# Large-scale Third-party Library Detection in Android Markets

Menghao Li*, Pei Wang†, Wei Wang*, Shuai Wang†, Dinghao Wu†, Jian Liu*§, Rui Xue‡§, Wei Huo*§ , Wei Zou*§
*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,zouwei}@iie.ac.cn, {pxw172,szw175,dwu}@ist.psu.edu,

***Abstract*—With the thriving of mobile app markets, third-party libraries are pervasively used in Android applications. The libraries provide functionalities such as advertising, location, and social networking services, making app development much more productive. However, the spread of vulnerable and harmful third-party libraries can also hurt the mobile ecosystem, leading to various security problems. Therefore, third-party library identification has emerged as an important problem, being the basis of many security applications such as repackaging detection, vulnerability identification, and malware analysis.**

**Previously, we proposed a novel approach to identifying third-party Android libraries at a massive scale. Our method uses the internal code dependencies of an app to recognize library candidates and further classify them. With a fine-grained feature hashing strategy, we can better handle code whose package and method names are obfuscated than historical work. We have developed a prototypical tool called LibD and evaluated it with an up-to-date dataset containing 1,427,395 Android apps. Our experiment results show that LibD outperforms existing tools in detecting multi-package third-party libraries with the presence of name-based obfuscation, leading to significantly improved precision without the loss of scalability.**

**In this paper, we extend our early work by investigating the possibility of employing effective and scalable library detection to boost the performance of large-scale app analyses in the real world. We show that the technique of LibD can be used to accelerate whole-app Android vulnerability detection and quickly identify variants of vulnerable third-party libraries. This extension paper sheds light on the practical value of our previous research.**

***Keywords*-Android; third-party library; software mining; code similarity detection**

## I. Introduction

The mobile app markets of Android have been rapidly growing in the past decade. By July 2015, Android has become the largest mobile application platform, measured in the number of available apps [2]. Third-party libraries make app development much more convenient by offering ready-made implementations of specific functionalities, e.g., advertisement, navigation, and social network services. A previous study shows that, in some extreme cases, an Android app can include more than 30 different third-party libraries [3].

Widely used third-party libraries leads to new software engineering problems that hurt the security and stability of the apps. For example, with advanced reverse engineering techniques, adversaries are able to tamper with popular advertising libraries and direct the revenues to a station under their control, while preserving the other functionalities of the original apps. The adversaries can then publish the compromised 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 with a few libraries. For another example, when a popular social network library contains a security vulnerability, the threat from this vulnerability would 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 [4], [5] and has to be constantly maintained to stay updated. Therefore, it is hard to guarantee that such a list is comprehensive, considering that there are currently millions of mobile apps available and new libraries 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 [6]–[10]. In the extraction process, a mobile app is first sliced 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 similar candidates are found in many different applications, the candidates are considered to be the same library, or variants of the same library.

The second approach is currently the state of the art. Although the results reported by historical research have been very promising, there is still room for improvement due to several common limitations of the existing methods. Our investigation shows that most of these methods are heavily dependent on Java package names and package structures for detecting and classifying library candidates. However, package names can be easily mangled by obfuscation and package structures may vary in different versions of the same library.

We have recently proposed a new library detection and classification technique that can effectively overcome the aforementioned limitations and improve the accuracy of third-

The first and second authors contributed equally to this work. Dinghao Wu and Jian Liu are the corresponding authors. A preliminary version [1] of this article appeared in *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, May 20–28, 2017.*

party library detection in Android apps [1]. Different from previous work that recognizes library candidates according to Java package names and structures, we extract the candidates based on the reference and inheritance relations among classes and methods, with the assistance of auxiliary information excavated from app metadata. In contrast of others, our method only treats Java package names and structures as supplementary information. After collecting these candidates, our classification technique decides if there exist enough apps sharing the same group of candidates. If so, that group is considered to be large enough to represent a third-party library. Our classification method is implemented through a novel feature hashing strategy, such that we can avoid pair-wise candidate comparison, which is required by many approaches based on binary similarity measurement. This design allows the classifier to scale to millions of Android apps. Overall, our research provides a more general solution to the third-party library identification problem on Android.

In this paper, we extend and enrich the library detection research by demonstrating the practical value of our new library detection method. We notice that many market-wide Android application analyses can be significantly enhanced and accelerated with an effective and scalable library detector. We frame two scenarios where our library detector serves as a boosting gadget for an important and heavy-weight Android analysis task. In the first scenario, we show that vulnerability analysis for a large number of apps can be made much less time-consuming by caching the partial analysis results of third-party libraries, which may be included by many apps, with only minor losses of overall analysis accuracy. In the second scenario, our library detection method is employed to identify defective variants of third-party libraries at a massive scale.

We have implemented our library detection method in a tool called LibD and evaluated it with 1,427,395 apps collected from 45 third-party Android app markets. Compared to similar tools like LibRadar [8] and WuKong [7], LibD not only identifies a much larger number of third-party libraries from the dataset but also finds them with better precision. We additionally designed two application scenarios to showcase how LibD can practically aid real-world Android analyses performed on market-size datasets.

In summary, we make the following contributions in this series of research:

- We developed 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 implemented our identification technique in a tool called LibD and tested its performance with over 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.
- We integrated LibDinto SmartDroid, a practical Android app vulnerability detection framework. With the help of LibD, SmartDroid becomes 5.5 times faster.
- With LibD, we were able to identify 10,801 vulnerable variants library from over a million of Android apps, featuring different vulnerability patterns.
- To benefit the research community, we LibD available 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 that motivated our research in Section II. We then present our third-party library detection method and its implementation in Section III. The experiment results are presented in Section IV. We elaborate on two case studies in Section V and Section VI, which are to demonstrate the practical value of third-party library detection in different scenarios. We discuss a few potential issues in Section VIII, review related work in Section IX, and conclude the paper in Section X.

