Orlis: Obfuscation-Resilient Library Detection for Android

Yan Wang
Ohio State University

Haowei Wu
Ohio State University

Hailong Zhang
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

Android apps often contain third-party libraries. For many program analyses, it is important to identify the library code in a given closed-source Android app. There are several clients of such library detection, including security analysis, clone/repackage detection, and library removal/isolation. However, library detection is complicated significantly by commonly-used code obfuscation techniques for Android. Although some of the state-of-the-art library detection tools are intended to be resilient to obfuscation, there is still room to improve recall, precision, and analysis cost.

We propose a new approach to detect third-party libraries in obfuscated apps. The approach relies on obfuscation-resilient code features derived from the interprocedural structure and behavior of the app (e.g., call graphs of methods). The design of our approach is informed by close examination of the code features preserved by typical Android obfuscators. To reduce analysis cost, we use similarity digests as an efficient mechanism for identifying a small number of likely matches. We implemented this approach in the Orlis library detection tool. As demonstrated by our experimental results, Orlis advances the state of the art and presents an attractive choice for detection of third-party libraries in Android apps.

KEYWORDS

Android, library detection, library identification, obfuscation, static analysis

ACM Reference Format:
Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. Orlis: Obfuscation-Resilient Library Detection for Android. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3197231.3197248

1 INTRODUCTION

The mobile app market has grown rapidly in the past decade. Android has become one of the largest mobile application platforms, with about 3.5 million apps in the Google Play Store [3], generating about 31 billion USD revenue [8]. Some apps contain sensitive private information or valuable business logic. Due to security considerations, as well as intellectual property concerns, developers often release closed-source application. To further hinder decompilation tools and manual reverse engineering, apps are additionally protected using *obfuscation*. There are several available choices of obfuscation tools, including an obfuscation component in the default Android IDE. An earlier study [55] estimates that about 15% of Google Play apps are obfuscated, and more recent studies indicate even wider use of obfuscation [55].

Android apps often contain third-party libraries. Some usage statistics gathered commercially track 450 popular libraries [2] and show that many of them have significant usage in the Android ecosystem. For example, advertisement libraries, social networking libraries, and mobile analytics libraries are very popular. Some apps use more than 20 third-party libraries [35]. Developers use these components to monetize their apps, integrate with social media, include single-sign-on services, or simply leverage the utility and convenience of libraries developed by others.

Program analysis of Android apps often requires detecting or removing third-party library code as a pre-processing step, since the libraries could introduce significant noise and could substantially affect the results of many analyses. Such library detection may be needed for security analysis [4, 6, 21, 27, 41, 43, 58, 59], clone and repackaging detection [14, 26, 61], and library removal and isolation [49, 60]. The use of libraries can also have security and privacy implications [5, 9, 20, 22, 28, 42, 43, 49, 52–54]—for example, malicious libraries or unintended security vulnerabilities in library code could lead to leaks of sensitive data.

Library detection is complicated significantly by app obfuscation. Attempts to match package/class names or method signatures with ones from known libraries are easily thwarted by the renaming typically done by Android app obfuscators. It is highly desirable to develop obfuscation-resilient third-party library detection. There have been several efforts and research tools aiming to solve this problem. However, as discussed in the next section, these tools have various limitations.

Our contributions. We performed an investigation of popular obfuscators and characterized their features that hinder library detection. We then studied how existing advanced publicly-available library detection tools perform in the presence of such features. Although some of the state-of-the-art tools are intended to be resilient to obfuscation, we argue that there is still room to improve recall, precision, and analysis cost.

Based on these observations, we propose a new approach to detect third-party libraries in obfuscated apps. The approach relies on obfuscation-resilient method-level and class-level information. The richness of this information is significantly higher than what is commonly used by library detection tools—for example, it considers transitive calling relationships between methods, as well as the structural relationships in class hierarchies and methods defined

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5712-8/18/05...$15.00
https://doi.org/10.1145/3197231.3197248

in them. These features capture the *inter*-procedural structure and behavior of the app. This is in contrast with existing approaches, which conceptually consider analysis of each separate method but not the relationship between methods—that is, *intra*-procedural analysis. The design of our approach is informed by close examination of the code features preserved by typical Android obfuscators. The resulting library detection approach exhibits higher precision and recall than prior work of similar nature.

Library detection may have to be performed at large scale—for example, when app-market-scale analyses are applied for clone detection and security analysis. To reduce the cost of the analysis, we use the machinery of *similarity digests* [16, 32, 39, 45]. Such digests are similar to cryptographic hashes such as MD5 and SHA, but with the additional property that a small change in the data object being hashed results in a small change to the hash itself. In our context, similarity digests provide an efficient mechanism for identifying a small number of likely matches. The use of such digests, rather than more traditional hashes, is critical: due to obfuscation, the library code that is preserved in the obfuscated app does not match precisely the code in the original library; thus, direct comparison of hash codes is not possible. We explore several similarity digests and demonstrate empirically that different ones are needed for different aspects of the library detection process.

We implemented this approach in the Orlis tool, which performs obfuscation-resilient library detection using interprocedural code features and similarity digests. Our experimental evaluation studies the effects of various choices in the analysis implementation, and then compares precision/recall with the state-of-the-art LibDetect tool [26]. As demonstrated by our results, Orlis presents an attractive choice for detection of third-party libraries in Android apps. Our tool and benchmarks are available at http://web.cse.ohio-state.edu/presto/, for the benefit of other researchers and tool builders.

2 BACKGROUND

2.1 Android Obfuscation

Code obfuscation is commonly used to protect against reverse engineering. A variety of obfuscation tools for Android are available—for example, the ProGuard tool included in the Android Studio IDE from Google. Many released Android apps are obfuscated [55] since the unobfuscated bytecode is relatively easy to decompile using a number of existing tools (e.g., [50]). When obfuscation is applied, reverse engineering becomes much harder [30].

In our study we investigated three obfuscators: ProGuard [44], Allatori [1] and DashO [18]. They are widely used in Android development. As mentioned earlier, ProGuard is part of Google's Android Studio. Allatori is a tool developed by a Russian company, with corporate clients such as Amazon, Fujitsu, and Motorola. DashO is another commercial tool, produced by a company with thousands of corporate clients.

The features of each obfuscator is summarized in Table 1. All three can obscure identifier names, method names, class names, and package names. Moreover, the values of the new names can be customized, which means that developers can have their own unique renaming schemes. In addition, the tools can modify the package hierarchy, in particular (1) repackaging classes from several packages into a new, different package and (2) flattening the package hierarchy. Both Allatori and DashO can modify the code's control flow and encrypt the constant strings in the program. Finally, the tools can remove unused code ("Code removal" in the table) and add their own utility methods in the new code ("Code addition").

