LibD: Scalable and Precise Third-party Library Detection in Android Markets

Menghao Li*‡§, Wei Wang*, Pei Wang†, Shuai Wang†, Dinghao Wu†¶, Jian Liu*§¶, Rui Xue‡§, Wei Huo*§

*Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences, China

†College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA

‡State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China

§School of CyberSpace Security at University of Chinese Academy of Sciences, China

{limenghao,wwei,liujian6,xuerui,huowei}@iie.ac.cn, {pxw172,szw175,dwu}@ist.psu.edu, ¶corresponding author

***Abstract*—With the thriving of the mobile app markets, third-party libraries are pervasively integrated in the Android applications. Third-party libraries provide functionality such as advertisements, location services, and social networking services, making multi-functional app development much more productive. However, the spread of vulnerable or harmful third-party libraries may also hurt the entire mobile ecosystem, leading to various security problems. The Android platform suffers severely from such problems due to the way its ecosystem is constructed and maintained. Therefore, third-party Android library identification has emerged as an important problem which is the basis of many security applications such as repackaging detection and malware analysis.**

According to our investigation, existing work on Android library detection still requires improvement in many aspects, including accuracy and obfuscation resilience. In response to these limitations, we propose a novel approach to identifying third-party Android libraries. Our method utilizes the internal code dependencies of an Android app to detect and classify library candidates. Different from most previous methods which classify detected library candidates based on similarity comparison, our method is based on feature hashing and can better handle code whose package and method names are obfuscated. Based on this approach, we have developed a prototypical tool called LibD and evaluated it with an update-to-date and large-scale dataset. Our experimental results on 1,427,395 apps show that compared to existing tools, LibD can better handle multi-package third-party libraries in the presence of name-based obfuscation, leading to significantly improved precision without the loss of scalability.

***Keywords*-Android; third-party library; software mining**

I. Introduction

Mobile app market has been rapidly growing in the past decade. By July 2015, Android has become the largest mobile application platform in terms of the number of available apps [1]. Third-party libraries make app development much more convenient by offering ready-made implementations of specific functionality, e.g., advertisement, navigation, and social network services. A previous study shows that in some extreme cases, an Android app can refer to more than 30 different third-party libraries [2].

On the other hand, the widely used third-party libraries can also lead to new problems that hurt the security and stability of the Android ecosystem. For example, with advanced reverse engineering techniques, an adversary is able to modify popular advertising libraries and direct the revenues to a station under his control, while preserving the other functionality of the original apps. The adversary can then deploy the tampered and repackaged apps into an unofficial Android market to lure downloads. In this way, an attacker can contaminate a large number of apps by just tampering a few libraries. For another example, if a specific version of a popular social network library contains a security vulnerability, the threat from this vulnerability would be spread to many different apps and influence tons of users.

To countermeasure the emerging threats caused by vulnerable and harmful third-party libraries, the security community has longed for reliable techniques to accurately identify libraries in mobile apps at a large scale. There are currently two approaches to recognizing third-party libraries in Android apps. The first is based on whitelists of known libraries. A whitelist is typically generated through manual analysis [3], [4] and has to be constantly maintained to stay updated. Even though, it is hard to guarantee that such a list is comprehensive, considering there are currently millions of mobile apps available and new library providers keep emerging. Therefore, the whitelist-based method usually leads to both precision loss and high operation cost.

The other approach is to directly extract libraries from apps without *a priori* knowledge about the libraries [5]–[9]. In the extraction process, a mobile app is first divided into different components which are regarded as library candidates. Then a similarity metric or a feature-based hashing algorithm is designed to classify these candidates. If a group of similar candidates exists in different applications, components in that group are considered variants of the same library.

The second approach is currently the state of the art. Although the results reported by previous work have been very promising, they are still not as good as they could have been due to several limitations of the employed methods. Our investigation shows that most existing methods are heavily dependent on Java package names and package structures when detecting and classifying library candidates. However, package names can be easily mangled by name-based obfuscation and package structures may vary in different versions of the same library.

To further improve the accuracy of third-party library identification on the Android platform, we propose a new library detection and classification technique that can effectively overcome the limitations discussed above. Different from previous work which recognizes library candidates purely based on the Java package names and structures, we extract library candidates based on the reference and inheritance relations between the Java classes and methods, with the assistance of auxiliary information excavated from app metadata. After collecting these candidates, our classification technique will decide if there exist enough apps sharing the same group of candidates. If so, that group of candidates indeed forms a third-party library. Our classification method is implemented through a novel feature hashing algorithm, such that we can avoid pair-wise candidate comparison which is required by approaches based on binary similarity measurement. This design makes it easier for the classifier to scale to millions of Android apps. Compared to previous work which heavily depends on Java package names and structures, our method only treat them as supplementary information. As such, our research provides a more general solution to the problem of third-party library identification for Android.

We have implemented our detection technique in a tool called LibD and evaluated it with 1,427,395 apps collected from 45 third-party markets. Compared with similar tools like LibRadar [7] and WuKong [6], LibD can not only identify a much larger number of third-party libraries from the dataset but also find them with a higher precision.

In summary, we make the following contributions in this research.

- We develop a new third-party library identification technique for the Android mobile platform. Our method can overcome various limitations shared by the majority of previously proposed approaches. In particular, our method is resilient to Java package name-based obfuscation and diversified package structures.
- We implement our identification technique in a tool called LibD and test its performance with more than a million Android apps collected from 45 different markets. Compared to other similar tools, LibD is able to report better results in terms of both quantity, i.e., the number of identified third-party libraries, and quality, i.e., the identification precision.
- To benefit the research community, we share LibD at https://github.com/IIE-LibD/libd.git. Other researchers will be able to build various software engineering and security applications based on our work.

The rest of the paper is organized as follows. We first discuss the limitations of the previous work which motivate our research in Section II. We then present our third-party library identification method and its implementation in Section III. The experimental results are presented in Section IV. We discuss a few potential issues in Section V, review related work in Section VI, and conclude the paper in Section VII.

II. Motivation

In this section, we elaborate on two major limitations of previous research that motivate the development of our new third-party library identification technique. According to our investigation, the two limitations stem from similar design decisions shared by existing techniques. The assumptions behind these design choices, although valid in many cases, do impose constraints that affect the generality of the techniques.

