Detecting Third-Party Libraries in Android Applications with High Precision and Recall

Yuan Zhang*§, Jiarun Dai*§, Xiaohan Zhang*§, Sirong Huang*§, Zhemin Yang*§, Min Yang*†‡§, and Hao Chen¶
*School of Computer Science, Fudan University, Shanghai, China
†Shanghai Insitute of Intelligent Electroics & Systems, Shanghai, China
‡Shanghai Institute for Advanced Communication and Data Science, Shanghai, China
§Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
¶University of California, Davis, CA, USA
{yuanxzhang, jrdai14, xh_zhang, huangsr15, yangzhemin, m_yang}@fudan.edu.cn, chen@ucdavis.edu

***Abstract*—Third-party libraries are widely used in Android applications to ease development and enhance functionalities. However, the incorporated libraries also bring new security & privacy issues to the host application, and blur the accounting between application code and library code. Under this situation, a precise and reliable library detector is highly desirable. In fact, library code may be customized by developers during integration and dead library code may be eliminated by code obfuscators during application build process. However, existing research on library detection has not gracefully handled these problems, thus facing severe limitations in practice.**

In this paper, we propose LIBPECKER, an obfuscation-resilient, highly precise and reliable library detector for Android applications. LIBPECKER adopts signature matching to give a similarity score between a given library and an application. By fully utilizing the internal class dependencies inside a library, LIBPECKER generates a strict signature for each class. To tolerate library code customization and elimination as much as possible, LIBPECKER introduces adaptive class similarity threshold and weighted class similarity score when calculating library similarity. To quantitatively evaluate the precision and the recall of LIBPECKER, we perform the first such experiment (to the best of our knowledge) with a large number of libraries and applications. Results show that LIBPECKER significantly outperforms the state-of-the-art tools in both recall and precision (91% and 98.1% respectively).

***Index Terms*—Library Detection, Code Similarity, Obfuscation Resilience**

I. INTRODUCTION

Third-party libraries are widely used by Android application (app for short) developers to build new functionalities, integrate external services, and the most important, reduce the time to release an app. App developers can find a lot of useful third-party libraries from popular open-source project hosting services (such as GitHub [14], BitBucket [6]) or online package repositories (such as Maven [19]). Besides, it is quite convenient for developers to incorporate libraries, with the built-in support of Maven/Gradle in most Android Integrated Development Environments (IDE). Sometimes, libraries need to be further tailored by developers to fit the app, e.g. adjusting the appearance of some UI widgets. In this scenario, developers usually download the source code of a library from project hosting service or online package repository and customize the source code directly.

The community of library developing, publishing, and hosting greatly eases the development of Android applications. However, every coin comes with two sides. The incorporated third-party libraries are potentially security and privacy hazards for end-users. For example, an advertisement library named Taomike SDK, which is used by more than 63,000 apps, has been reported to spy on user text messages [26]. Besides, even high-profile libraries such as Baidu SDK [4], Facebook SDK [13] and DropboxSDK [17], have been discovered with severe vulnerabilities which could be exploited by attackers to remotely control victim's device, steal sensitive information and inject malicious payload.

The above problems call into examining applications to find if they incorporated a suspicious library (such as a malicious or vulnerable SDK), which is also known as the problem of finding the provenance of a software entity [33]. To recognize third-party libraries, existing research (such as application clone detection [28], [31], [32], [35], [54], [60], [61], etc.) relies on a list of package names of popular third-party libraries and naively utilizes package name matching. Obviously, this approach is not reliable since package names of the libraries may be automatically transformed by obfuscated tools (such as ProGuard [20]) or manually modified by application developers (see our evaluations in Section IV-B). Besides, the package names of different libraries may conflict (e.g. `com.squareup.retrofit:converter-gson` [21] and `com.squareup.retrofit:converter-jackson` [22] share a same package name of `retrofit2.converter`). Thus, this simple approach can not reliably and precisely recognize third-party libraries.

Recent works on library detection adopt two approaches. The first approach (such as LibD [36] and LibRadar [40]) mines shared code features in a large number of applications and extracts similar code clusters as library instances. However, this approach requires a large set of applications as input and is limited to detect libraries that are incorporated by many applications, thus can not be used to check the presence of an arbitrary library in a single application.

To check whether a given library is included by an application, *LibScout* [27] proposed a new approach. By constructing

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

obfuscation-resilient class profiles (i.e. fingerprints) for both libraries and applications, *LibScout* utilizes high-level class organization information in profile matching to give a similarity score between a given library and an application. Although *LibScout* got some resilience against obfuscation, we found that it is still problematic when applied to large-scale library detection due to the following limitations.

Too Relaxed Class Profile. During obfuscation, all names except the references to system libraries may be refactored. To be obfuscation resilient, *LibScout* extracts relaxed class profile, which is constructed by hashing all method descriptors of the class. The relaxed class profile increases the possibility of conflicts between two distinct classes, especially when the class has few number of methods. It would not be surprising when *LibScout* reports a lot of false positives when applied to large-scale library detection.

Developer Customization. It is quite common for developers to customize a third-party library, especially when the library can not be directly applied to an application but still constructs an appropriate foundation for further improvements. For example, UI widget libraries are frequently adjusted by application developers to fit the user interface and interaction mechanism of the application. During the customization, the class profiles of the library may be modified by either removing some members or adding new members. Since *LibScout* needs to match the exact same class profile, it is hard to effectively recognize the customized libraries inside an app.

Dead Code Elimination. Library integration is so convenient that developers usually incorporate a lot of libraries into an app, while actually not all functionalities of a library are used by the app. To reduce the executable size, developers commonly use code obfuscation tools such as ProGuard [20] to safely eliminate dead library code. The code elimination may occur at both the class level and the field/method-level [18]. Similar to the effect of code customizations, code elimination can also significantly limit the effectiveness of *LibScout*.

Our work. All the above problems are quite practical and non-trivial to address any of them. This paper proposes LIBPECKER, an obfuscation-resilient, highly precise and reliable library detector for Android applications. Different from the relaxed class profile in *LibScout*, LIBPECKER exploits the class dependencies inside a library to construct strict class signatures. To increase the efficiency of class dependency comparison, LIBPECKER employs a signature generation algorithm to encode class dependencies into a signature for each class and each class member. Besides, only obfuscation-resilient information is preserved in the signature. Comparing with the relaxed class profile in *LibScout*, the proposed class dependency signature is harder to conflict.