Table 1: Obfuscator features.

	ProGuard	Allatori	DashO
Identifier renaming	Yes	Yes	Yes
Method renaming	Yes	Yes	Yes
Class renaming	Yes	Yes	Yes
Package renaming	Yes	Yes	Yes
Code addition	Yes	Yes	Yes
Code removal	Yes	Yes	Yes
String encryption	No	Yes	Yes
Repackaging/Package flattening	Yes	Yes	Yes
Control flow modifications	No	Yes	Yes
Customized configuration	Yes	Yes	Yes

Note that all three tools can be easily configured to merge the app code with the code of included third-party libraries, and obfuscate this combined code. Such a setup obfuscates the interfaces between the app and its libraries, hides the library features being used by the app, and reduces code size (e.g., unused library classes/methods can be removed). However, this obfuscation makes library detection significantly harder, as described below.

2.2 Existing Library Detection Tools

Given the APK for an (obfuscated) Android app, the goal of library detection is to determine which components of the app belong to some third-party library that was used by the app developers. The granularity of the answer is typically considered at two levels: package level and class level. In package-level matching, a package in the package tree of the app is determined to be from some library. In class-level matching, a class in the app code is detected to be a library class. Package-level matching has a number of limitations, as described below. For the purposes of subsequent analyses (e.g., app clone detection [26]), class-level matching is both more general and more useful.

There are currently two approaches for recognizing third-party libraries in Android apps. The first is to identify a component that occurs in many apps, and to consider it to be an instance of some unknown library. The specific library that was the original source of the component remains undetermined. The second approach is based on a database of known libraries. In this case, given an never-before-seen app, the analysis determines which libraries from the database are present in the app.

2.2.1 Detection of unknown libraries. Several approaches perform library detection without having a database of known libraries. Such approaches divide an app into components which are regarded as library candidates. Then a similarity metric or a feature-based hashing algorithm is used to classify these candidates. If a group of similar candidates exists in many different apps, components in that group are considered instances of the same unknown library.

An advantage of this approach is that there is no need to maintain a library database. One example from this category is the approach used by Chen et al. [12], which mined libraries from a large number of apps. However, when obfuscation is considered, this approach cannot perform well because several key assumptions are violated. The first assumption is that all instances of a library included by different apps have the same package name. This assumption is the basis of clustering algorithms used in similarity-based library identification. However, using off-the-shelf obfuscators, it is easy to violate this assumption.

Some researchers have considered the possibility that using package names makes library identification less robust. A recent tool called LibRadar [37] uses an algorithm that takes obfuscated package names into consideration. LibRadar classifies library candidates through feature hashing and therefore does not need package-name-based clustering. However, LibRadar recognizes library candidates according to the directory structures of packages. In particular, it requires a library candidate to be a subtree in the package hierarchy. This is another assumption that is easily violated through standard transformations available in obfuscation tools (as illustrated in Table 1). LibD [34] is another tool that does not rely on the hierarchy of the package tree, but instead uses the relationships among packages—e.g., inclusion and the inheritance relationships between classes across different packages. However, these relationships can be dramatically changed by obfuscation because classes from several packages may be merged into one single package.

Tools from this category (e.g., WuKong [56] and AnDarwin [15]) may also use features such as the number of different types of instructions, a method's control-flow graph, or the number of API calls, in order to define hashes for methods or classes. These hashes are compared during matching, with hash equality used as evidence of a precise match. This is problematic because obfuscation may (1) modify a method's control-flow graph, (2) remove unused methods and classes, and (3) add new utility methods. All three of these changes are illustrated in Table 1. Such changes modify method hashes and class hashes. To summarize, existing work on detection of unknown libraries is not resilient to obfuscation because it lacks prior knowledge of possible libraries and the detection is entirely based on obfuscation-sensitive similarity metrics.

2.2.2 Detection using a repository of known libraries. The alternative approach, which relies on knowledge of existing libraries (using some pre-assembled repository of such libraries), is more promising when dealing with obfuscation. A representative example of a tool from this category is LibScout [5]. This tool's matching is based on the package structure and hashes of classes in the package. Class hashes themselves are based on hashes of simplified ("fuzzy") forms of method signatures. Although robust against control flow modification and package/class/identifier renaming, LibScout is not fully resilient to obfuscation. For example, flattening the package hierarchy would still defeat this approach because the package hierarchy is modified and the boundaries between app and library code become blurred. Another problem is the removal of unused library code, or the addition of obfuscation-tool-specific utility methods. As a result of these changes, it is common for an app to contain a strict subset of the set of classes and methods that are observed in the library code inside the database. This happens when the obfuscation tool removes library methods and classes that are not used, directly or transitively, by the app code. This causes mismatches for class hashes and the ultimate failure of library detection. Our experience with various obfuscation tools [57] confirms that such code changes are common in practice.

Currently, the most sophisticated tool in this category is LibDetect, which is a component of the CodeMatch tool for app clone and repackage detection [26]. The evaluation of LibDetect indicates that it outperforms alternative approaches [33, 37]. This approach uses five different abstract representations of a method's bytecode to match app methods against library methods. Then a mapping between an app class and a library class is established according to method matches. The approach is resilient to many obfuscation techniques, including package renaming and removal of unused library code. However, the bytecode in a method's body can the changed in a variety of ways during obfuscation. A few examples of such changes are adding dead code (unreachable statements as well as spurious conditionals with dead branches), replacing statements with equivalent constructs (e.g., changing an if-then to a try-catch), and modifying expressions to use different operators (e.g., replacing multiplication with bitshift). These bytecode changes can affect the code representations used by LibDetect and the matching based on them. Section 4 presents a detailed comparison of the library detection performance of LibDetect and Orlis.

There are other studies that use pre-computed information about known libraries to detect libraries in Android apps. For example, AdRisk [28] proposes to identify potential risks posed by advertisement libraries; they use a whitelist of such libraries. Book et al. [9] also use a whitelist of advertisement libraries to investigate library changes over time. For Android app clone detection, Chen et al. [11] use a library whitelist to filter third-party libraries when detecting app clones. Because such approaches only compare package names, they cannot handle aggressive obfuscation.

To summarize, none of the existing tools are highly resilient against the code modification features that are easily available in popular obfuscators. This motivates our work to develop a new library detection approach with better resilience to a wide range of obfuscation transformations.