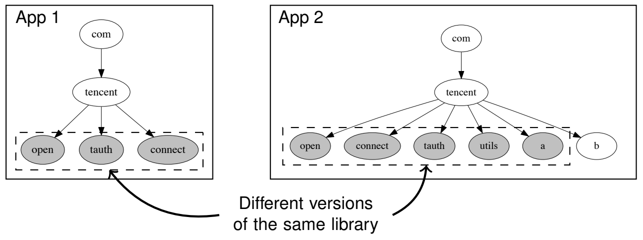

Fig. 1: Variants of the same library with different package structures

The first assumption which may be problematic is that the instances of an Android library included by different apps have the same package name. This assumption is the basis of the pre-clustering algorithms used in similarity-based library identification [8], [9]. Since these methods need to compute the pair-wise similarity among all library candidates in the dataset, they have to first partition the dataset and group candidates that are likely to be in the same cluster; otherwise the classification will not scale. Most similarity-based identification techniques use package names to tentatively cluster the candidates before undertaking fine-grained comparison. However, using package names as a feature for clustering becomes unreliable when obfuscation is considered. Package name obfuscation is one of the most widely used obfuscation methods for Java code. A recent study on Android libraries showed that over half of the inspected instances are protected by obfuscation techniques [2]. As a consequence, identification methods utilizing package names as the primary features to detect and classify libraries are likely incapable of handling a considerable portion of Android apps on the market.

Some researchers have realized that deeply depending on package names can make the identification method less robust. A recent tool called LibRadar [7] developed an algorithm that takes obfuscated package names into consideration. Rather than binary diffing, LibRadar classifies library candidates through feature hashing. Therefore, LibRadar does not need pair-wise similarity comparison between library candidates and does not need package names for pre-clustering. However, LibRadar recognizes library candidates according to the directory structures of the packages. In particular, LibRadar requires a library candidate to be a subtree in the package hierarchy. This is another assumption that may not be valid in reality, because we found that a library can be differently packaged in its different versions, as illustrated by a real-world example in Fig. 1.

Motivated by the reasoning above, we aim to develop a new third-party library identification method that does not take the two aforementioned assumptions for granted. Although our method does not completely abandon the package-level information, we utilize it as supplementary features in the identification process.

III. Method

A. Overview

We now outline the design of the proposed approach. As shown in Fig. 2, the overall workflow consists of four steps. We first decompile the input app and recover the intermediate representation (IR). Information is then retrieved from the IR regarding multiple levels of the Android app organizations (i.e., packages, methods, and classes) and the primary relations among them (i.e., inclusion, inheritance, and call relations). We then leverage the retrieved information to build the instances of potential libraries. Note that each instance has standalone functionality, and it consists of one or multiple packages. The next step is to generate features from each instance; we propose techniques at this step to select features that are mutation-sensitive and resilient to name-based obfuscation (hereinafter referred to as simply obfuscation). With a predefined threshold of occurrence, third-party libraries are identified by clustering instances with equivalent features.

We have implemented our technique in a prototype tool called LibD, which consists of 3,529 lines of code in Python. We deployed our experiment environment on OpenStack, a cloud computing platform [10]. We implemented 408 lines of code to manage machines and schedule experiments on this platform. Ten virtual machines were employed to analyze Android apps in parallel.

B. App Decompilation

The first step of our approach is to decompile the input Android apps. As shown in Fig. 3(a), a directory tree is generated by decompiling an Android app. Each node on the directory tree can include Java class files as well as subdirectories (i.e., the edges to the successor nodes). Note that each tree node with a set of class files is actually an Android package [11]; in this research, we group package nodes on the directory tree to recover third-party library instances. There also exist some other nodes that only contain subdirectories (e.g., `com/tencent` node in Fig. 3(a)); we ignore such nodes due to their trivial contents.

Same as existing work [6], [12]–[15], we employ two commonly-used app analysis tool Apktool [16] and Androguard [17] to decompile the input apps. Apktool is used to extract the tree structures of the decompiled apps. We recover the whole directory structures with all the class files. In addition, we use Androguard to find relations between packages, classes, and methods. Three informative relations are particularly collected to construct the homogeny graphs (§III-C1) and call graphs (§III-C2). We now introduce each relation in details.

- **Inclusion relation.** The first relation describes the parent-child structures on a directory tree. Considering the path that leads from `com/tencent` to `/connect` in Fig. 3(a), such path represents an inclusion relation.
- **Inheritance relation.** We also record the program inheritance relations; inheritance relations can be directly read from the decompiled class files. Fig.3(b) shows the inheritance relations between package `/common` and two other packages.
- **Call relation.** This relation represents the inter-package function calls. Fig.3(c) describes the call relation between packages `/connect/auth`, `/tauth` and `/open`. For example, by identifying the function call between methods in `Auth..$..listener.smali` and `AuthActivity.smali`, `/tauth` (i.e., callee) and `/connect/auth` (i.e., caller) are considered to have the call relation.

C. Library Instance Recovery

One of the key contributions in this paper is our systematic approach to recovering the boundaries of third-party libraries. Given the decompiled outputs of an input app, we traverse its homogeny graphs (explained in §III-C1) as well as the call graph for multiple iterations, each of which relies on different conditions to prioritize nodes and edges. The outputs of our traversals are weakly connected components; each component (including one or several packages) has an independent functionality, and such components are assumed as the instances of potential libraries. We now detail our technique to recover library instances in a two-step approach.

1) Homogeny Package Union Construction: The first step is to find highly-correlated packages regarding the inclusion and inheritance relations (§III-B). Before discussing our algorithm, we first define three terms as follows.

Definition 1: **Homogeny package.** Let P_i and P_j be two packages of the input app, we say P_i and P_j are homogeny packages if there are inclusion or inheritance relations between them.

Definition 2: **Homogeny graph.** A homogeny graph is a directed graph $\mathcal{H} = (V, E)$, where V is the set of all the app packages, and E is the set of inclusion or inheritance relations.

Definition 3: **Homogeny package union.** A homogeny package union consists of one or several homogeny packages; each union is a weakly connected component on the homogeny graph. A weakly connected component is a maximal connected subgraph of the undirected graph resulted from replacing all the directed edges with undirected edges in the original directed graph.