Furthermore, to deal with code customization and code elimination, LIBPECKER applies fuzzy class matching instead of exact class matching. However, simply adopting fuzzy class matching would increase the possibility of matching two different classes. Thus, LIBPECKER further proposes adaptive class similarity threshold and weighted class similarity. The key idea is that classes with more contributions in making a library different to another should have more significant impact in library matching. With these kindly-crafted library matching techniques, LIBPECKER can effectively peck embedded libraries out of an application even when customization and dead code elimination exist. Note that this does not mean LIBPECKER can still detect libraries whose most functionalities are not present in an app. Actually, it is hard for a code-level analyzer to detect libraries when most library code has been removed or customized, without sacrificing precision. The major contribution of LIBPECKER is that it can still recognize a library even after a certain amount of code modification is applied while existing works cannot.

To quantitatively evaluate both precision and recall of LIBPECKER, this paper performs the first such experiment (to the best of our knowledge) with a large number of libraries and applications. We collected more than 10,000 libraries and 20,000 popular market applications. We compared LIBPECKER with the most relevant work *LibScout*, and the results show that the recall and precision of LIBPECKER is 91% and 98.1% respectively, while the recall and precision of *LibScout* is 69.5% and 27.7% respectively. By exploiting the internal dependencies of a library and relaxing the class matching algorithm, LIBPECKER significantly improves the state-of-the-art library detection techniques in terms of both precision and recall. Our evaluation also confirms the doubt about the reliability of using package name matching to recognize libraries.

Contributions. We make the following major contributions:

1) New idea in profiling a library by exploiting the internal dependencies inside a library which is not only critical to improve the precision of library detection, but also resilient to common obfuscation techniques.

2) New techniques in performing library matching which are more resilient to code customization and code elimination. Our key idea is to introduce fuzzy class matching instead of exact class matching, and assign different weights to each library class according to their contributions to the library. These carefully-designed techniques boost the recall and precision of library detection.

3) New results on quantitively measuring the recall and precision of library detection with a large-scale data set. The evaluation shows LIBPECKER significantly outperforms state-of-the-art tool in not only recall but also precision.

II. APPROACH

This paper considers the problem of library detection in a more realistic scenario for application developers to incorporate third-party libraries, that is code customization and code elimination can occur at both class-level and class-member-level. In an app, there is no clear boundary between app code and library code, making library detection quite challenging. Our approach utilizes class-level similarity matching to test if the main functionalities of a library are still present in an application. It takes an exact copy of a given library and an app as inputs, and calculates a similarity score (ranging from 0 to 1) between the library and the application. If the similarity

score exceeds a threshold, LIBPECKER reports the library is present in the application. Overall, our detection strategy has the following features:

1) Our approach does not rely on the names of classes, methods and fields, nor the implementation of methods, thus is resilient to common obfuscation techniques, including *API hiding* [9], [11], *control-flow randomization* [2], [7], *identifier renaming* [2], [7], [9], [11], [20], [24], *string encryption* [2], [7], [9], [11], [24]. Other offensive techniques, such as *class encryption* [9], [11], *virtualization-based protection* [59] are not considered and have already been addressed by other orthogonal research works [30], [44], [45], [48].

2) Our approach exploits the class dependencies inside libraries to improve the precision of class similarity matching. Our insight is that although class names may be changed by obfuscators, the dependencies are preserved. For example, suppose class A is a subclass of class B, when the names are obfuscated, class A' (obfuscated class name of A in the app) is still a subclass of class B' (obfuscated class name of B in the app). The dependency information among classes would help to improve the precision of recognizing two similar classes.

3) Our approach does not require two classes exactly matched. Instead, it would give a similarity score for every class pair. This feature is quite important to tolerate code customization and code elimination.

4) Our approach relies on package hierarchy to organize the process of library matching, thus we assume the package hierarchy information of a library is preserved during obfuscation. In fact, this assumption is quite weak, since obfuscators such as ProGuard do not manipulate package hierarchy by default [18]. Meanwhile, our tool is indeed evaluated to be quite effective in detecting libraries.

Approach Overview. As depicted in Figure 1, our approach consists of two major parts: a) signature generation for all library classes and application classes; b) library matching process to give a similarity score between a given library and a given application based on the class signatures. The detailed approach is divided into three steps which are described in the following. Specifically, Section II-A introduces the concept of the dependency-based class signature and presents the algorithm to generate class signatures, Section II-B describes the adaptive class matching technique which performs fuzzy class matching between a library class and an application class with adaptive class similarity threshold and unequal weights, and at last Section II-C utilizes package hierarchy information to complete the whole library matching process.

A. Dependency-based Class Signature

During library matching, if two distinct classes are matched as a pair, we label this matching as a conflict. Obviously, conflict matching would degrade the precision of library detection. As described in Section I, too relaxed class profile (such as used in *LibScout*) would cause small classes to conflict quite easily. To address this problem, we utilize class dependency information among classes to reduce the possibility of class profile conflict.

Class Dependencies. There are many kinds of class dependencies in Android apps. To be resilient to common obfuscation techniques, our approach only utilizes a reduced set of class dependencies. To define class dependencies, we designate application class ac_A, ac_B as the matched classes of library class lc_A, lc_B respectively. Specifically, we consider three kinds of class dependency.

1) *Class-inheritance Dependency.* Class inheritance relationship should be preserved during obfuscation, otherwise the type system of Java would be broken. Thus, if lc_A is a super class of lc_B, ac_A is also a super class of ac_B. Note that we do not consider interface dependencies, because code obfuscators such as ProGuard may remove interfaces of a class when performing dead code elimination.

2) *Field-in Dependency.* If lc_A has a field `f` with type of lc_B, we call lc_A is field-in dependent on lc_B. After obfuscation or customization, if field `f` is kept in ac_A, ac_A should be also field-in dependent on ac_B.

3) *Method-prototype Dependency.* If lc_A has a method `m` whose prototype contains lc_B, we call lc_A is method-prototype dependent on lc_B. We consider two kinds of method-prototype dependencies: 1) lc_B is the type of return type of `m`; 2) lc_B is the type of a parameter of `m`. After obfuscation or customization, if method `m` is kept in ac_A, ac_A should be also method-prototype dependent on ac_B.

Class Dependency Signature Generation. Class dependencies connect separate classes into a graph. However, directly utilizing class dependency graph in library matching is quite inefficient, due to the computational complexity of graph matching. To increase the efficiency, we devise an algorithm to encode class dependency graph into signatures.

Specifically, for each class dependency, LIBPECKER generates a dedicated *class dependency signature (CDS)* to encode the dependency information. For example, if class A depends on class B, the dependency signature is calculated by hashing a composed string from the following three elements.