3 ORLIS: AN OBFUSCATION-RESILIENT APPROACH FOR LIBRARY DETECTION

We developed a library detection approach that belongs to the second category described in the previous section: a given obfuscated app is compared against a repository of known libraries, and high-likelihood matches are reported to the user. However, unlike existing tools, we do not consider the app package structure and the detailed method body bytecode, because they can be modified significantly by obfuscation. The output of our approach is class-level mapping between application code and library code: for each app class, the analysis reports (1) whether this class likely originated from one of the libraries in the repository, and (2) the specific library class that is the most likely match for the app class. As discussed earlier, class-level matching provides essential information for subsequent analyses such as app clone/repackaging detection. Furthermore, the reported app-to-library matching is injective: at most one app class is matched with a particular library class.

3.1 Assumptions About Obfuscation

Our approach is based on several assumptions derived from our studies of Android obfuscators (Section 2). Prior work on library detection typically does not define or validate its assumptions on the properties of obfuscation. As a result, it is hard to reason about the generality of various techniques and to perform systematic comparisons between them.

3.1.1 Method-level assumptions. We assume that a method from a library is either included in the app as a separate entity, or not included at all. This means that during obfuscation, a method is neither split into several new methods, nor merged with another method. We also assume that the number and types of method formal parameters are preserved, as well as the return type. Of course, we need to assume that the name of the method could have been modified by the obfuscator. Similarly, if a method parameter/return type is defined by a library class, it is possible that this class was renamed by the obfuscator—both in the class definition and in the method signature.

Our approach also assumes that regardless of changes to the method body (e.g., modifications to the method's control-flow graph, changes to statements and expressions, addition of unused code), the *calling relationships between two library methods are preserved*. Specifically, we assume that if method m_1 contains a call site that may call m_2 at run time, and the obfuscator includes m_1 in the app, it will also include m_2 and will preserve the corresponding call site inside m_1. We have not observed obfuscators that violate this assumption; one reason may be that removal of such call sites (e.g., via inlining) is complex to implement and could introduce semantic errors due to polymorphic calls, reflection, and dynamic class loading.

3.1.2 Class-level assumptions. Another group of assumptions relates to the properties of library classes. As with methods, we assume that a library class is either fully excluded from the app, or included as a separate entity and not split/merged. We also assume that library classes may have been renamed, but the class hierarchy is preserved—that is, if a library class C is included in the app, so is the chain of C's transitive superclasses. In addition, we assume that some methods from C may have been removed by the obfuscator, but the ones that are preserved are still members of C after obfuscation. Note that the obfuscator may have added new methods to C—for example, we have seen cases where an obfuscator adds utility methods for string decryption.

3.2 Code Features and Detection Workflow

Based on these assumptions, we define a set of code features used for library detection. First, for each method in a library or in an analyzed app, we use a *fuzzy signature* as defined by others [5]. These signatures are used to build a fuzzy call graph for each method, which is then mapped to a *string feature* for the method (Section 3.3). The method features for a library are used to create a *library digest*; a similar *app digest* is built from the features of app methods (Section 3.4). Method features are also used to compute a *class digest* for each class, based on the methods appearing in the class and in its transitive superclasses. In all cases, similarity digests are used (Section 3.5).

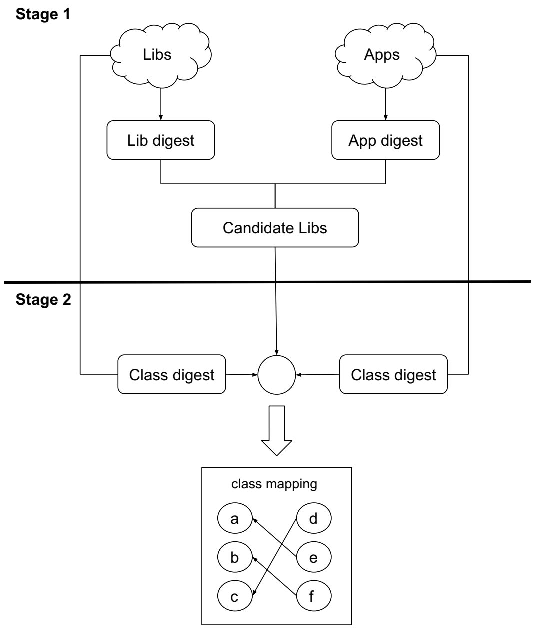


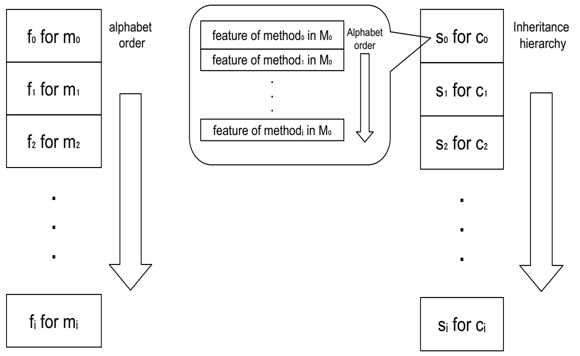
Figure 1: Workflow for library detection.

Using this approach, for each library from the repository we can pre-compute a library digest as well as class digests for its classes. Given an unknown obfuscated app, its app digest and class digests are computed. Based on this information, the library detection workflow proceeds in two stages, as shown in Figure 1. Stage 1 compares the app digest with each library digest. Libraries that are highly unlikely to be included in the app are removed from further consideration. This decision is based on similarity scores derived from the digests. After this stage, only a small number of libraries from the repository are left as candidates. In Stage 2, an injective mapping from app classes to library classes is determined, based on the similarity of class digests. Only classes from the candidate libraries are considered for this mapping.

3.3 String Features for Methods

Fuzzy signatures are computed as follows. Given a method signature containing a name, parameter types, and return type, a simplified signature is created by (1) removing the method name and (2) replacing all classes defined in the library/app with a single placeholder name. Names of Android framework classes and Java standard library classes remain unchanged. For example, if we consider a library method with signature `int methodA(android.view.View, ClassA,int,java.lang.String)`, the corresponding fuzzy signature is `int(android.view.View,X,int,java.lang.String)`. Here `View` and `String` are standard library classes that would not be renamed by an obfuscator, while library class `ClassA` is replaced with a placeholder `X` because obfuscation may change its name when the library is included in an app and that app is obfuscated.

Recall the assumption that (transitive) calling relationships between methods would not be affected by obfuscation. If a library



(a) String feature for an app or a library **(b) String feature for a class C_0**

Figure 2: String features used for matching.

method m is included in an obfuscated app, m's call graph in the library (i.e., m, its transitive callees, and the call edges between them) would be a subgraph of m' call graph in the app. Note that we cannot claim that the two call graphs are identical: it is possible that there are extra callees of m in the app, due to (1) app subclasses that override methods defined in library superclasses, and (2) calls to utility methods inserted by the obfuscator. One can try to determine whether m is present in a given obfuscated app by (1) constructing m's call graph in the library, (2) constructing the app's call graph, (3) replacing each method in the graphs with its fuzzy signature, and (4) searching for an isomorphic subgraph. Unfortunately, subgraph isomorphism is a classic NP-complete problem.

