and discussed the differences among obfuscation, watermarking, and tamper-proofing. Some work has been done on evaluating the quality of obfuscation. Ceccato et al. [34] proposed an approach to assess the difficulty attackers have in understanding and modifying obfuscated code through controlled experiments involving human subjects. Anckaert et al. [35] developed a framework based on software complexity metrics measuring four program properties including code, control flow, data, and data flow. Some studies focus on de-obfuscation. Coogan et al. [36] considered instructions that will impact the interaction between the program and the system and built a program analysis for these important instruction. Madou et al. [37] proposed a graphical and interactive framework for code obfuscation and de-obfuscation.

All studies outlined above focus on general software obfuscation rather than programs for the Android platform. Although such general techniques may be applicable to Android apps, Android software has its own characteristics that make it different from "plain" Java and impose some constraints on the use of obfuscation.

Android obfuscation. Several studies have been performed on how to obfuscate Android apps. Kovacheva [38] studied the Android platform and proposed an obfuscator implementation for Android programs by adding native code wrappers, packing numeric variables, and adding bad code. For the evaluation of obfuscation, Freiling et al. [39] evaluated 7 methods on 240 APKs and showed that these methods are idempotent or monotonic. For Android program de-obfuscation, Bichsel et al. [40] built a probabilistic learning model to perform the de-obfuscation and predict the obfuscated items. Even though these studies focus on Android, none of them attempt to explore the differences between Android obfuscators for the purposes of provenance analysis: Freiling et al. consider the basic techniques of one particular tool, while Bichsel et al. only explore ProGuard.

Impact of Android obfuscation. As discussed earlier, obfuscation of Android apps will affect many legitimate program analyses. Work on clone/repackage detection [41], [42], [43], [5], [44], [45], [46], [47], [48], [49], [50], [51], [52], [6], [53] finds that obfuscation impairs detection results. Studies of malware detection [54], [8], [55], [7], [56], [57], [58] also show that obfuscation is an obstacle to malware analysis. Researchers working on detection of third-party libraries [9], [10], [11] discuss the negative impact of obfuscation on their techniques. Although some of these tools claim to still work well in the presence of obfuscation, none could completely eliminate the obfuscation effects in their experimental evaluations. This highlights the importance of understanding obfuscation, and designing obfuscator-tailored analysis techniques.

Provenance analysis. There is some work on identifying the toolchain used in compiling a given program. Rosenblum et al. [12] propose a machine learning approach to discover the type of compiler, based on properties of the resulting binary code. We also adopt a machine learning method, but for the purpose of obfuscator identification for Android apps. Due to these different targets, the details of the two techniques are significantly different.

VII. Conclusions

We propose a simple, efficient, and effective approach for provenance analysis of obfuscated Android applications. Using features extracted from APKs, we construct several classifiers that determine which obfuscator was used and how it was configured. Although the approach has some limitations, it exhibits high accuracy on apps from F-Droid, and provides insights about the use of obfuscation in popular Google Play apps. The information extracted with the proposed techniques can be potentially used to improve and refine a variety of existing analyses and tools for Android.

Acknowledgments

We thank the MobileSoft reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under awards 1319695 and 1526459, and by a Google Faculty Research Award.