- *Class Type of B*. Specifically, we consider three types of class B: system classes, classes in the same package, classes in other packages. Note that primitive types are treated as system classes in our approach. We use an integer to represent each type in the signature.
- *Array Dimension of B*. If class B is not an array class, the dimension is 0.
- *Class Name of B*. If class B is a non-system class, its name may be obfuscated, thus we could not use its name in the dependency signature. Instead, we use a constant value (such as `X`) to represent the class name in the signature. If B is a system class, its names would not be obfuscated and can be directly used in the signature.

Note that we only encode direct class dependencies into the class dependency signature. For example, if class A depends on class B and class B depends on class C, in the class dependency signature of class A, we only consider its dependencies on class B. In fact, the dependency information between class B and class C is encoded in the class signature of class B.

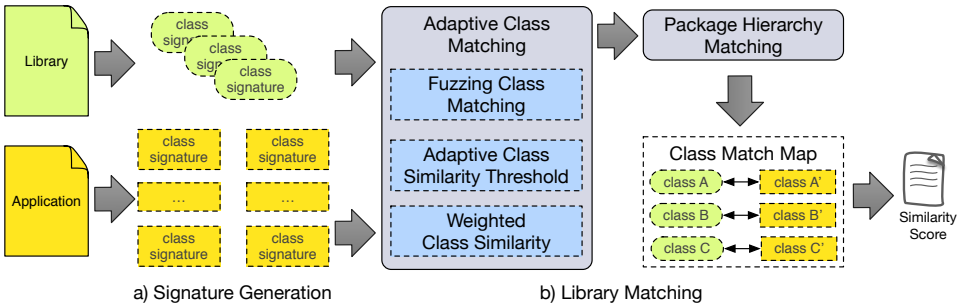

Fig. 1. Approach Overview. There are three steps in our approach. First, we generate *dependency-based class signatures* (see Section II-A) for all classes. Second, we introduce *adaptive class matching* (see Section II-B) to score similarity of two classes. At last, we perform *package hierarchy matching* (see Section II-C) to organize the whole library matching process.

Class Signature Generation. In LIBPECKER, each class is assigned with a signature, which includes all the dependency information about the class. Figure 2 depicts the class signature generation process. *Class signature* is calculated from a *basic signature* and all its member signatures (*field signature*s and *method signature*s). During signature generation, we remove all the `bridge` and `synthetic` methods that are generated by compiler, and sort basic signatures, method signatures and field signatures to compose them iteratively into a string. Finally, class signature is generated as the hash value of the composed string. The generation algorithms for *basic signature*, *field signature* and *method signature* are described as follows.

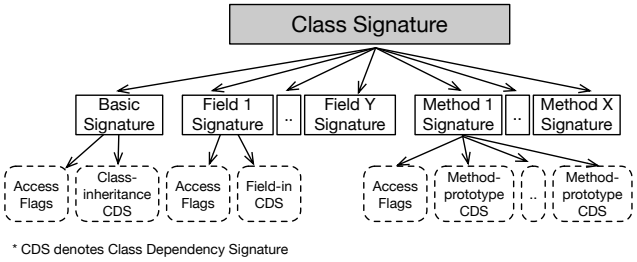

Fig. 2. Workflow of Class Signature Generation.

Basic Signature. It represents basic class information, including access flags of the class and the *class-inheritance dependency* signature. Since the access flags of a class/field/method may be manipulated during obfuscation, LIBPECKER only considers several flags that are required by Java runtime (thus will not be manipulated by obfuscators) such as STATIC flag.

Field Signature. It is calculated as a hash of the access flags of the field and the *field-in class dependency* signature.

Method Signature. It is calculated as a hash string of the access flags of the method and all the *method-prototype class dependency* signatures.

If two classes have the same signature, they are treated as the same class. However, during code customization or code elimination, some class members may be removed or modified, leading to different class signatures. To handle this problem, the next section will describe our adaptive class matching algorithm in detail.

B. Adaptive Class Matching

LIBPECKER considers a more realistic library integration scenario, where libraries may be customized and dead library code may be eliminated by obfuscators. If a library detector does not consider this scenario, it is hard to recall as many embedded libraries as possible. To address this problem, LIBPECKER proposes fuzzy class matching, which does not give a binary class matching indicator, that is either same or not same. Instead, LIBPECKER gives a similarity score between two classes.

Fuzzy Class Similarity. Fuzzy class matching does not require two classes to be exactly the same. When two classes have unequal signatures, fuzzy class matching compares their basic signatures. If they still have different basic signatures, they are marked as different classes. In this case, a similarity score of 0 is reported. When they share the same basic signature, LIBPECKER further examines their class member signatures (including method signatures and field signatures).

During code customization and code elimination, new class members may be added to a library class and existing class members of a library class may also be removed or modified. The similarity score between a library class and an application class reflects to what degree are class members in the library class preserved in the application class. Thus, we use Jaccard similarity to define class similarity.

$$sim\_clz(lc, ac) = \frac{\{lc\ member\ signatures\} \cap \{ac\ member\ signatures\}}{\{lc\ member\ signatures\} \cup \{ac\ member\ signatures\}}$$

Adaptive Class Similarity Threshold. Fuzzy class matching helps to tolerate code customization and code elimination, but it also causes two distinct classes having a non-zero similarity score. To clearly differentiate similar classes from distinct classes, LIBPECKER sets a similarity threshold, and assigns zero similarity score to two classes whose similarity is below the threshold.

Instead of a fixed similarity threshold, LIBPECKER designs adaptive class similarity threshold, that is to set varying class similarity threshold for each class according to its class

member count. The insight is that classes with more members should be in favor, because they support more functionalities of a library, while a fixed class similarity threshold may bias classes with fewer members. Thus, higher threshold should be given to classes with fewer class members. However, it is hard to accurately figure out the relationship between the count of class members and the threshold. Details on how we set the threshold function are described in Section IV.

Weighted Class Similarity. When calculating the library similarity, existing works on library detection mainly assign equal weight to each class in a library. Actually, classes with fewer members are easier to match than those have many members. Thus, in the scenario of equal class weight, libraries mainly consist of small classes would easy to match with applications which actually do no contain them, leading to high false positive rate.