Instead, we use a string representation of the fuzzy call graph of each method in a library or an app to define features that are then used to create similarity digests. Specifically, consider a library with a set of methods $\{m_1, m_2, \ldots, m_n\}$. For each method m_i, we compute a feature f_i: a string containing the fuzzy signatures of all methods reachable from m_i in the library call graph, including m_i itself. To make f_i deterministic, the fuzzy signatures are sorted alphabetically and then concatenated to create the final string feature f_i. The same approach is used to define a feature f_j for a method m_j in a given obfuscated app.

3.4 Stage 1: App-Library Similarity

As described earlier, the purpose of this stage is to determine the similarity between an app and a library in order to filter out libraries that are unlikely to be included in the app. For each app (or library), a string feature f_{app} (or f_{lib}) is computed based on the method string features described above. The string features for the methods are sorted alphabetically and then concatenated, as illustrated in Figure 2a. A digest is computed from this feature. The pre-computed digests in the library repository are compared for similarity with the digest of the given obfuscated app. Experimental results presented in Section 4 show that this approach is very effective in filtering out a large number of libraries.

Algorithm 1: Class-level mapping

```
Input: SortedPairs: list of $\langle ac_i, lc_j \rangle$ sorted by $s_{ij}$
1  Result ← { }
2  AppClasses ← { }
3  LibClasses ← { }
4  foreach $\langle ac_i, lc_j \rangle \in$ SortedPairs in order do
5      if $s_{ij} <$ threshold then
6          break
7      if $ac_i \in$ AppClasses then
8          continue
9      if $lc_j \in$ LibClasses then
10         continue
11     AppClasses ← AppClasses ∪ {$ac_i$}
12     LibClasses ← LibClasses ∪ {$lc_j$}
13     Result ← Result ∪ {$\langle ac_i, lc_j \rangle$}
```

3.5 Stage 2: Class Similarity

Stage 2 determines a mapping from app classes to library classes. Suppose an app has a set of classes $S_{app} = \{ac_1, ac_2, \ldots, ac_n\}$ and a set of candidate libraries $\{lib_1, lib_2, \ldots, lib_k\}$ as determined by Stage 1. Let $S_{lib} = \{lc_1, lc_2 \ \ldots, lc_m\}$ be the union of the set of classes in lib_i over all i.

For each class in S_{app} or S_{lib}, a string feature is computed as illustrated in Figure 2b. Consider a class C_0 and the chain of its superclasses $C_1, C_2, \ldots, C_k$, accounting only for classes that are defined in the app (or the library)—that is, excluding classes such as `java.lang.Object`. As discussed earlier, we assume that if C_0 is included in the app, so are all C_i in this list. Let M_i be the set of methods defined in C_i. We consider the string features of all methods in M_i, sort them alphabetically, and concatenate them to obtain a string s_i ($0 \leq i \leq k$). Then the concatenation of $s_0, s_1, \ldots, s_k$ (in that order) defines the string feature for C_0. Finally, a digest of this string is computed.

A pair-wise similarity score s_{ij} is computed for classes ac_i and lc_j by comparing their class digests. Then all pairs $\langle ac_i, lc_j \rangle$ are sorted based on s_{ij} and the sorted list is processed using Algorithm 1. The resulting class-level mapping is clearly injective.

3.6 Implementation

To evaluate this approach, we implemented it in the ORLIS tool. The library/app bytecode is analyzed using the Soot [50] analysis framework. The call graph is computed using class hierarchy analysis to resolve polymorphic call sites. Digest `sdhash` [48] is used in Stage 1, while digest `ssdeep` [51] is used in Stage 2. This choice of digests is discussed in Section 4.

To determine the set of transitive callees for each method, we first identify the strongly-connected components (SCC) in the call graph. They are used to create the SCC-DAG, in which nodes are SCCs and an edge represents the existence of a call graph edge that connects two SCCs. Reverse topological sort order traversal of the SCC-DAG is then used to compute, for each SCC, the SCCs transitively reachable from it. The method features are derived from this information.

4 EVALUATION

Our experimental evaluation had two main goals: (1) to study the effects of design choices in the performance of ORLIS, and (2) to

Table 2: Apps in *FDroidData*

	ProGuard	DashO	Allatori
#Apps	203	215	241

compare Orlis with the state-of-the-art LibDetect tool. Evaluation considered precision, recall, and analysis running time. All experiments were performed on a machine with an Intel Core i7-4770 3.40GHz CPU and 16GB RAM. The reported running time is the sum of user CPU time and system CPU time.

4.1 Evaluation with Open-Source Apps

Two different data sets were used in our evaluation. The first one, denoted by *FDroidData*, is obtained from the F-Droid repository [23]. The second data set, denoted by *LibDetectData*, is based on the data used in the evaluation of the LibDetect tool [26]; details of this data set and its uses are presented in Section 4.2.

To collect *FDroidData*, we gathered apps with available source code from a variety of app categories and tried to build them with the Gradle tool; 282 apps were built and used to construct our data set. A total of 453 unique library jars were included in this set of apps. These jars formed the library repository for this data set. The apps were then obfuscated by us. The app source code is necessary because (1) the obfuscators are applied to Java bytecode constructed from the source code, and (2) we can determine which third-party library jar files are included in the build process.

We attempted to obfuscate each app using the tools described in Section 2: ProGuard, DashO, and Allatori. The number of resulting apps is shown in Table 2. The numbers differ across obfuscators because sometimes an obfuscator may fail to obfuscate an app. The obfuscators were executed in a configuration that includes the third-party libraries in the scope of obfuscation, and performs package flattening and renaming. As discussed in Section 2, such obfuscation is not handled by many library detection tools. LibDetect [26], which according to its evaluation represents the most advanced current approach in terms of precision and recall, is designed to handle this issue. We used *FDroidData* (which contains the total of 659 apps from Table 2) to compare the performance of Orlis and LibDetect. In addition, this data set was used to study the effects of various similarity digests and thresholds in the design of Orlis.

4.1.1 Comparison with LibDetect. For each obfuscated app in this set, we established the ground-truth mapping from app-included library classes to the library classes in the library repository, using the logs provided by the obfuscation tools. We then considered the sets of pairs (app class,library class) reported by Orlis. For each app, the precision for this app was computed as the ratio of the number of reported true pairs to the total number of reported pairs. Recall for an app was computed as the ratio of the number of reported true pairs to the total number of true pairs. The F1 score for each app was also computed (2 times the product of recall and precision, divided by the sum of recall and precision). Similar metrics were computed for LibDetect. That tool outputs information at the package level (i.e., pairs of packages). We modified the tool to print the set of class-level pairs used to compute the package-level pairs, and used those class-level pairs to compute recall and precision.