References

[1] Gartner, Inc., "Worldwide traditional PC, tablet, ultramobile and mobile phone shipments," Mar. 2014, www.gartner.com/newsroom/id/2692318.
[2] *ProGuard*, developer.android.com/studio/build/shrink-code.html.
[3] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," in *SIGMETRICS*, 2014, pp. 221–233.
[4] S. Luo and P. Yan, "Fake apps feigning legitimacy," Trend Micro, Tech. Rep., 2014.
[5] M. L. Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting Android reuse studies in the context of code obfuscation and library usages," in *MSR*, 2014, pp. 242–251.
[6] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *TRUST*, 2013, pp. 169–186.
[7] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan, "Evaluation of Android anti malware techniques against Dalvik bytecode obfuscation," in *TrustCom*, 2015, pp. 414–421.
[8] Y. Zhou and X. Jiang, "Dissecting Android malware: characterization and evolution," in *IEEE S&P*, 2012, pp. 95–109.
[9] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Detecting third-party libraries in Android apps," in *ICSE*, 2016, pp. 641–644.
[10] W. Hu, D. Octeau, and P. Liu, "Duet: Library integrity verification for Android applications," in *WiSec*, 2014, pp. 141–152.
[11] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *CCS*, 2016, pp. 356–367.
[12] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *ISSTA*, 2011, pp. 100–110.
[13] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for Android," in *ASE*, 2015, pp. 658–668.
[14] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in Android applications," in *AST*, 2016, pp. 64–70.
[15] Y. Wang and A. Rountev, "Profiling the responsiveness of Android applications via automated resource amplification," in *MobileSoft*, 2016, pp. 48–58.
[16] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," U. Auckland, Tech. Rep. TR-148, 1997.
[17] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, p. 4, 2012.
[18] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proffing, and obfuscation: Tools for software protection," *IEEE Trans. Software Engineering*, vol. 28, pp. 735–746, 2002.
[19] *Soot Analysis Framework*, www.sable.mcgill.ca/soot.
[20] *Androguard*, github.com/androguard/androguard.
[21] *Baksmali*, github.com/JesusFreke/smali.
[22] *Apktool*, ibotpeaches.github.io/Apktool/.
[23] *Dex2jar*, github.com/pxb1988/dex2jar.
[24] *Dexdump source code*, android.googlesource.com/platform/dalvik/+/eclair-release/dexdump/DexDump.c.
[25] R. Harrison, "Investigating the effectiveness of obfuscation against Android application reverse engineering," Royal Holloway University of London, Tech. Rep. RHUL-MA-2015-7, 2015.
[26] *Allatori*, www.allatori.com/.
[27] *DashO*, www.preemptive.com/company.
[28] *Legu*, legu.qcloud.com/.
[29] *Bangcle*, www.bangcle.com/.
[30] *F-Droid*, f-droid.org/.
[31] *Scikit-learn framework*, scikit-learn.org/stable/.
[32] A. Özgür, L. Özgür, and T. Güngör, "Text categorization with class-based and corpus-based keyword selection," in *ISCIS*, 2005, pp. 606–615.
[33] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, 2016.
[34] M. Ceccato, M. D. Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "Towards experimental evaluation of code obfuscation techniques," in *QoP*, 2008, pp. 39–46.
[35] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel, "Program obfuscation: A quantitative approach," in *QoP*, 2007, pp. 15–20.
[36] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *CCS*, 2011, pp. 275–284.
[37] M. Madou, L. V. Put, and K. D. Bosschere, "LOCO: An interactive code (de)obfuscation tool," in *PEPM*, 2006, pp. 140–144.
[38] A. Kovacheva, "Efficient code obfuscation for Android," in *IAIT*, 2013, pp. 104–119.
[39] F. C. Freiling, M. Protsenko, and Y. Zhuang, "An empirical evaluation of software obfuscation techniques applied to Android APKs," in *ICST*, 2014, pp. 315–328.
[40] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev, "Statistical deobfuscation of Android applications," in *CCS*, 2016, pp. 343–355.
[41] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *WiSec*, 2014, pp. 25–36.
[42] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in Android applications through analyzing user interfaces," in *ICPC*, 2015, pp. 163–173.
[43] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *ACSAC*, 2014, pp. 56–65.
[44] M. Sun, M. Li, and J. C. Lui, "DroidEagle: Seamless detection of visually similar Android apps," in *WiSec*, 2015, pp. 9:1–9:12.
[45] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of Android application clones based on semantics," *IEEE Trans. Mobile Computing*, vol. 14, pp. 2007–2019, 2015.
[46] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to Android app clone detection," in *ISSTA*, 2015, pp. 71–82.
[47] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *ICSE*, 2014, pp. 175–186.
[48] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *CODASPY*, 2012, pp. 317–326.
[49] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: detecting cloned applications on Android markets," in *ESORICS*, 2012, pp. 37–54.
[50] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "Piggybacked" mobile applications," in *CODASPY*, 2013, pp. 185–196.
[51] S. Hanna, L. Huang, E. X. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among Android applications," in *DIMVA*, 2012, pp. 62–81.
[52] Q. Guan, H. Huang, W. Luo, and S. Zhu, "Semantics-based repackaging detection for mobile apps," in *ESSoS*, 2016, pp. 89–105.
[53] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Trans. Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.
[54] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang, "Detect Android malware variants using component based topology graph," in *TrustCom*, 2014, pp. 406–413.
[55] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli, "Protection against code obfuscation attacks based on control dependencies in Android systems," in *SERE*, 2014, pp. 149–157.
[56] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on Android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.
[57] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *CCS*, 2013, pp. 329–334.
[58] ——, "Catch me if you can: evaluating Android anti-malware against transformation attacks," *TIFS*, vol. 9, no. 1, pp. 99–108, 2014.