We believe that classes in a library have unequal contributions to the library in making a library differ to another. Thus, more weights need to be given to classes which make a library more significantly differ to another. Intuitively, classes with more methods should have heavier weights in library matching, because functionalities of a library is implemented in methods. Similarly, classes with more dependencies also need more weights because they reflect the internal connectivity of a library which is a significant feature to differ libraries. Specifically, for a library class `lc`, its weights is calculated with the following function. Note that function `bbCount` counts the amount of basic blocks in a method and function `depClassCount` counts how many classes a class depends.

$$weight(lc) = bbCount(\textit{all methods in } lc) + depClassCount(lc)$$

Class weights are pre-computed for the libraries before library detection, and the weighted similarities reflect the significance of classes in differentiating libraries. In the next section, we will detail our library matching algorithm by combining class similarities and class weights.

C. Package Hierarchy Matching

A simple way to calculate the library similarity with an application is to sum up the maximum similarity score of every library class with the application. However, two library classes may be matched to a same application class at their maximum similarity scores. To achieve global maximum similarity, we need to choose the matched class pair which leads to a higher global similarity score. Considering there are a lot of classes in an application, it is quite inefficient to enumerate all possible class-level matching pairs to calculate global maximum similarity. Instead, we adopt package-level matching to calculate library similarity with a given application. Specifically, LIBPECKER transforms the problem of similarity calculation between a given library and an application to the problem of how to map library packages to application packages with global maximum similarity. The problem is solved in the following four steps.

1) We calculate the similarity scores between all library packages and all application packages;

2) For each library package, we select application package candidates whose similarity scores exceed a threshold;

3) We calculate the optimal library similarity score by applying a greedy algorithm to map each library package with its max-similarity application package. If the optimal library similarity score does not exceed the threshold, the whole library matching process quits immediately and a zero similarity score is returned.

4) Finally, we start the package enumeration process to achieve the maximum library similarity score. We prefer to map a library package to an application package candidate with the same name. For library packages with no same-name application package candidate, we enumerate all possible candidates under the guidance of package hierarchy, and select the package mapping which maximizes the similarity for whole library. The detailed enumeration process will be explained later.

Package Similarity Score. The similarity score between a library package and an application package is calculated from the similarity scores of classes in the library package. Considering that classes in a library have different weights, we sort all library classes in descending weights and map library classes with the highest weight first. To map a library class, we calculate the similarity scores between it and every unmapped class in the application package and choose the application class with the highest similarity score. After class-level mapping between two packages, the similarity score between library package (`lp`) and application package (`ap`) is calculated using the following function, where lc_i is the i-th class in `lp`, and function `map(`lc_i`, ap)` returns the mapped class for lc_i in `ap`.

$$sim\_pkg(lp, ap) = \frac{\sum_{lc_i \in lp}^{n} sim\_clz(lc_i,\ map(lc_i,\ ap)) * weight(lc_i)}{\sum_{lc_i \in lp}^{n} weight(lc_i)}$$

Library Similarity Score. Based on the class mapping constructed during package matching, the overall similarity between a given library (`lib`) and an application (`app`) is calculated as follows.

$$sim(lib, app) = \frac{\sum_{lc_i \in lib}^{n} sim\_clz(lc_i,\ map(lc_i,\ app)) * weight(lc_i)}{\sum_{lc_i \in lib}^{n} weight(lc_i)}$$

Package Hierarchy Guided Enumeration. For library packages with more than one application package candidate, we need to enumerate all possible package mappings between the library package and the application package candidates. For each enumerated package mapping, we calculate the overall library similarity score, and select the mapping with the highest library similarity score.

During enumeration, we utilize package hierarchy information in the library to prune inappropriate package mappings. For example, if a library package `okhttp3.internal.connection` has already been mapped to an application package `okhttp3.b.c`, we should not further allow mapping from library package `okhttp3.internal` to application package `okhttp3.d`.

Specifically, for any two library packages lp_1 and lp_2, we denote their mapped application packages as ap_1 and ap_2 respectively. We apply Algorithm 1 to check package hierarchy compliance, and discard violated package mappings.

Algorithm 1 Package Hierarchy Check Algorithm

Input: lp_1, lp_2, ap_1, ap_2: Package Name
Output: violated: boolean

```
1: //Step 1: test parent relationship
2: if lp1 is parent of lp2 & ap1 is not parent of ap2 then
3:     return false
4: end if
5:
6: //Step 2: test child relationship
7: if lp1 is child of lp2 & ap1 is not child of ap2 then
8:     return false
9: end if
10:
11: //Step 3: test sibling relationship
12: if lp1 is sibling with lp2 & ap1 is not sibling with ap2 then
13:     return false
14: end if
15:
16: //Step 4: test ancestor inheritance relationship
17: //common_pkg() returns longest common parent package
18: lp_common = common_pkg(lp1, lp2)
19: ap_common = common_pkg(ap1, ap2)
20: //distance() returns the delta-depth of two packages
21: if distance(lp_common, lp1) != distance(ap_common, ap1) then
22:     return false
23: end if
24: if distance(lp_common, lp2) != distance(ap_common, ap2) then
25:     return false
26: end if
27:
28: // pass all tests, no violation detected
29: return true
```

III. Data Set

To evaluate the effectiveness of LibPecker, we need to collect a large number of libraries and applications. This section describes how we setup a database of third-party libraries and applications, and presents some basic statistics of our data set.

A. Library Collection

LibPecker performs code-level similarity analysis to detect third-party libraries in applications, thus it requires the code of the library to perform analysis. Usually, application developers use Maven/Gradle to incorporate Maven libraries. Thus, we decide to base our data set of third-party libraries on top of Maven repositories. However, since there are quite a large volume of libraries in Maven repositories, and many of them only target Java applications (such as JavaEE applications), we need to get a list of libraries that are indeed used in Android applications.

We use two ways to collect a Android library list. First, we write a crawler to download all applications from F-Droid [12] (an open-source application market for Android) with source code. We parse all the configuration files in these application projects to find the GAV (groupName, artifactId, and version) pairs of the third-party libraries used in these projects. Note that in Maven/Gradle, a GAV pair can uniquely identify a library. Second, we find two webpages (Top 100 Android Libraries [15] and Awesome Android [3]) which collect top popular libraries in the Android developer community. Similarly, we manually collect the groupName and artifactId for each library in these pages.

For libraries on the list, to increase the possibility of matching them with an application, we need to collect as many versions of them as possible. Thus, we ignore the version value in a GAV pair and use a crawler to list all the versions of a library in the Maven repository. For each version of a library, our crawler fetches the JAR file from the Maven Central Repository. In all, we successfully collect 758 libraries with 11,620 versions.