Algorithm 1 describes how we find homogeny package unions. We construct the homogeny graph with every package in the app and their inclusion and inheritance relations as the graph edges (line 2, line 5–6). Note that before constructing the graph, we first eliminate two kinds of special packages (line 3–4). The first elimination (line 3) rules out packages at the root of a directory tree. According to our observation, an app usually does not have class files in the root directory; developers would put the code base (packages) starting from the second level of the directory tree. Actually our study on 2,000 commonly-used apps only finds *three* apps to have class files in the root nodes. Further investigation shows that all of these class files are used to impede reverse engineering. As a result, we rule out packages if they are in the root

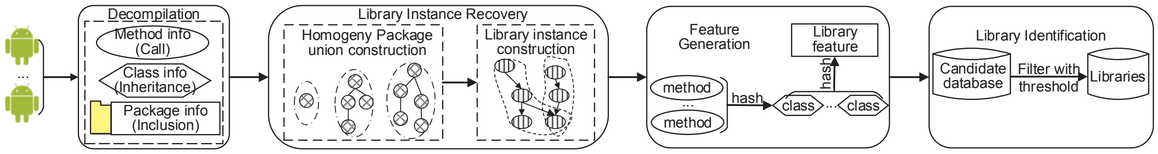

Fig. 2: The architecture of LibD

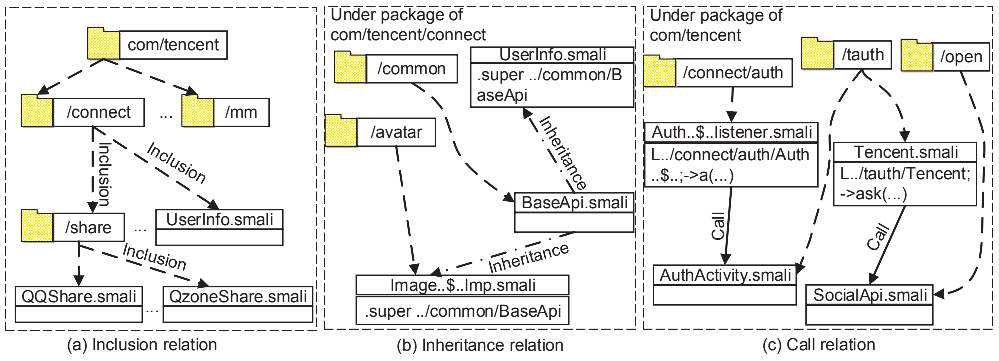

Fig. 3: Packages in a typical app directory tree and three critical relations.

nodes. Besides, the Android official libraries (e.g., `android/support/v4`) are also trimmed off (line 4), as our main focus is on the third-party libraries.

After the elimination, we search for weakly connected components on the homogeny graph (line 7–8); such components could contain single or multiple nodes (packages). As nodes in each component are connected by inclusion and inheritance edges, each identified component is one *homogeny package union* following our definition.

Algorithm 1: Homogeny package union construction

```
Input: Android app $p$
Output: Homogeny package union set $\mathcal{H}_p$
1 $\mathcal{H}_p \leftarrow \varnothing$; $\mathcal{H} \leftarrow \varnothing$,
2 $\mathcal{H}.V \leftarrow$ packages in the input app; /* $V$ is the set of vertices. */
3 filter out packages in the root nodes in $\mathcal{H}$;
4 filter out Android official packages in $\mathcal{H}$;
5 $\mathcal{H}.E \leftarrow$ inclusion relation set; /* $E$ is the set of edges. */
6 $\mathcal{H}.E \leftarrow \mathcal{H}.E \cap$ inheritance relation set;
7 for each weakly connected component $g$ in $\mathcal{H}$ do
8     $\mathcal{H}_p$.add(g);
9 return $\mathcal{H}_p$
```

2) Library Instance Construction: Given the constructed homogeny package unions, the next step is to group one or several unions together to recover the instances of potential third-party libraries. Our manual investigation of over 200 real-world commercial apps reports method calls as a quite informative feature. Thus, we first recognize all the inter-union function calls and build the call graph. As a result, identifying library instances essentially becomes a task to collect all the reachable nodes on the call graph from the root nodes.

Algorithm 2 presents our approach to generating the call graph for homogeny package unions and finding the instances of potential libraries. We first build the call graph $\mathcal{I}$, according the inter-union calls (line 2–5). We then filter out noisy calls (line 6–7) in terms of two criteria. Finally, for each weakly connected component, we search for "root nodes" and collect all the reachable components from one root node as one instance of a potential library (line 9–12). Naturally, the root node is defined as a node on the call graph with no incoming edges. On the other hand, if there is no root node, we output the connected component as one library instance (line 14).

In this research, we identify and eliminate two noisy calls that could impede our analysis. The first one describes the call graph edges connecting the application code and the libraries. Such connections could incorrectly bridge two library instances through the application code, thus overestimating the library boundaries. We identify application code according to the manifest files in the input apps; the application code and evolved call edges are trimmed off on the call graph (line 6).

We also observe a special call that could lead to false positive in this research; we name it *ghost call*. A ghost call appears in a method, but neither the caller nor the callee exists in the DEX code of the decompiled app. Actually such ghost calls are not rare; we find 82 apps containing the ghost calls during our analysis of 10,043 test samples. Most of "ghost calls" are calling functions from customized Android frameworks. For example, there is a call invoking `com.samsung.android.SsdkInterface.getVersionCode` which exists only on Samsung phones. The decompiler failed to consider these cases, leading to dangling function targets. To filter out such errors, we check the appearance of both caller and callee for each call relation, and eliminate those ghost calls (line 7).

D. Feature Generation

As previous mentioned, a library instance includes one or more homogeny package unions, while a union can consist of multiple packages. The feature of a library instance can be defined as the combination of package features, and further

Algorithm 2: Library instance construction

```
Input: Homogeny package union set $\mathcal{H}_p$
Output: Library instance set $\mathcal{I}_l$
1  $\mathcal{I} \leftarrow \varnothing$;
2  $\mathcal{I}.V \leftarrow \mathcal{H}_p$; /* $V$ is the set of vertices. */
3  for any union $u_1$ and $u_2$ in $\mathcal{I}$ do
4      if there is a call relation in $\langle u_1, u_2 \rangle$ then
5          add $\langle u_1, u_2 \rangle$ in $\mathcal{I}.E$; /* $E$ is the set of edges. */
6  filter out application code-related calls in $\mathcal{I}$;
7  filter out ghost calls in $\mathcal{I}$;
8  for each weakly connected component $g$ in $\mathcal{I}$ do
9      if there are root nodes in $g$ then
10         for each root do
11             $cl \leftarrow$ reachable components from this root;
12             $\mathcal{I}_l$.add($cl$);
13     else
14         $\mathcal{I}_l$.add(g);
15 return $\mathcal{I}_l$
```

divided into features of classes in each package. Since each class usually consists of several methods, in this research we employ method-level features as the basic elements to construct the library instance-level features.