Table 3: Comparison with LibDetect on *FDroidData*

	recall	precision	F1
LibDetect	0.10	0.63	0.17
Orlis	0.63	0.71	0.67

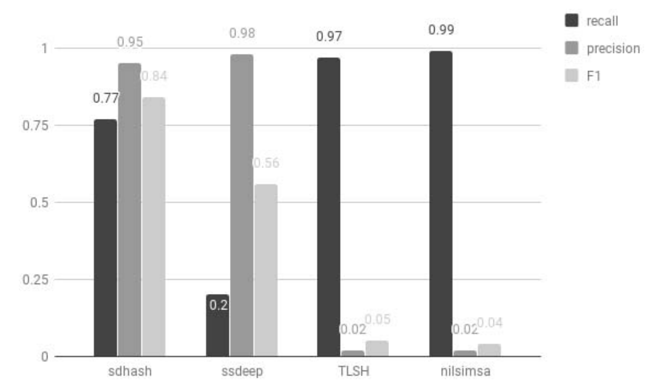
Table 4: Similarity score metrics

Similarity digest	`sdhash`	`ssdeep`	`TLSH`	`nilsimsa`
Score range	0–100	0–100	0–∞	0–128
No similarity	0	0	N/A	0
Identical	100	100	0	128

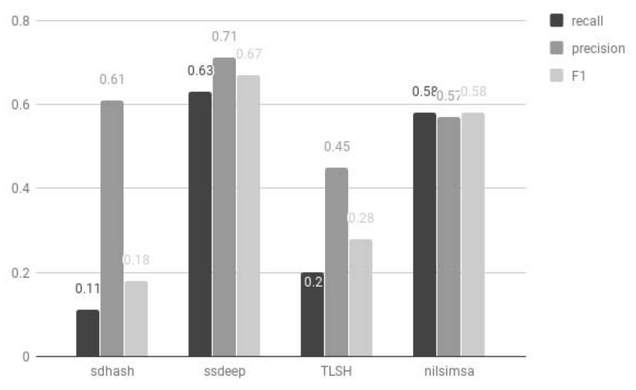
The results of this experiment are presented in Table 3. Each cell in the table shows the average value across all apps in the data set. In these experiments Orlis was configured with `sdhash` for Stage 1 and `ssdeep` for Stage 2, as described below. We examined manually the output of LibDetect to determine the reasons for the reported low recall. We observed that the tool computes a relatively small number of matching class pairs; a possible explanation is that the approach does not consider methods whose size is below a certain threshold. When mapped to package-level information, the recall increases significantly. This can be observed in the results presented in Section 4.2, which describes another comparison with LibDetect.

4.1.2 Selection of similarity digests. At present, there are four popular public implementations of similarity digests that could be used for our purposes: `sdhash` [45], `ssdeep` [32] , `TLSH` [40], and `nilsimsa` [16]. All of them are able to (1) generate a digest for a given byte array, and (2) compute a similarity score between two digests to indicate the similarity of the represented byte data. Their similarity score metrics are shown in Table 4. The scores for `sdhash`, `ssdeep` and `nilsimsa` are in a fixed range. For data without any similarity, the similarity score is 0. `TLSH` uses a similarity score of 0 to indicate that the compared digests are identical. As the digest difference increases, so does the similarity score, without any pre-defined bound.

Figure 3a shows how Stage 1 filtering performs when using these digests. Recall that this stage compares an app digest with each library digest in the repository, in order to determine which libraries are likely to have been included in the app. Conservatively, the tool reports a library as a possible candidate if there is any similarity between the digests. For `sdhash`, `ssdeep`, and `nilsimsa` this means a similarity score greater than zero. `TLSH` does not have a pre-defined value to represent no similarity; we used 300 because this is the value used in the tool's own evaluation [39]. Given the reported candidate library jars, we can compute precision, recall, and F1 score for each app. The averages of these measurements over the apps in the data set are presented in Figure 3a. When gathering and analyzing these measurements, we found that the results were skewed if the repository contained several versions of the same library, and each was treated as a separate library for computing these metrics. Thus, for Figure 3a, we computed the metrics by treating different versions of the same library as being the same entity (in the numerator and denominator of recall



(a) Precision, recall and F1 for stage I



(b) Precision, recall and F1 for stage II

Figure 3: Performance of similarity digests for *FDroidData*

and precision). This produced metrics that faithfully represent the relevant properties of the similarity digests.

Among the four choices, sdhash exhibits the highest F1 value and the best trade-off between recall and precision. Thus, we used this choice for the remaining experiments described in this section. The average number of candidate library jars per app is around 22, which is only 5% of the total number of libraries in the repository. The results are similar for the *LibDetectData* dataset described later: the average number of candidate library jars per app is also around 22, for a repository containing 7519 library jars. This is a promising indication that the number of candidate libraries produced by Stage 1 is independent of the total number of libraries. Note that depending on the desired trade-offs of the tool, another choice of a similarity digest could be made. For example, recall can be increased with TLSH and nilsimsa, at the expense of precision; as a result, fewer libraries would be filtered and the cost of subsequent Stage 2 matching would increase.

Figure 3b shows the performance of Stage 2. This stage compares app classes with classes in the candidate library jars. Recall from Algorithm 1 that class pairs are processed in sorted order of their similarity scores: decreasing order for sdhash, ssdeep, and nilsimsa and increasing order for TLSH. In addition, a pair is considered only if its similarity score exceeds a pre-defined threshold (or is below that threshold, for TLSH). The default thresholds are the same as the ones for Stage 1: 0 for sdhash, ssdeep, and nilsimsa and 300 for TLSH. The effect of different threshold values is studied further in Section 4.3. For each app in *FDroidData*, we computed precision and recall as described earlier. The results for ssdeep in the figure match those in Table 3. As the measurements show, ssdeep has the best recall, precision, and F1 score.

It is worth noting that sdhash and TLSH require certain amount of data: in order to compute a digest, the data size should exceed 512 bytes and 256 bytes, respectively. In *FDroidData*, only about 40% of library class features exceed the 512 byte threshold and about 51% exceed the 256 byte threshold. Classes for which this threshold is not met do not have digests and do not appear in the reported mapping. This causes the low recall for sdhash and TLSH.