Library Filtering. In the library database, we found that some libraries are mainly incorporated by developers for their resource files (such as layout files, icon files) to ease UI development, and contain very few code in their JAR files. This kind of library is difficult for code-level analyzers (such as LibPecker, *LibScout*) to detect. A better way to detect this kind of libraries may need to utilize UI features as proposed by Fangfang et al. [56]. Some other libraries mainly provide annotation classes which may be removed after application build process, thus this kind of libraries are out of scope for our tool to detect. Thus, we filter out those libraries that have no more than 2 methods per class and with no more than 1 basic block per method. Finally, we remove 32 libraries from the database, that is 4.2% of the whole data set. After library filtering, there are still 726 libraries in the data set with 11,017 versions in all. On average, there are 16 versions for each library. Since Android applications are assembled in DEX code (which is the binary format of Android applications and libraries) [8], we write a script to translate all downloaded JAR files into DEX files.

B. Application Collection

To construct a representative application data set, we write crawlers to collect top popular applications from Google Play market and three third-party application markets in China. As Table I shows, we totally collect 20,996 applications from 4 representative application markets. After removing redundant applications with same hash value, there are still 20,315 applications in our data set.

TABLE I
Popular Applications Collected from Google Play and Mainstream Third-party Application Markets in China.

Market Name	# of Top Apps	Date
Google Play [16]	8,399	2017/02/15
Baidu Market [5]	2,434	2017/03/22
360 Market [1]	6,421	2017/03/22
Tencent Market [25]	3,742	2017/03/22

IV. Evaluation

This section evaluates the performance of LibPecker from both the precision and the recall. Our experiments are based upon more than 10,000 libraries and more than 20,000 applications, which is a quite large data set for the evaluation. To the best of our knowledge, our paper is the first to give a quantitive measurement against the precision and recall of library detectors.

Implementation. LibPecker is implemented within about 8K lines of Java code. LibPecker utilizes dexlib2 [10] to parse DEX files, extracts the inheritance and dependency information, and implements a simple control-flow-graph (CFG) analysis module to count basic blocks of a method. Since *LibScout* is the state-of-the-art and the only tool we know for detecting the existence of a reference third-party library, we choose *LibScout* to make comparison with LibPecker. Recently, the authors of *LibScout* have released its source code [23]. In our evaluation, *LibScout* is built from its commit on Sept 4, 2017 (commit hash: 51696c55).

Setting Thresholds. Our approach needs to define several thresholds. For package similarity threshold and library similarity threshold, we use the same setting as *LibScout*, which are 0.5 and 0.6 respectively. To set the adaptive class similarity threshold, we select 25 true positive application-library pairs and 25 true negative application-library pairs from our data sets with manual analysis, and adjust the threshold to maximize the performance of LibPecker on these pairs. Note that these cases are carefully selected to not overlap with the cases used in the evaluation. Specifically, we set threshold for classes with less than 5 members to 1, for classes with members between 5 and 10 to 0.9, for classes with members between 10 and 15 to 0.8, for classes with members between 15 and 20 to 0.7, for classes with members between 20 and 25 to 0.6, and for classes with more than 25 members to 0.5. Note that we do not claim the thresholds are our contributions. Certainly, there are better ways to determine these thresholds which need further investigation and study (see Section V).

A. Recall

First, we want to measure the ability of LibPecker in pecking out libraries from applications. To answer this question, we need to construct ground truth of true positive library-application pairs (i.e. a library is indeed included by an application). Obviously, it is not feasible to manually check existence of a library in an application for the whole data set. Meanwhile, it may be unconvincing to limit the scale of the experiment to few applications and libraries.

Ground Truth Selection. Thus, we need a way to automatically construct a large base of ground truth. Our insight is that not all package names are obfuscated during application build process. However, it is unreliable to simply utilize package name patching for ground truth construction. Hence, we propose two rules to refine package name matching process to select true positive library-application pairs.

- *Rule 1.* If package names of a library are all covered by another library, we can not reliably determine whether this library is present by simply checking package names. Thus, we need to exclude this library from ground truth.
- *Rule 2.* Only when an application contains all the package names of a library, we flag this application-library pair as true positive.

By applying the above rules in our data set, we successfully constructed a ground truth base of 33,964 application-library pairs, covering 9,834 applications and 310 libraries. On average, each application includes 3.45 libraries. Since our ground truth is constructed by selecting only true positive pairs, both tools would have no false positive in this experiment. Recall is calculated as follows.

$$recall = \frac{\#\ of\ detected\ pairs}{\#\ of\ pairs\ in\ ground\ truth}$$

Table II presents the recall for each tool. From Table II we can find that although LibPecker places more constraints (such as checking class dependencies) on matching classes, the recall of LibPecker is significantly better than *LibScout*. The better recall is mainly attributed to the adaptive class matching adopted by LibPecker which better tolerates code customization and code elimination.

TABLE II
Recall of LibPecker and *LibScout*.

Tool	Detected Pairs	Ground Truth	Recall
LibScout	23,598	33,964	69.5%
LibPecker	30,924	33,964	91.0%

False Negative Breakdown. Since LibPecker fails to detect 3,040 application-library pairs in the ground truth, we randomly select 50 pairs and perform manual examination to scrutinize why LibPecker does not flag them as positive pairs. Finally, we find 4 kinds of false negatives: 1) for 19 pairs, we find the libraries are dramatically modified by developers to the extent that most class signatures generated by LibPecker are broken; 2) for 21 pairs, we find very few functions of the libraries are used by developers, thus most code in these libraries are useless and aggressively eliminated by code obfuscators such as ProGuard; 3) for 7 pairs, we also find no matching library in our database for the application with same package name, this may be caused by package name conflicts between the library and the application or the missing of a special version of the library in our database; 4) for 3 pairs, several classes in the library are moved to other packages in the application, thus LibPecker can not effectively recognize these libraries. We argue that false negatives are quite acceptable for LibPecker, especially when libraries are modified beyond recognition. There is always a balance between false negative and false positive. Although bearing some false negatives, LibPecker still successfully achieves a high recall (91%) and notably proceeds existing tools.

B. Precision

Since we could not manually validate all the detection results in our data set, we randomly select 12,000 application-library pairs to evaluate the precision of LIBPECKER. Precision is calculated using the following function.

$$precision = \frac{\#\ of\ true\ positive\ pairs}{\#\ of\ detected\ pairs}$$

Overall Result. In all, *LibScout* reported 365 positive application-library pairs, while LIBPECKER reported 155 positive pairs. By manually checking the correctness of these pairs, the precisions for both tools are presented in Table III. We can find that the precision of LIBPECKER is also significantly better than that of *LibScout*.

TABLE III
PRECISON OF LIBPECKER AND *LibScout*.