To this end, we first build the control flow graph (CFG) of each method. The feature of every basic block on the CFG is calculated by hashing all the opcodes inside the block. We then concatenate the features of the basic blocks on the CFG in a depth-first order. For a parent node with two or more children, we sort the values of the children nodes and prioritize the node with the smallest value.

We then construct the feature of a class with features of all its methods. To this end, we concatenate the feature values of all methods in a non-decreasing order. Such feature sequences is then hashed again as the class-level features. Finally, we build the library instance-level feature following the same strategy—sorting all of its class-level features in a non-decreasing order and hashing the feature sequences.

Note that one of our central design choice is to generate mutation-sensitive and obfuscation-resilient features for each library instance; such design choice can enable finer-grained Android app analysis in an efficient way. We now discuss how we satisfy such requirements.

1) Mutation Sensitive: To produce features that are sensitive to library mutations, we generate hash value from opcodes of *all* the instructions in the basic blocks. Since even subtle modifications would lead to the changes of the underlying instructions, our instruction-level hashing should be surely updated regarding almost all the mutations.

Actually many (security-related) mutations, e.g., the remote control vulnerability exposed in *Baidu moplus SDK* [18], would only update a single line of code in one specific version of the library. That means, previous system API-based library detection algorithm is not able to distinguish such mutations. On the other hand, by hashing the underlying instructions within each basic block, features utilized in our research can preserve the sensitivity in front of various real-world scenarios.

Naturally, mutations with different features are considered as different library instances. That means, instances of one library can be put into different groups if they have different features. To further cluster mutations, we compare the package names of mutations; in our current design, two mutations are considered from the same library if they have the same name.

2) Obfuscation Resilient: Our in-depth study of obfuscated Android apps shows that commonly-used Android obfuscators are typically designed for renaming; package, class, and even method names will be obfuscated into meaningless strings (e.g., `/t`, `/a`, `/b`). Stating such observation, LibD is designed to only hash the underlying *opcode* sequences as the features of each basic block. Note that by extracting features from the underlying implementation, LibD is naturally resilient towards renaming on package names. In addition, although renaming on class and method names can change the operands of certain control-flow instructions, the original opcodes are preserved. For example, method call instructions would have different operands when the callee's name is obfuscated. However, since we only calculate the hashing value of the *opcode* sequences within basic blocks and do not consider the *operands*, LibD is suitable to defeat the class and method-level renaming obfuscations. In sum, features extracted by LibD are obfuscation-resilient, as shown in our experimental results.

Note that given our renaming-resilient features, obfuscated library instances should be clustered into the same group as their normal versions. In other words, we are able to recover the original identity of the obfuscated libraries by investigating instances clustered into same groups.

E. Library Identification

Given an input app, the aforementioned techniques can generate instances of potential libraries (§III-C) as well as features of each instance (§III-D). We actually repeat such process towards a large amount of apps and collect all the identified instances and their features (experimental details are disclosed in §IV).

The next step is to cluster instances regarding their features; instances with equivalent features are put into one group. We set a threshold according to our empirical study results. A group of library instances is considered to truly represent a third-party library only if the number of instances in this group is equal or greater than the threshold. Details on how we set the threshold are presented in §IV-B.

We label each cluster with the library name found in the package information. We also try to merge certain clusters if they have the same library name; such merged clusters should indicate library mutations.

IV. Evaluation

A. Dataset

We collect in total 1,427,395 Android apps from 45 third-party markets. Although the official app market, Google Play, contains over a million apps [1], it employs more rigorous reviews on the uploaded apps, including both static and dynamic analysis, and presumably many malicious or vulnerable

TABLE I: Markets and the number of apps collected from each market.

Market	# of apps	Market	# of apps
mumayi	55,682	3533	15,871
appfun	47,090	apk91	27,190
520apk	13,048	Nduo	8,965
lenovo	151,426	lmobile	16,659
baidu	30,275	sougou	27,795
jifeng	30,661	anzhi	401,578
yingyongbao	5,184	anzow	12,521
hiapk	76,066	zs2345	4,538
gezila	11,030	7xz	4,871
xiaomi	63,494	huawei	6,804
yy138	5,073	16app	38,003
liqucn	10,134	apk3310	22,376
angeeks	54,432	appchina	244,413
others	62,216		
		total	1,427,395

TABLE II: Detected third-party libraries with different threshold settings.

Threshold	# of different instances	# of different libraries	Threshold	# of different instances	# of different libraries
50	9,868	2,350	30	17,298	3,827
45	11,061	2,584	25	21,405	4,550
40	12,576	2,893	20	27,763	5,811
35	14,563	3,298	15	38,150	7,576
32	16,074	3,567	10	60,729	11,458

third-party libraries could have been rejected during the review process [19], [20]. On the other hand, third-party markets usually do not have such review process, and we expect to collect more diverse library instances. As reported in our experiments, we successfully collect a considerable number of mutation and obfuscation samples. We choose broadly-used third-party markets (e.g., Huawei and apk91) as well as some infamous Android forums for evaluation. Table I lists the markets we used and the corresponding apps collected in each market.

B. Threshold

Recall we need to define a threshold to decide whether a cluster of library instances surely represents a third-party library (§III-E). The threshold is the number of appearances of a potential library candidate in the dataset, in our research and previous studies [6], [8]. In this section, we first study how many libraries can be detected regarding different thresholds and try to set a reasonable threshold by comparing with a previous whitelist approach [3].

Previous work sets different thresholds to cluster libraries. For example, Wukong [6] sets the threshold as 32 while Li et al. use 10 [8]. To present an in-depth and thorough study, we iterate different thresholds from 10 to 50 and record the clustered libraries. Table II shows the detection results regarding different threshold settings. Recall LibD clusters library instances with the same feature into one group, and further clusters groups together if they have the identical name (§III-E). In Table II, we report the number of library instances with different features, as well as the number of different library names (i.e., the detected libraries). Naturally, with the increase of the threshold, less libraries can be found.

We consider the total number of detected libraries is quite promising. Even by setting the threshold as a relatively-high value, i.e., 50, we report LibD can still detect over two thousand libraries.