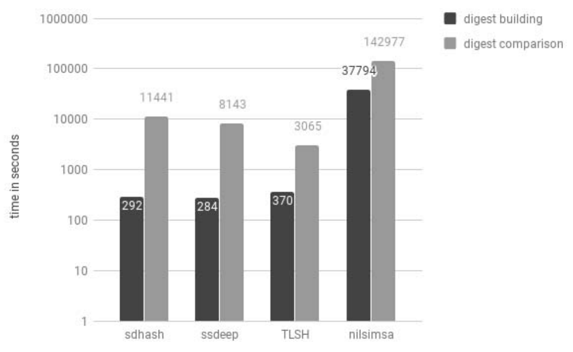
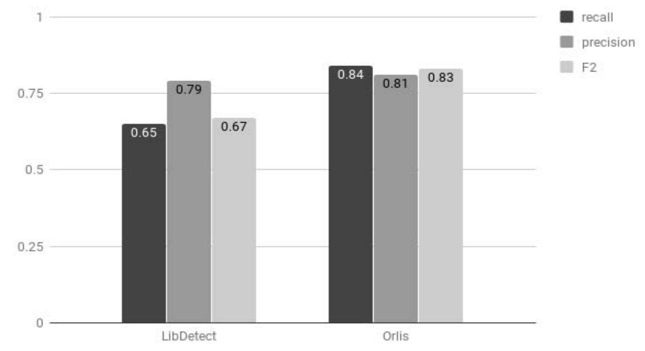


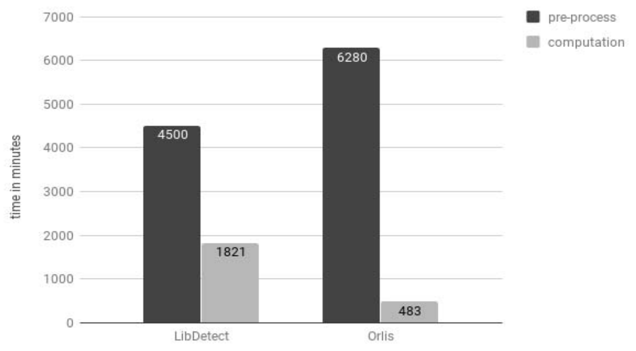
Figure 4: Running time for *FDroidData*

We further examined various missing and mis-mapped classes. Most of them have very simple call graphs; some even have no callees at all. As a result, their string features are very short and likely to be similar. This leads to a high chance of digest similarity (or no digest at all) and makes them indistinguishable.

We also measured the running time of the four digest functions. There are two components to this cost. The first one is digest building, which is the time to calculate the digest value based on the given string feature. The second component is digest comparison, which computes a similarity score based on the digest values. Figure 4 shows measurements for both components. "Digest building" represents the time to calculate digests for the 659 apps and 453 libraries in *FDroidData*, while "digest comparison" is the time to compute the similarity scores between apps and libraries in Stage 1 and between app classes and library classes in Stage 2. The results show that nilsimsa is significantly more expensive than the remaining digests. For digest building, the other three choices do not show substantial differences: all three complete in around 5–6 minutes. For digest comparison there are more significant differences in running time. Although slower than TLSH, sdhash and ssdeep can complete the digest comparison in less than 200 minutes. Taking



(a) Recall, precision and F2



(b) Running time

Figure 5: Comparison with LibDetect

into account the recall, precision, and cost of each digest function, we decided to use `sdhash` in Stage 1 and `ssdeep` in Stage 2 of ORLIS.

4.2 Evaluation with Closed-Source Apps

The second dataset we used, denoted by *LibDetectData*, is based on the data set used for the evaluation of the LibDetect tool [26]. As mentioned earlier, this tool presents the state of art in Android library detection. In order to compare the performance of ORLIS with LibDetect, we used data made publicly available by the authors of this work.[1] We tried to collect the same data used in the evaluation of LibDetect and to use the exact same metrics. In this prior work, 1000 apps were collected from five different app stores. Then, the ground truth for those apps was constructed manually. Since the authors of LibDetect were unable to distribute these apps, we searched for and downloaded the apps with the same names. To ensure that the app is the same as the one used in the evaluation of LibDetect, we matched all app packages with ones in the ground truth from LibDetect's web site [13]. If any package name did not match, we considered this as evidence that we may have a different version of the app, and removed that app from the data set. As a result, we were able to obtain 712 apps that matched ones used in this prior work. To gather the libraries used in the repository, we attempted to download all libraries used in LibDetect's experiments, based on the library URL from the same web site. We successfully obtained 7519 libraries out of the 8000 listed at the web site. Some libraries are missing because the corresponding URLs are no longer available. The evaluation of LibDetect uses F2 scores, and we also computed the F2 scores for our evaluation.

The metrics presented in the evaluation of LibDetect are at the package level: the ground truth is a set of library packages for each app. The class-level pairs computed by the analysis are used to construct the corresponding package sets; we did the same for the output of ORLIS. Recall and precision are computed based on these package sets. Following the metrics used in this prior work, both precision and recall were computed by using the average number of true positives, false positives, and false negatives across all apps.

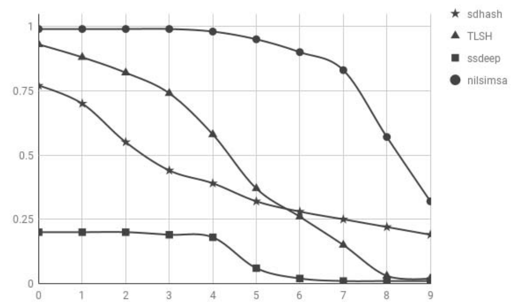
[1]We sincerely thank the authors of LibDetect for their valuable help with providing information about their experiments and benchmarks.

For the i-th analyzed app, let the number of true positives, false positives, and false negatives be tp_i, fp_i, and fn_i respectively. The average number of true positives $\overline{tp}$ is the average of all tp_i. The average number of false positives $\overline{fp}$ and the average number of false negatives $\overline{fn}$ are defined similarly. The overall precision, recall, and F2 score are computed as follows:

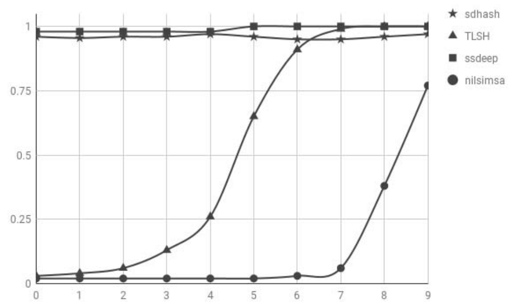
$$recall = \frac{\overline{tp}}{\overline{tp} + \overline{fn}}$$
$$precision = \frac{\overline{tp}}{\overline{tp} + \overline{fp}}$$
$$F2 = \frac{5 \times precision \times recall}{4 \times precision + recall}$$

We computed the same metrics in our evaluation. Since the ground truth provided at LibDetect's web site does not contain class-level mappings, we were unable to compute precision and recall at the class level (unlike for *FDroidData*, where we know the class-level ground truth). The results from this experiment are shown in Figure 5a. Overall, ORLIS exhibits higher precision and recall. This indicates that using interprocedural features such as call graphs and class hierarchies is a viable approach for library detection.