Tool	Detected Pairs	True Positive	Precision
LibScout	365	101	27.7%
LibPecker	155	152	98.1%

Figure 3 depicts the coverage of the reported positive results from LIBPECKER and *LibScout*. As this figure shows, there are 96 true positives detected by both tools, while LIBPECKER detected 56 new true positives. Besides, from the perspective of false positive, most false positives incurred by *LibScout* are excluded by LIBPECKER. Overall, LIBPECKER achieves quite high precision in detecting third-party libraries. We further examined the results to answer the following questions.

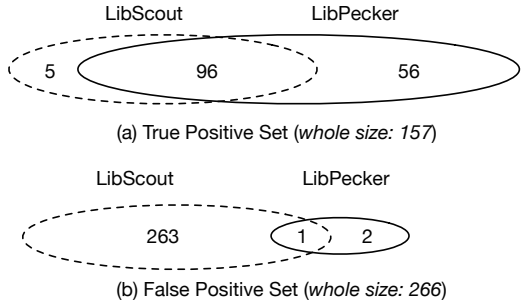

Fig. 3. All the true positives and false positives in LIBPECKER and *LibScout*.

RQ1: Why do *LibScout* have so many false positives? After manually checking the 263 false positives only reported by *LibScout*, we found most false positive pairs are related to small libraries which are mostly composed of classes with few methods. Since *LibScout* assigns equal weight to every library class in calculating whole library similarity, libraries with many small classes are quite easy to be mistakenly flagged as positive cases. However, this problem has quite limited impact on LIBPECKER, owing to its adaptive weighted class similarity design in calculating library similarity.

RQ2: Why do LIBPECKER miss 5 true positives that are detected by *LibScout*? We found that for the 5 true positives, libraries have been heavily modified to the extent that LIBPECKER can not recognize them. An interesting finding is that although *LibScout* has successfully reported the 5 pairs, it mistakenly matches the library classes with incorrect application classes due to the conflict of class profiles.

RQ3: How about using package name matching in detecting libraries? For the 152 true positive pairs detected by LIBPECKER, we wrote a script to test how many pairs can be recognized by simple package name matching. Finally, we found only 63 application-library pairs can be recognized by simple package name matching. This result confirms the doubt about the reliability of using a list of library package names to recognize third-party libraries. Obviously, our work would benefit existing works [28], [34], [52], [53], [55] which rely on simple package name matching to recognize third-party libraries.

C. Library Prevalence

We also measured the prevalence of third-party libraries in our data set. On average, our tool detects 8.6 distinct libraries per app. The app `com.wallstreetcn.news` (hash:18b4224a3da6f668c74a45d5fe456936), a news app from Tencent Market is recognized to contain the most libraries (59). For each recognized library, we also counted how many applications incorporate this library. We found that libraries with basic functionalities (excluding Android Support Libraries) have been recognized in most applications, such as `com.jakewharton:disklrucache` (for image cache management), `com.google.code.gson:gson` (for JSON support in Java), etc. There are also some powerful libraries detected in quite few applications, because these libraries are mostly incorporated to implement very specific functionalities, such as `jcifs:jcifs` (a Java implementation of CIFS/SMB networking protocol), `org:jaudiotagger` (for editing tag information in audio files), `org.apache.commons:commons-csv` (for manipulating CSV files), etc.

V. DISCUSSION

To clarify main contributions we want to claim in this paper, we discuss the following topics.

Comparison with *LibScout*. LIBPECKER detects third-party libraries by checking the the presence of reference libraries in a given application. This workflow is similar to *LibScout* which is the most relevant work. However, LIBPECKER differs from *LibScout* significantly in its detection techniques. First, our tool fully exploits the internal class dependencies to fingerprint a library, while *LibScout* failed to take advantage of this dependency. Second, to calculate class similarity, LIBPECKER employs partial class matching which better tolerates the common practice of code customization and code elimination during library incorporation, while *LibScout* requires whole class matching. Third, since different classes of a library contribute unequally to the library, our tool assigns unequal weights to different classes and set adaptive class similarity thresholds in calculating library similarity. These novel techniques are quite important for effectively

detecting libraries with many small classes. As evidenced by our evaluation, *LibScout* incurs most of its false positives in detecting small libraries. Finally, both LIBPECKER and *LibScout* adopt package-level matching to organize library matching process and rely on package hierarchy information to exclude inappropriate package matching. However, different to *LibScout* which only utilizes three kinds of package relationships (i.e. parent, child, sibling), LIBPECKER also considers a general ancestor inheritance relationship (see Algorithm 1 in Section II-C). This design helps to improve the precision of package matching for libraries with many packages.

Limitations. Although LIBPECKER does not look into the concrete code implementations, it still relies on some code features (e.g. class/package structures) of the library and application to give a similarity score. Thus, for libraries without significant code features, such as libraries defining annotations and libraries providing UI resources, LIBPECKER can not effectively recognize them. Actually, these kinds of libraries are hard to detect for all existing library detectors. Perhaps combining other kind of features in these libraries may help. For example, we can learn from ViewDroid [56] to utilize UI similarities to detect libraries providing UI resources.

Besides, although LIBPECKER can better tolerate code customization and code elimination than *LibScout*, it is unable to detect libraries that are dramatically modified beyond recognition. In fact, if most code of a library is modified during integration, it has turned into a new library with quite different logic and behavior, thus it is quite acceptable for not recognizing this group of code as the original library.

To efficiently calculate library similarity, LIBPECKER chooses to utilize package hierarchy information in organizing the class matching process. This choice is made upon the assumption that package structure of a library is not broken during the application build process. However, this assumption sometimes does not hold. Fortunately, this issue is not difficult to solve. In this situation, LIBPECKER can calculate the similarity between a library and an application by finding the pairs for each library class in the whole scope of the application with the global maximum library similarity. Besides, our ground truth selection has a slight bias against data points without package hierarchy obfuscation. Actually, it is hard to construct a large data set without significant human efforts, and to the best of our knowledge, our paper uses the largest data set in evaluating recall. Considering obfuscators such as ProGuard do not manipulate package hierarchy by default [18], we believe this selection strategy is acceptable in practice.

Further Improvements. Current design and implementation of LIBPECKER can be further improved, at least in the following two aspects. First, several thresholds (in Section IV) are defined to set adaptive threshold for partial class matching. These thresholds are proposed to tolerate code customization and code elimination; however, currently they are made according to manual experience. In our future work, we plan to try machine learning techniques to optimize the determination of these thresholds with respects to unique characteristics of each library. Nevertheless, as the evaluation shows, these parameters have already helped LIBPECKER to achieve quite good performance in both precision and recall.