TABLE III: The number of libraries reported in the whitelist but not found in LibD's outputs.

Threshold	# of neglected libraries	Threshold	# of neglected libraries
45, 50	16	20	4
40	13	15	1
35	8	10	0
25, 30	7		

When comparing with previous work, we report that LibD can detect more third-party libraries than both whitelist and system API-based methods [3], [6]. We will present further discussions in §IV-D.

Setting threshold according to a library whitelist [3]. Chen et al. [3] present a whitelist, including names of 72 commonly-used third-party libraries in the market. We validate each threshold used in Table II to find a proper one that could include *all* the libraries reported in the whitelist. As shown in Table III, when the threshold decreases to ten, all the libraries reported in the whitelist can be found in LibD's outputs. As a result, ten is used as the threshold in our following experiments.

C. Comparison with LibRadar

To our best knowledge, there is no systematic approach to giving us the ground truth (i.e., third-party libraries) from Android apps—the boundary of a library is unknown. Manual collection is also not feasible with over a million apps.

To present a convincing and feasible evaluation, we randomly collect 1,000 apps from our dataset as a subset, and manually investigate the subset to get the ground truth. We assume a package name indicates an instance of a third-party library if it is a legal domain name. We employ `nslookup` to check the package names on the Domain Name System (DNS) according to the following conditions.

- If the library name represents a legal domain name, then it is a library instance.
- If the package name is a subdomain of a domain name, we will then check the entire name on search engines (e.g., Google) to verify whether the suspicious name has been used by others. If so, we consider to find a new library instance.

Following the above strategy, we acquire in total 2,613 libraries as the ground truth from the 1,000 apps. We then use LibD and a state-of-the-art library detector, LibRadar [7], to detect third-party libraries in the 1,000 apps. LibRadar [7] provides an online service to detect libraries; this enables convenient comparison with LibD. As LibRadar only provides the names of the detected libraries, we compare LibD with LibRadar regarding the library names.

Table IV presents the performance of both LibRadar and LibD. We also validate the detected libraries according to the ground truth. In general, LibD identifies 1,954 libraries, among which 1,456 libraries are true positive. LibRadar finds 670 different libraries in total, and 264 libraries are true positive. Considering the false positive and negative rates, we report that LibD can notably outperform LibRadar in both criteria, and the overall result is quite promising.

TABLE IV: Comparison with LibRadar. We also validate the results according to the ground truth (2,613 libraries).

	# of detected libraries	# of true positive	false positive rate (%)	false negative rate (%)
LibRadar	670	264	60.6	89.9
LibD	1,954	1,465	25.0	43.9

Besides the validation according to the ground truth, we also evaluate the correctness in terms of obfuscated libraries. Both LibRadar and LibD are able to map the obfuscated libraries to their unobfuscated instances. As the online service of LibRadar is not stable during our experiment, we were only able to test 100 apps at this step. Among these 100 apps, LibRadar reports to find 13 obfuscated libraries, while LibD reports 14. Our manual investigation on the outputs of LibRadar shows five false positive. For example, LibRadar incorrectly considers library `com/avos/avospush` as an obfuscation version of the Android official library `android/support/v4`. Note that the implementations of these two libraries are quite different. On the other hand, no error is reported when we manually correlate the obfuscated libraries detected by LibD with their original instances. We interpret the main reason for LibRadar's high false positive is that it captures features from the System APIs used by the libraries; two libraries are considered identical if they use the same APIs. On the other hand, since LibD captures the features regarding the underlying implementation, commonly-used obfuscation methods would not impede LibD.

D. Comparison with Other Work

Table II reports our library detection results regarding different thresholds. In the following section, we present an in-depth study on this library detection results and compare the results with previous work in terms of different aspects.

Comparing with Li et al. [8]. Li et al. detect 1,113 libraries from apps on Google Play [8]. Like the whitelist approach [3], Li et al. only provide the detected (unobfuscated) library names. We compare their reported names with LibD.

In general, among 1,113 libraries detected by their approach, we report 262 libraries are new to LibD. Further study shows that 249 of the 262 libraries are indeed *highly correlated* to our results; they have similar names and structures. For example, `com/comScore/exceptions` and `com/comScore/stramsense` are two libraries reported in their list, which have no match in our results. However, we found a library named `com/comScore` in the outputs of LibD; this library has the same root and second directory names (i.e., `com` and `comScore`). Given the similar names and structures, we assume these 249 unmatched libraries are caused by different techniques deployed to recognize library boundaries.

Our finding also shows that there are 13 libraries totally new to LibD. Since Li et al. analyze apps on the Google Play, we expect libraries that are only used by apps in the Google Play would be unknown to LibD. For example, several apps on Google Play are linked with library `com/android/psu`, and this library is absent in our third-party market dataset.

Comparing with WuKong [6]. Comparing with WuKong [6], our approach finds more libraries with the *same* threshold. WuKong takes 32 as the threshold to detect libraries, and we evaluate LibD with the same threshold. In addition, since WuKong considers each mutation (referred as "version" in their paper) as one library, here we use the same measurement for LibD (i.e., the "# of different instances" columns in Table II). In general, WuKong reports to detect around 10,000 libraries in [7], while LibD finds 16,074 libraries with the same threshold.

We interpret the comparison results as quite promising. In particular, we consider such improvement mainly comes from the much finer-grained features retrieved by LibD. As previously discussed, features captured by LibD are sensitive to the underlying mutations of the apps. On the other hand, since WuKong uses system-level APIs to detect libraries, this approach is not suitable to find many (subtle) library mutations (§III-D).

E. Processing Time

Our system is deployed on top of OpenStack, including ten virtual machines. All the virtual machines are configured with a Xeon E3-1230 CPU and 2GB RAM. The operation system is Ubuntu 14.04 LTS x64.

We report that LibD takes no more than 10 seconds to analyze an app. App decompilation (§III-B), including intermediate representation recovery and package relation construction, takes around 6 seconds. Library instance recovery (§III-C) takes around 2 seconds. Average clustering time of one library instance is less than 12 milliseconds, and we report on average it takes 100 milliseconds to cluster all the library instances in an app.

Comparing with LibRadar, we report LibD's average processing time for one app is around 2 seconds longer. Naturally, as LibD undertakes much finer-grained analysis, it can cost more time. Overall, LibD is quite efficient and scalable.