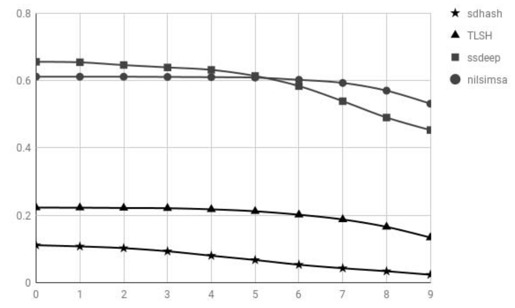
We also measured the running time for both tools. LibDetect needs to parse each known library and store the necessary information in a database. Similarly, ORLIS processes each known library, builds call graphs, derives string features and digests, and stores them in its repository. Repository building is a one-time cost; it is also highly parallelizable. Typically, the number of known libraries is smaller than the number of apps that would be analyzed against the repository. Figure 5b shows the time for both repository building ("pre-process") and app analysis ("computation"). ORLIS takes more time to construct its repository, but the cost of app analysis is lower. Besides the time cost, both LibDetect and ORLIS have to store the library information to disk. For the libraries in the *LibDetectData* data set, LibDetect uses 8812 MB space in a MySQL database, while ORLIS uses 460 MB of files on disk. These measurements do not include the space to store the library jars.



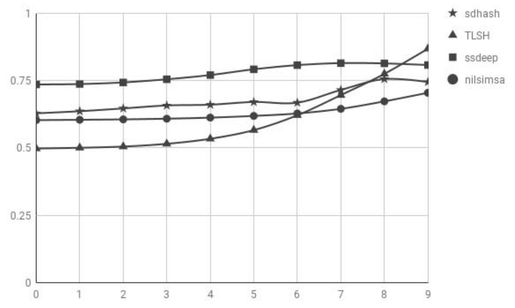
(a) Changes of recall in Stage 1



(b) Changes of precision in Stage 1



(c) Changes of recall in Stage 2



(d) Changes of precision in Stage 2

Figure 6: Effect of thresholds for *FDroidData*

4.3 Effect of Thresholds

In our approach, it is possible to set a cut-off threshold in Stage 1 or Stage 2. Pairs whose similarity score does not exceed the threshold (or, for TLSH, is not smaller than the threshold) are considered non-matching by default. The experiments described so far use the threshold of 300 for TLSH and 0 for the remaining digest functions. It is interesting to see how different threshold values affect both stages. We selected 10 thresholds, equally spaced, from the "least restrictive" end to the "most restrictive" end of the similarity score range. Figure 6 shows the changes of the recall and precision for Stage 1 and Stage 2 using *FDroidData*. As expected, recall in Figure 6a and Figure 6c decreases while the precision in Figure 6b and Figure 6d increases when the threshold becomes more restrictive. Depending on the use case, the desired trade-off may be selected by choosing the appropriate threshold. For example, when using library detection as a pre-processing step in clone/repackage detection (to remove library classes from further consideration), a relatively loose threshold may be more appropriate in order to detect as many as library classes as possible.

5 RELATED WORK

Android obfuscation. There is a small but growing body of work related to Android obfuscation. One representative example is a new obfuscation approach that integrates several techniques including native code, variable packing, and dead code insertiončitekovacheva-iait13. Another study considered several obfuscation techniques and their properties (e.g., monotonicity) [24]. De-obfuscation approaches have also been proposed. One recent example is an approach based on a probabilistic model [7]. ProGuard is the only obfuscator considered in that work; further, the technique considers only the renaming of program elements (e.g., methods and classes), but not repackaging or control-flow modifications. Wang et al. [57] propose a machine learning approach to identify the obfuscator used to modify a given obfuscated app. Hammad et al. [29] studied the impact of obfuscation on Android anti-malware products by investigating 7 obfuscators and 29 obfuscation techniques. This work aimed to analyze the influence of obfuscation on existing tools. Garcia et al. [25] proposed a machine learning approach to identify the family of Android malware even in the presence of obfuscation, by using Android APIs, reflection, and native calls as features. This study focused on anti-malware analysis and considered obfuscation-resilient properties different from the ones used in our work. We also investigate popular obfuscators and the techniques they employ, but our goal is to identify code features that can be used to identify third-party libraries.

Third-party library detection in Android. As discussed earlier, there is a body of work on library detection for Android [5, 9, 11, 12, 15, 26, 28, 34, 37, 56]. Some of these techniques have obfuscation-resilient aspects, but as described in Section 2, several assumptions used in these tools are violated by commonly-used Android obfuscators. Our approach is specifically designed to exploit obfuscation-resilient features. AdDetect [38] and PEDAL [36] use machine learning to detect advertisement libraries in Android apps. However, these approaches can only handle this particular category of libraries, while our approach is applicable to any library. Li et al. [33] investigated the common libraries used in Android apps by mining libraries in a large number of apps. The purpose of this work is to study popular Android libraries. The resulting library package list is not aimed at detecting libraries for a given obfuscated app.

Similarity digests. Similarity digests are somewhat similar to standard hashes, but allow one to measure the similarity between two data objects based on comparison of their digests. Several different techniques have been developed in this area. One scheme is based on feature extraction (e.g., `sdhash` [45]), which is a statistical approach for selecting fingerprinting features that are most likely to be unique to a data object. Another scheme is fuzzy hashing (e.g., `ssddep` [32]), which uses rolling hashes and produces a pseudo-random value where each part of the digest only depends on a fragment of the input. The result is treated as a string and is compared with other digests on the basis of edit distance. Roussev et al. [46] proposed a similarity approach that uses partial knowledge of the internal object structure and Bloom filters. Follow-up work [47] attempts to balance performance and accuracy by maintaining hash values at several resolutions, but requires understanding of the syntactic structure of data objects, which affects its generality. Locality sensitive hashing is also a type of similarity digest. Locality-sensitive hashing [31] was originally used for a randomized hashing framework for efficiently approximating nearest neighbor search in high dimensional space. Since then, it has become one of the most popular solutions for this type of problem. In general, there are two approaches: (1) approximating the distance between data objects by comparing their hashes, and (2) mapping similar objects to the same bucket in a hash table, after which search is performed within a bucket. The hashing functions `TLSH` [40] and `nilsimsa` [16] discussed in Section 4 belong to the former category. For the second category, many hash functions (e.g., [10, 17, 19]) have been developed for Euclidean distance, angle-based distance, Hamming distance, etc. However, such distance metrics cannot be applied directly to our data.