Second, according to the usage scenario of LIBPECKER where it needs to give a similarity score between a given library and an application, LIBPECKER naturally adopts pairwise comparison strategy. To improve the efficiency of library detection, further innovations are needed. One possible way is to split the whole library data set into small clusters, and use code clustering to filter out irrelevant libraries before pairwise comparison. However, this paper concentrates on improving the precision and recall of existing library detection techniques, while leaving the scalability issue as a orthogonal fundamental problem to solve in the future.

VI. RELATED WORK

We present the most relevant works to LIBPECKER in the following three areas.

Library Detection. Library detection plays a very important role in library sandboxing, clone detection [37] and other related works. However, simply using a whitelist of popular libraries [28], [34], [52], [53] is not reliable and scalable to detect third-party libraries, as names may be obfuscated and new libraries keep emerging. There are mainly two approaches in automatically detecting libraries.

One approach is to detect incorporated libraries from apps without a priori knowledge about the libraries. A common idea for this line of research is to divide an app into several code modules and identify similar modules in different apps as library candidates. AdDetect [42] uses package hierarchy clustering to split an app into primary modules and non-primary modules. PEDAL [39] extracts code-features and package relationship information to train a classifier to detect libraries. AnDarwin [31] and WuKong [54] also use clustering techniques to efficiently filter library code. LibRadar [40] extends the clustering technique in WuKong to identify possible libraries from analyzing one million apps, and uses hashing-based profile to detect similar libraries. The above works rely on package name and package structure to detect and classify libraries, while package names may be obfuscated and package structure may vary among different versions of the same library. A recent work, LibD [36] advances this line of research by using a way to extract code modules based on the reference and inheritance relationships among classes and methods.

The first approach is suitable to detect libraries which are extensively used by many apps, but it can not pinpoint an exact version of the library and even can not know the identity of the detected library without human assistance. Differently, the second approach detects libraries by comparing the similarity between a reference library and an application. The strength of this approach is that it is also capable of detecting unpopular libraries and enables further investigation of the detected library, e.g. studying the library customization practice by differing with the reference library. LibScout [27] is the first library detector adopting this approach. It does not analyze concrete

names nor the implementation of a library, but extracts profiles from the class interfaces and organizations, thus is more resilient to obfuscation techniques than previous approaches. However, through a deep study of library integration practice, we find LibScout is limited in its adaptability to library code customization and elimination. Considering these problems, this paper proposes LIBPECKER which employs several novel techniques and significantly improves the effectiveness of *LibScout* from both precision and the recall (as evidenced by the evaluation results in Section IV).

Clone Detection. Malicious Android applications mainly repackage legitimate apps to infect users [62]. Thus, repackaged app clone detection is quite important for Android malware analysis. Since Android apps are mostly released in binary format, existing work [38], [41] based on source code-level similarity analysis is not applicable. DroidMOSS [61] and Juxtapp [35] apply fuzzy code hashing to localize and detect the modifications repackagers placed over the original apps. To accurately compare application behaviors, DNADroid [32] constructs program dependency graphs for applications, and uses isomorphic subgraph matching to detect cloned apps.

To avoid pairwise comparison, more scalable techniques have also been studied. AnDarwin [31] extracts semantic vectors from program dependency graphs, and leverages hashing-based approximate near-neighbor finding algorithm to scale the detection of app clones. PiggyApp [60] locates primary modules in an app and map semantic features from the primary modules into a metric space, thus a linear search algorithm can be used to detect piggybacked apps. WuKong [54] adopts two-phase detection strategy which first uses API calls as semantic features to find suspicious similar apps and then applies a fine-grained analyzer to check application clones. Chen et al. [28] introduce the definition of centroid on top of method control-flow-graph, and utilize method centroid distance to efficiently compare application similarity.

Since repackaged apps share most UI interfaces with their original apps, UI similarity analysis is also shown to be very effective in detecting similar apps [29], [47], [50], [56].

Although both library detection and application clone detection rely on similarity analysis, they meet quite different technique challenges. For clone detectors, they assume app clones share most similar features and base their analysis upon this assumption. However, a single third-party library usually occupies a very small part of an app, thus library detectors need invent new techniques to precisely recognize embedded libraries from applications. Meanwhile, library detectors can improve the effectiveness of existing app clone detectors by automatically excluding library code from app code in similarity comparison.

Library Sandbox. Since incorporated third-party libraries may bring new security and privacy issues, researchers have proposed many sandbox techniques to isolate untrusted libraries. Ad libraries are the main targets to isolate. AdDroid [43] puts ad libraries into a separate system service with constrained permissions. AdSplit [49] and AFrame [57] both rely on a dedicated process to isolate third-party ad libraries.

Besides, in-process library separation is proposed for general third-party library isolation. Compac [55] tracks third-party library code from Java call stack at the VM-level and enforces policy checks against third-party library code. FlexDroid [46] leverages ARM memory domain to provide JNI (Java Native Interface) sandboxing, so that both native libraries and dynamic loaded code are under constrained. FineDroid [58] supports system-wide execution context tracking to control permission usage of third-party libraries.

The above frameworks all require system modifications, thus are hard to deploy. In light of this limitation, NativeGuard [51] and CASE [63] propose user-level library sandboxes.

To recognize third-party libraries in apps, most of these works either use white lists of libraries or require developers to manually declare the exact library components. LIBPECKER can benefit these works by automatically checking the presence of libraries and zoning the code belong to libraries in an application.

VII. CONCLUSION

Library integration is a common practice in developing Android applications, and with no doubt more and more libraries would be used by developers. In this trend, library detection becomes a quite fundamental and important basis for analyzing and securing Android applications. However, existing research does not gracefully handle the cases of code customization and code elimination during library integration which are actually quite common in reality. Considering these problems, this paper proposes LIBPECKER, an obfuscation-resilient, highly precise and reliable library detector for Android applications. The key insight is that existing library profile algorithm does not fully utilize a very important yet also obfuscation-resilient feature, that is the class dependencies inside a library. Class dependencies connect separate classes into a library and apparently represent a significant feature for a library. During library matching, LIBPECKER generates class-dependency-based signatures for both library classes and application classes. Furthermore, to tolerate code customization and code elimination, LIBPECKER employs fuzzy class matching and assigns different weights to each library class according to their contributions to the whole library. To evaluate the performance of LIBPECKER, we perform several quantitive experiments with a large data set. The results show that our tool significantly outperforms the state-of-the-art tool (*LibScout* [27]) from both the precision and the recall.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Program on Key Basic Research (NO. 2015CB358800), the National Natural Science Foundation of China (61602123, U1636204, 61602121, U1736208). The work of Yuan Zhang was also supported in part by the Shanghai Sailing Program under Grant 16YF1400800.