F. Further Investigation

In the following subsections, we study three typical challenges in Android library detections, i.e., multi-package libraries, obfuscated libraries, and library mutations. Note that as we confirm a library (mutation) according to the number of instances in a cluster (§III-E), some instances—even if they are multi-package, obfuscated or library mutations—would be ignored if the total number of their appearances is less than the threshold. To present a thorough study, we use all the different instances of potential libraries, in the whole set of 1,427,395 apps, for multiple evaluations in the following subsections (i.e., data reported in the "# of different instances" columns in Table II).

1) Multi-Package Libraries: A third-party library may contain more than one package. Benefited from our novel library boundary identification technique, LibD discovers many multi-package libraries. In particular, when setting the threshold as ten, we report to find 5,141 multi-package libraries (8.4% of all the detected libraries in total).

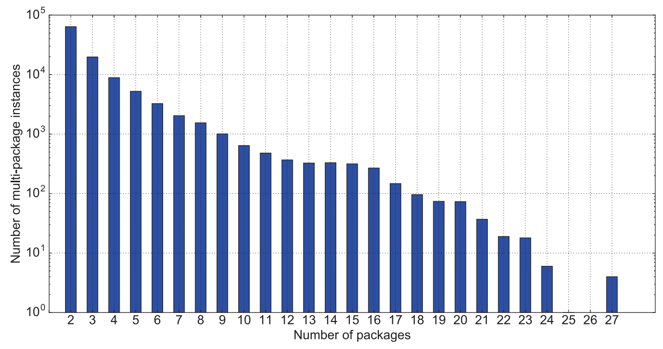

Fig. 4: Distribution of different multi-package library instances regarding the number of packages.

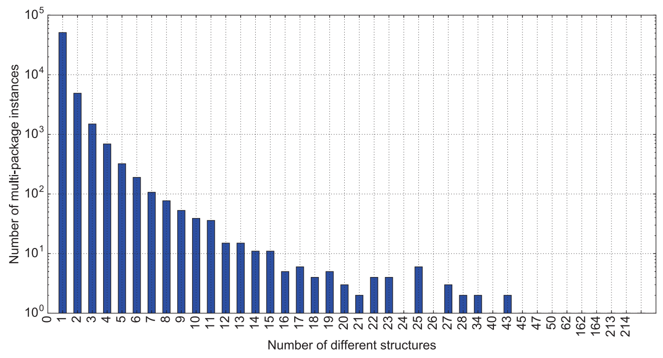

Fig. 5: Distribution of different multi-package library instances regarding the number of structures.

We also use all the different library instances for evaluation, as they can reveal potential rare changes on the third-party libraries. Fig. 4 presents the distribution of different multi-package library instances. The number of library instances decreases quickly with the increase of the packages each instance contains. Most of the multi-package instances contain two packages; there are in total 63,948 two-package instances (58.8% of all the multi-package instances). We manually analyzed 10 commonly-used instances (e.g., `/com/tencent/wap`) and the result shows that the library boundaries are reasonable.

We also find that multi-package library instances can have different internal structures. For example, library `/fly/fish/adil` has three different structures; each of which contains two, three and four packages, respectively. We consider two library instances have different structures if their internal package names or the number of packages are different. We also report the distribution of different multi-package instances regarding the number of structures. As shown in Fig. 5, while there are 51,099 instances with only one structure, 67,147 instances actually contain more than two different structures (56.7% of the multi-package instances). Our experiments also report that a library could have 214 different structures at most.

Further investigation of multi-package libraries also reports that some packages are shared by several multi-package libraries. Those shared packages usually provide some common utilities. Table V presents 5 packages that are shared by at least two libraries. Given the observation that the first two

TABLE V: Five shared packages and evolved libraries.

Shared packages	Evolved libraries
/cn/sharesdk/framework	/cn/sharesdk/douban /cn/sharesdk/sina /cn/sharesdk/wechat /cn/sharesdk/oneshare /cn/sharesdk/tencent /cn/sharesdk/twitter /cn/sharesdk/google /cn/sharesdk/whatsapp
/com/weibo/sdk	/com/weibo/net /com/weibo/android
/cn/emagsoftware/sdk	/cn/emagsoftware/android /cn/emagsoftware/sms
/com/umeng/common	/com/umeng/update /com/umeng/analystic /com/umeng/newxp /com/umeng/socilize
/com/mobi/tool	/com/mobi/controller /com/mobi/weather /com/mobi/assembly

TABLE VI: Top ten commonly-used first two segments of package names.

Directory	# of libraries
/org/fmod	2,613
/twitter4j/util	2,480
/LBSAPIProtocol/a	2,217
/twitter4j/management	2,184
//com/unionpay	2,167
/twitter4j/json	1,723
/com/tencent	1,192
/roboguice/content	1,109
/com/umeng	1,308
/com/facebook	764

segments of these package names are the same, we assume that they should come from the same developers.

Many library names have three or even more segments (e.g., `/com/facebook/util` has three "segments"). However, we observe that many of the first two segments of package names are identical. We report that there are in total 18,594 different kinds of first two segments in the outputs of LibD; we list the top ten in Table VI. Note that if the first two segments of two library names are identical, they are likely from the same developers. In other words, Table VI shows that most library developers prefer to provide a series of libraries instead of one.

2) Obfuscated Libraries: LibD is designed to address name-based obfuscation techniques. Obfuscators (e.g., Proguard [21]) replace the library name with several meaningless strings while preserve the original directory structures. Our experimental results report two kinds of renaming strategies. The first one obfuscates the library full names; all the names in the directory structures are replaced with meaningless strings, such as `c/a/b` or `u/y/e`. For such obfuscation, we are unable to get any useful information by only analyzing the name. The second obfuscation partially changes the library names (e.g., the last segment of the library name `/com/tencent/t` is obfuscated). Libraries with such partial obfuscation can usually provide some information of their functionalities or developers.

As shown in Table II, by setting the threshold as ten, LibD can detect 11,458 different libraries in total. With our manual

TABLE VII: Distributions of the obfuscated library instances.

# of obfuscated names	# of instances	Percentage (%)
<5	14,931	76.41
5–10	2,238	11.45
10–50	1,736	8.88
50–100	258	1.32
100–1,000	340	1.74
>1,000	37	0.20
Total	19,540	100

TABLE VIII: Libraries with the top ten number of mutations.