6 CONCLUSIONS

Identification of third-party libraries in Android apps is of significant interest for a variety of clients. Existing library detection techniques do not handle well the variety of code transformations employed by current Android obfuscators. We propose to use call graphs and class hierarchies as the basis for library matching, together with similarity digests for efficiency. Our experimental results indicate that the proposed Orlis tool advances the state of the art in obfuscation-resilient library detection for Android.

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

REFERENCES

[1] Allatori 2017. *Allatori. www.allatori.com.*
[2] Appbrain 2018. *Appbrain: Android library statistics. www.appbrain.com/stats/libraries.*
[3] Appbrain 2018. *Appbrain: Number of Android applications. www.appbrain.com/stats/number-of-android-apps.*
[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android Apps. In *PLDI.* 259–269.
[5] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *CCS.* 356–367.
[6] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. 2016. R-Droid: Leveraging Android app analysis with static slice optimization. In *ASIACCS.* 129–140.
[7] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical deobfuscation of Android applications. In *CCS.* 343–355.
[8] Bloomberg 2017. *Google's Android generates 31 billion revenue. www.bloomberg.com/news/articles/2016-01-21/google-s-android-generates-31-billion-revenue-oracle-says-ijor8hvt.*
[9] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal analysis of Android ad library permissions. *CoRR* abs/1303.0857 (2013).
[10] Moses Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC.* 380–388.
[11] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE.* 175–186.
[12] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS. In *IEEE S&P.* 357–376.
[13] CodeMatch 2017. *CodeMatch. http://www.st.informatik.tu-darmstadt.de/artifacts/codematch/.*
[14] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on Android markets. In *ESORICS.* 37–54.
[15] Jonathan Crussell, Clint Gibler, and Hao Chen. 2015. AnDarwin: Scalable detection of Android application clones based on semantics. *IEEE Trans. Mobile Computing* 14 (2015), 2007–2019.
[16] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2004. An open digest-based technique for spam detection. In *ISCA.* 559–564.
[17] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2011. Fast locality-sensitive hashing. In *KDD.* 1073–1081.
[18] DashO 2017. *DashO. www.preemptive.com/company.*
[19] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG.* 253–262.
[20] Dropbox Blog 2017. *Security bug resolved in the Dropbox SDKs for Android. blogs.dropbox .com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android.*
[21] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *CCS.* 73–84.
[22] William Enck, Damien Octeau, Patrick D. McDaniel, and Swarat Chaudhuri. 2011. A study of Android application security. In *USENIX Security.*
[23] F-Droid 2017. *F-Droid Repository. f-droid.org.*
[24] Felix C. Freiling, Mykola Protsenko, and Yan Zhuang. 2014. An empirical evaluation of software obfuscation techniques applied to Android APKs. In *ICST.* 315–328.
[25] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of Android malware. *TOSEM* 26, 3 (2018), 11:1–11:29.
[26] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: Obfuscation won't conceal your repackaged app. In *FSE.* 638–648.
[27] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of Android applications in DroidSafe. In *NDSS.*
[28] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec.* 101–112.
[29] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *ICSE.*
[30] Rowena Harrison. 2015. *Investigating the effectiveness of obfuscation against Android application reverse engineering.* Technical Report RHUL-MA-2015-7. Royal Holloway University of London.
[31] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC.* 604–613.

[32] Jesse D. Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation* 3 (2006), 91–97.
[33] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An investigation into the use of common libraries in Android apps. In *SANER*. 403–414.
[34] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and precise third-party library detection in Android markets. In *ICSE*. 357–376.
[35] Library 2017. *Apps with most 3rd party libraries. www.privacygrade.org/stats.*
[36] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys*. 89–103.
[37] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Detecting third-party libraries in Android apps. In *ICSE*. 641–644.
[38] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *ISSNIP*. 1–6.
[39] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–A locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop*. 7–13.
[40] Jonathan Oliver, Scott Forman, and Chun Cheng. 2014. Using randomization to attack similarity digests. In *International Conference on Applications and Techniques in Information Security*. 199–210.
[41] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. 2015. To pin or not to pin—Helping app developers bullet proof their TLS connections. In *USENIX Security*. 239–254.
[42] Parse Blog 2017. *Discovering a major security hole in Facebook's Android SDK. blog.parse.com /learn/engineering/discovering-a-major-security-hole-in-facebooks-android-sdk/.*
[43] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*.
[44] Proguard 2017. *ProGuard. developer.android.com/studio/build/shrink-code.html.*
[45] Vassil Roussev. 2009. Hashing and data fingerprinting in digital forensics. *IEEE Security & Privacy* 7 (2009), 49–55.
[46] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Richard III. 2006. md5bloom: Forensic filesystem hashing revisited. *Digital Investigation* 3 (2006), 82–90.
[47] Vassil Roussev, Golden G Richard, and Lodovico Marziale. 2007. Multi-resolution similarity hashing. *Digital Investigation* 4 (2007), 105–113.
[48] sdhash 2017. *sdhash. http://roussev.net/sdhash/sdhash.html.*
[49] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FlexDroid: Enforcing in-app privilege separation in Android. In *NDSS*.
[50] Soot Framework 2017. *Soot Analysis Framework. www.sable.mcgill.ca/soot.*
[51] ssdeep 2017. *ssdeep. https://ssdeep-project.github.io/ssdeep/index.html.*
[52] The Hacker News 2017. *Backdoor in Baidu Android SDK puts 100 million devices at risk. www.thehackernews.com/2015/11/android-malware-backdoor.html.*
[53] The Hacker News 2017. *Facebook SDK vulnerability puts millions of smartphone users' accounts at risk. www.thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html.*
[54] The Hacker News 2017. *Warning: 18,000 Android apps contains code that spy on your text messages. www.thehackernews.com/2015/10/android-apps-steal-sms.html.*
[55] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of Google Play. In *SIGMETRICS*. 221–233.
[56] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A scalable and accurate two-phase approach to Android app clone detection. In *ISSTA*. 71–82.
[57] Yan Wang and Atanas Rountev. 2017. Who changed you? Obfuscator identification for Android. In *MobileSoft*. 154–164.
[58] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*. 1329–1341.
[59] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*. 499–514.
[60] Wenbo Yang, Juanru Li, Yuanyuan Zhang, Yong Li, Junliang Shu, and Dawu Gu. 2014. APKLancet: Tumor payload diagnosis and purification for Android applications. In *ASIACCS*. 483–494.
[61] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party Android marketplaces. In *CODASPY*. 317–326.