REFERENCES

[1] "360 app market," accessed: 2017-03-22. [Online]. Available: http://zhushou.360.cn/

[2] "Allatori java obfuscator," accessed: 2017-10-10. [Online]. Available: http://www.allatori.com

[3] "Awesome android," accessed: 2017-10-10. [Online]. Available: https://snowdream.github.io/awesome-android/

[4] "Backdoor in baidu android sdk puts 100 million devices at risk," accessed: 2017-10-10. [Online]. Available: http://thehackernews.com/2015/11/android-malware-backdoor.html

[5] "Baidu app market," accessed: 2017-03-22. [Online]. Available: http://shouji.baidu.com/

[6] "Bitbucket," accessed: 2017-10-10. [Online]. Available: https://bitbucket.org/

[7] "Dasho java obfuscator," accessed: 2017-10-10. [Online]. Available: https://www.preemptive.com/products/dasho/overview

[8] "Dex bytecode format," accessed: 2017-10-10. [Online]. Available: https://source.android.com/devices/tech/dalvik/dex-format

[9] "Dexguard android obfuscator," accessed: 2017-10-10. [Online]. Available: https://www.guardsquare.com/en/dexguard

[10] "Dexlib2 in smali/baksmali," accessed: 2017-10-10. [Online]. Available: https://github.com/JesusFreke/smali/tree/master/dexlib2

[11] "Dexprotector android obfuscator," accessed: 2017-10-10. [Online]. Available: https://dexprotector.com

[12] "F-droid: Free and open source android app repository," accessed: 2017-10-10. [Online]. Available: https://f-droid.org/

[13] "Facebook sdk vulnerability puts millions of smartphone users' accounts at risk," accessed: 2017-10-10. [Online]. Available: http://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html

[14] "Github," accessed: 2017-10-10. [Online]. Available: https://github.com/

[15] "Github android libraries top 100," accessed: 2017-10-10. [Online]. Available: https://github.com/Freelander/Android_Data/blob/master/Android-Librarys-Top-100.md

[16] "Google play app market," accessed: 2017-02-15. [Online]. Available: https://play.google.com/store

[17] "Ibm discloses vulnerability in dropbox's android sdk," accessed: 2017-10-10. [Online]. Available: http://www.infoworld.com/article/2895016/mobile-technology/ibm-discloses-droppedin-vulnerability-for-dropboxs-android-sdk.html

[18] "Introduction to proguard," accessed: 2017-10-10. [Online]. Available: https://www.guardsquare.com/en/proguard/manual/introduction

[19] "Maven central repository," accessed: 2017-10-10. [Online]. Available: http://search.maven.org/

[20] "Proguard java obfuscator," accessed: 2017-10-10. [Online]. Available: http://proguard.sourceforge.net/

[21] "Retrofit gson converter," accessed: 2017-10-10. [Online]. Available: https://mvnrepository.com/artifact/com.squareup.retrofit2/converter-gson

[22] "Retrofit jackson converter," accessed: 2017-10-10. [Online]. Available: https://mvnrepository.com/artifact/com.squareup.retrofit2/converter-jackson

[23] "Source code for libscout," accessed: 2017-10-10. [Online]. Available: https://github.com/reddr/LibScout.git

[24] "Stringer java obfuscator," accessed: 2017-10-10. [Online]. Available: https://jfxstore.com/stringer

[25] "Tencent app market," accessed: 2017-03-22. [Online]. Available: http://sj.qq.com/

[26] "Warning: 18,000 android apps contains code that spy on your text messages," accessed: 2017-10-10. [Online]. Available: http://thehackernews.com/2015/10/android-apps-steal-sms.html

[27] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 356–367. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978333

[28] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[29] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 659–674. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831185

[30] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 275–284. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046739

[31] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of android application clones based on semantics," *IEEE Transactions on Mobile Computing*, vol. 14, no. 10, pp. 2007–2019, Oct 2015.

[32] ——, "Attack of the clones: Detecting cloned applications on android markets," in *Proceedings of 17th European Symposium on Research in Computer Security (ESORICS)*, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33167-1_3

[33] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 183–192. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985468

[34] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 101–112. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185464

[35] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'12. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 62–81. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37300-8_4

[36] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 335–346. [Online]. Available: https://doi.org/10.1109/ICSE.2017.38

[37] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 242–251. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597109

[38] M. Linares-Vsquez, A. Holtzhauer, and D. Poshyvanyk, "On automatically detecting similar android apps," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[39] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '15. New York, NY, USA: ACM, 2015, pp. 89–103. [Online]. Available: http://doi.acm.org/10.1145/2742647.2742668

[40] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 653–656. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889178

[41] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 364–374. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337267

[42] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, April 2014, pp. 1–6.

[43] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12. New York, NY, USA: ACM, 2012, pp. 71–72. [Online]. Available: http://doi.acm.org/10.1145/2414456.2414498

[44] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2016. [Online]. Available: http://www.bodden.de/pubs/ssme16harvesting.pdf

[45] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855876.1855877

[46] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin, "FlexDroid: Enforcing In-App Privilege Separation in Android," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[47] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 56–65. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664275

[48] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 94–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.27

[49] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362821

[50] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 163–173. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820305

[51] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, ser. WiSec '14. New York, NY, USA: ACM, 2014, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/2627393.2627396

[52] A. Tongaonkar, S. Dai, A. Nucci, and D. Song, "Understanding mobile app usage patterns using in-app advertisements," in *Proceedings of the 14th International Conference on Passive and Active Measurement*, ser. PAM'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 63–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36516-4_7

[53] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 221–233. [Online]. Available: http://doi.acm.org/10.1145/2591971.2592003

[54] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771795

[55] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: ACM, 2014, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557560

[56] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks*, ser. WiSec '14. New York, NY, USA: ACM, 2014, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2627393.2627395

[57] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: ACM, 2013, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/2523649.2523652

[58] Y. Zhang, M. Yang, G. Gu, and H. Chen, "Finedroid: Enforcing permissions with system-wide application execution context," in *Proceedings of International Conference on Security and Privacy in Communication Networks (SECURECOMM)*, Oct. 2015.

[59] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "Divilar: Diversifying intermediate language for anti-repackaging on android platform," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: ACM, 2014, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557558

[60] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 185–196. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435377

[61] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[62] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[63] S. Zhu, L. Lu, and K. Singh, "Case: Comprehensive application security enforcement on cots mobile devices," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 375–386. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906413