Rank	Library name	# of mutations in each identified library	# of mutations in total
1	/com/sina/sso	84	175
2	/com/ut/device	10	42
3	/com/nineold/androids/animation	79	222
4	/com/alipay/android	381	2,485
5	/m/framework/utils	55	131
6	/com/google/gson	421	2,422
7	/com/android/vending	161	2,126
8	/com/alipay/mobilesecuritysdk	173	463
9	/com/tencent/mm	288	1,552
10	/cn/sharesdk/wechat	168	495

effort, we report that there are about 5,000 obfuscated library instances in our dataset, among which 1,453 are completely renamed, while the rest (around 3,500) are partially renamed. According to our best knowledge, there is no well-developed automatic approach to distinguishing a (partially) renamed library from the others. In other words, our manual verification of library obfuscation is already the best effort.

In total, we have found that 19,540 different library instances (i.e., library instances with different features) are obfuscated. Table VII presents six groups of obfuscated instances; instances in each group have different number of obfuscated names. In general, around 24.5% library instances have equal or greater than 5 different obfuscated names. We interpret that obfuscations are actually quite common in real-world Android applications.

3) Library Mutations: In this section we study the library mutations. In general, our experimental results report plenty of libraries with more than 100 mutations. For example, `/com/google/gson` has 421 different mutations, while `/com/baidu/android` has 197 mutations.

Table VIII lists the identified third-party libraries with the top ten number of mutations. We also report the number of mutations when considering all the different library instances (§IV-F). Our study shows that many mutations indeed only modify a few lines of code. For example, each updating on library `/com/ut/device` only adds a few `move` opcodes. We also find some major updates among mutations of certain libraries, i.e., library structure-level changes. For example, some mutations of `/com/google/gson` contain only a few classes, while others can even include multiple packages.

Another finding is that the number of the "ignored" mutations (fourth column in Table VIII) is even greater than the confirmed library mutations. In other words, we consider there are actually many libraries having "stealthy" mutations; mutations that are only used by less than 10 apps in the third-party markets we studied. We consider these mutations potentially indicate illegal or even malicious behaviors.

V. Discussion

Obfuscation. Obfuscation has been broadly used by many Android applications. In this section, we investigate the commonly-used Android obfuscation tools and discuss the potential advantages and weakness of our techniques in front of them. Proguard [21], the official obfuscation tool provided by the Android SDK, is considered the most popular obfuscator in the Android developer community. This tool is essentially designed for renaming obfuscation. Typical renaming obfuscation can modify the package, class, and even method names into meaningless strings. As previously discussed (§III-D2), by hashing the underlying opcodes, LibD is resilient to the renaming obfuscations. Our evaluation also presents promising results in detecting obfuscated third-party libraries (§IV-F2).

We have also noticed that some obfuscation tools can perform code encryption or even program control flow-based obfuscations [22]–[24]. Given our current design, such advanced obfuscations can impede LibD to certain degree. However, since most code encryption and control flow-based obfuscations follow predefined patterns, deobfuscation is mostly feasible. Actually there has been much orthogonal work proposing to deobfuscate those techniques [25], [26].

Setting threshold. As previously discussed (§III-E), LibD identifies libraries according to a predefined threshold, and we set the threshold by validating a board set of candidates regarding an existing work (§IV-B). Although our experiments report promising results given the threshold as ten, conceptually, a "module" shared by only two Android apps can be considered as a library. In other words, determining a foolproof threshold regarding real-world Android applications may need further investigation and study.

The current implementation of LibD can be easily configured with different thresholds. Besides, we consider a rigorous training step regarding the ground truth should also be applicable in our research. On the other hand, since there is no systematic approach to acquiring the ground truth, our current ground truth set constructed by manual efforts may not be sufficient for training (§IV-C). In sum, we leave it as further work to extend the size of our ground truth set and launch a rigorous training procedure to decide the threshold.

Semantics-based similarity analysis. LibD detects instances of potential libraries (§III-C2); instances with identical features are clustered into one group (i.e., a third-party library). Conceptually, we are indeed searching for the hidden "similarity" among different code components (e.g., Java packages).

Note that features extracted by LibD (e.g., opcode sequences) are essentially from the program syntax. Syntactic features are straightforward representations of the target programs, and they have been widely used by many existing work for program similarity comparison and code clone detection [27]–[29]. Our experimental result has also demonstrated efficient and precise detection of Android third-party libraries (§IV-C) using syntactic features.

On the other hand, we have also observed some program semantics-based similarity analysis work [30]–[32]. Ideally, similarity analysis work in this category retrieves features by modeling the functionality of the program, and it can usually reveal the underlying similarities of code snippets in a more accurate way. However, existing semantics-based similarity work may not be very scale [31], [32]. Given the high scalability as a requirement for Android third-party library related search, we consider it may not be feasible to directly adopt previous techniques in our new context. We leave it as future work to integrate more scalable semantics-based methods in our research.

VI. Related work

A. Third-Party Library Identification

Early work on third-party mobile library identification mostly focuses on advertising libraries. Book et al. [33] and Grace et al. [34] use the whitelist-based method for detecting advertising libraries. After collecting the names of well-known advertising libraries, they examine the existence of such libraries in a mobile app by package name matching. Later techniques like AdDetect [35] and PEDAL [4] start to employ machine learning methods to provide more accurate and comprehensive results, but they still target advertising libraries only. AdRob [36] analyzes the network traffic generated by the advertising services in Android apps to identify which libraries are bundled, with both static and dynamic analysis.

Identification techniques specialized for advertising libraries are not suitable for many security analysis on mobile apps. Recent research has proposed more general methods that do not rely on *a priori* knowledge about what types of libraries are to be identified. WuKong [6] is an Android app clone detection technique which needs to filter out third-party libraries before the actual detection starts. WuKong adopts the assumption that a library consists of only one package. For each package, WuKong assigns the set of invoked Android API functions as its signature. Given a large set of apps, WuKong clusters all packages by this signature and reports clusters that are large enough to be recognized as a third-party libraries. LibRadar [7] is an online service that implements the identification method of WuKong, with a better-performing package clustering algorithm.

To distinguish app-specific classes from third-party-library classes, Vásquez et al. [37], [38] extracted the package name (i.e., main package) from `AndroidManifest.xml` for an app. Then, they considered all the classes inside the main package and its sub-packages as app-specific classes; classes outside the main package were considered as classes from third-party libraries. An empirical study conducted by Li et al. [8] investigated the usage patterns of third-party Android libraries. Another study by Chen et al. [9] tried to find potentially harmful libraries in iOS as well as Android apps. Both studies need to identify Android third-party libraries first, but they adopt an approach different from the one employed by WuKong. Instead of matching packages by signature, the two studies cluster library candidates by computing a distance metric between each pair of them. The distance is based on binary similarity and computed through binary diffing algorithms. With the distances computed, candidates close to each other are clustered and considered to belong to the same library. Since binary diffing is usually very costly, both studies have to perform pre-clustering based on package names to narrow the scope of pair-wise library candidate comparison, which could be impeded by obfuscation.

B. Applications

Third-party library identification has been used to implement many security applications targeting the Android ecosystem, one of which is Android app clone and repackaging detection [3], [6], [14], [39]–[41]. In this application, third-party libraries are considered noises, so they need to be detected and filtered out before app plagiarism is checked.

Another important application of library identification is mobile vulnerability analysis. Paturi et al. [42] and Stevens et al. [43] extracted advertising libraries from popular Android apps and studied the privacy leakage problems residing in these libraries. Jin et al. [44] discovered that some third-party libraries providing HTML5 support for mobile developers can be easily exploited by code injection attacks. SMV-HUNTER [45] analyzed the man-in-the-middle SSL/TSL vulnerabilities in Android apps and third-party libraries. Li et al. [46] found a vulnerability in a specific version of the Google cloud messaging library that leads to private data leakage. Since these vulnerabilities are sometimes closely coupled with specific libraries, identifying those libraries can be very helpful to searching for certain kinds of security threats. LibD can in general assist with these applications.

VII. Conclusion

In this paper, we present a novel approach to identifying third-party libraries in Android apps. Our method overcomes some long existing limitations in previous work that affect library identification accuracy. We have implemented our method in a tool called LibD. From a dataset of 1,427,395 Android apps recently collected from 45 markets, LibD identified 60,729 different third-party libraries with a manually validated accuracy rate that clearly surpasses similar tools. In particular, our tool possesses certain degrees of obfuscation resilience. Our experimental results show that LibD can find 19,540 libraries whose package names are obfuscated.

Acknowledgments

We thank anonymous reviewers for their invaluable comments and suggestions. This research was supported in part by the National Science Foundation of China (No. 61572481, 61402471, 61472414 and 61602470), the Program of Beijing Municipal Science & Technology Commission (No. Y6C0021116), the US National Science Foundation (Grant No. CCF-1320605), and Office of Naval Research (Grant No. N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912).

References

[1] "Number of apps available in leading app stores as of July 2015," http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[2] J. Lin, B. Liu, N. Sadeh, and J. I. Hong, "Modeling users mobile app privacy preferences: Restoring usability in a sea of permission settings," in *Proceedings of the 2014 Symposium On Usable Privacy and Security*, ser. SOUPS '14, 2014, pp. 199–212.

[3] 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 '14, 2014, pp. 175–186.

[4] 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, 2015, pp. 89–103.

[5] J. Crussell, C. Gibler, and H. Chen, "Scalable semantics-based detection of similar Android applications," in *Proc. of Esorics*, 2013.

[6] 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 '15, 2015, pp. 71–82.

[7] 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 (Demo Track)*, ser. ICSE '16 Companion Volume, 2016, pp. 653–656.

[8] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in Android apps," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, 2016.

[9] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zhou, "Following devils footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, ser. S&P '16, 2016.

[10] "Openstack," https://www.openstack.org/.

[11] "The Java tutorial: What is a package?" https://docs.oracle.com/javase/tutorial/java/concepts/package.html.

[12] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13, 2013, pp. 185–196.

[13] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Proceedings of the 17th European Symposium on Research in Computer Security*, ser. ESORICS '12, 2012, pp. 37–54.

[14] 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 and Mobile Networks*, ser. WiSec '14, 2014, pp. 25–36.

[15] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in Android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12, 2012, pp. 93–104.

[16] "Apktool," http://ibotpeaches.github.io/Apktool/, 2016.

[17] L. Adesnos, "Androguard," 2016, accessed: 2016-03-21. [Online]. Available: https://github.com/androguard/androguard

[18] Trendmicro, "Setting the record straight on Moplus SDK and the Wormhole vulnerability," http://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/.

[19] Google, "Android security white paper," https://static.googleusercontent.com/media/enterprise.google.com/en//android/files/android-for-work-security-white-paper.pdf.

[20] ——, "How we keep harmful apps out of Google Play and keep your Android device safe," https://static.googleusercontent.com/media/source.android.com/en//security/reports/Android_WhitePaper_Final_02092016.pdf.

[21] "Proguard," https://www.guardsquare.com/proguard.

[22] "Dexguard," https://www.guardsquare.com/dexguard.

[23] "Dash-O," https://www.preemptive.com/products/dasho/overview.

[24] "Dexprotector," https://dexprotector.com/.

[25] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," in *Proccedings of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05, 2005.

[26] D. Low, "Java control flow obfuscation," Ph.D. dissertation, The University of Auckland, 1998.

[27] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, ser. OSDI'04, 2004.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 96–105.

[30] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01, 2001, pp. 40–56.

[31] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 389–400.

[32] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. S&P '15, 2015, pp. 709–724.

[33] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of Android ad library permissions," *CoRR*, vol. abs/1303.0857, 2013.

[34] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '12, 2012, pp. 101–112.

[35] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of Android ad libraries using semantic analysis," in *Proceedings of the 9th IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, ser. ISSNIP '14, 2014.

[36] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: examining the landscape and impact of Android application plagiarism," in *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13, 2013, pp. 431–444.

[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*. ACM, 2014, pp. 242–251.

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

[39] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar Android applications," in *Proceedings of the 18th European Symposium on Research in Computer Security*, ser. ESORICS '13, 2013, pp. 182–199.

[40] ——, "Attack of the clones: Detecting cloned applications on Android markets," in *European Symposium on Research in Computer Security*, 2012, pp. 37–54.

[41] 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 *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.

[42] A. Paturi, P. G. Kelley, and S. Mazumdar, "Introducing privacy threats from ad libraries to Android users through privacy granules," in *Proceedings of NDSS Workshop on Usable Security (USEC'15). Internet Society*, 2015.

[43] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in Android ad libraries," in *Workshop on Mobile Security Technologies (MoST)*, 2012, p. 10.

[44] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014, pp. 66–77.

[45] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, “SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps,” in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium*, ser. NDSS ’14, 2014.

[46] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, “Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14, 2014, pp. 978–989.