## II. Motivation

In this section, we elaborate on two major limitations of previous research, which motivated 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 decisions, although valid in many cases, do impose constraints that affect the generality of the techniques.

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 [9], [10]. 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 in place. 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 [3]. 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 shelves.

Some researchers have realized that deeply depending on package names can make the identification method less robust. A recently developed library detection tool called LibRadar [8] employed an algorithm that takes package name obfuscation into consideration. Instead of 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

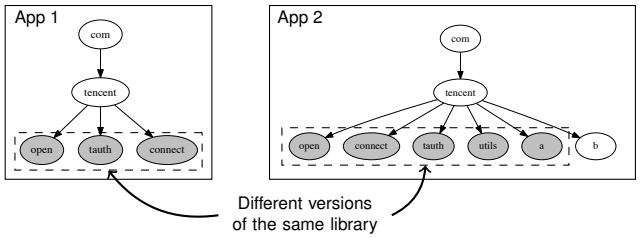



Fig. 1: Variants of the same library with different package structures

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) of the Dalvik bytecode of the app, named smali [11]. Information is then retrieved from the smali code at different levels, i.e., packages, classes, and methods. We also collect the relations among these program elements, including inclusion, inheritance, and invocation relations. We then leverage the retrieved information to build the instances of potential libraries. Note that each instance has standalone functionality and consists of one or multiple packages. The next step is to extract the features of each instance as the signature for testing equivalence. In this work, we propose a feature that is sensitive to minor code mutations while resilient to name-based obfuscations. With a predefined threshold of occurrence, third-party libraries are identified by clustering instances with equivalent signatures. That is, if the number of occurrences of a feature in the dataset reaches the threshold, we consider the corresponding code component as a library. For efficient comparison, we encode the instance feature into a text sequence and hashes it into a short representation using MD5.

We implemented our technique in a prototype called LibD, which consists of 3,529 lines of Python code. We set up the experiment environment on OpenStack, a cloud computing platform [12]. We implemented a scheduler with 408 lines of code to manage machines and issue tasks 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 classes as well as subdirectories (i.e., the edges to the successor nodes). Note that each tree node with a set of class is a Java package [13]. 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)). These nodes are mostly ignored by our analysis.

Following the practice of previous work [7], [14]–[17], we employ two widely used analysis tools, i.e., Apktool [18] and Androguard [19], 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 classes in each directory. In addition, we use Androguard to find relations between packages, classes, and methods. Three kinds of relations are collected to help us infer boundaries of closely coupled components in an app. We now introduce each relation in detail.

- **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 smali code. 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. We introduce a concept called homogeny package union, which is the basic unit in app partition. We further group homogeny package unions into different components based on inter-union function calls. Each component is expected to be highly cohesive while loosely coupled with the rest of packages in the app. Such components are considered to candidates of 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.

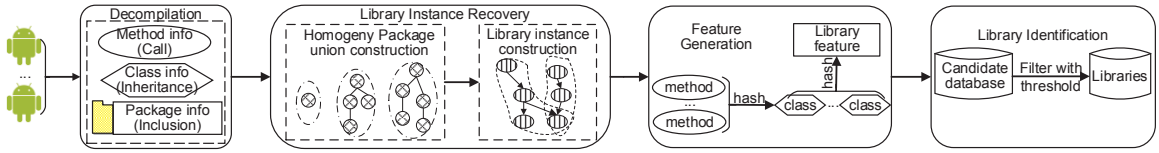
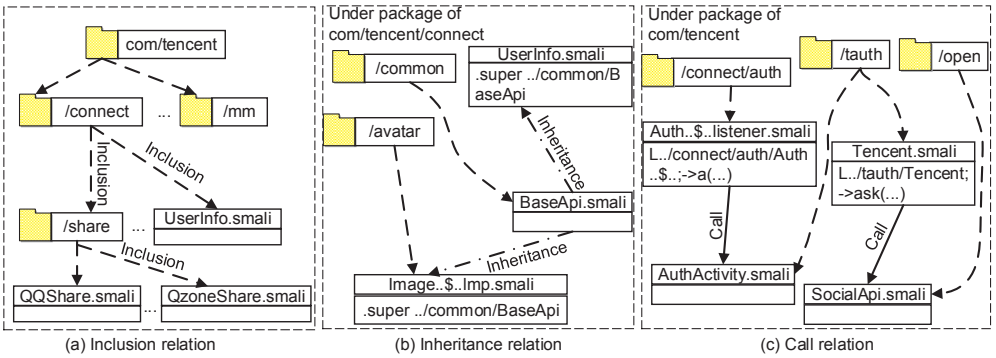



Fig. 2: The architecture of LibD




(a) Inclusion relation (b) Inheritance relation (c) Call relation

Fig. 3: Packages in a typical app directory tree and three critical 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 and lines 5–6). Note that before constructing the graph, we first eliminate two kinds of special packages (lines 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 contain classes in the root directory. Instead, developers typically deploy code bases starting from the second level of a directory tree. Studies on 2,000 popular apps found that only *three* apps have classes at directory roots. Further investigation shows all of these class are used to impede reverse engineering. The second elimination (line 4) trims off all standard Android libraries (e.g., `android/support/v4`), as our main focus is on the third-party libraries.

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

*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 indicates that method invocations are 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 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 $\quad \mathcal{H}_p$.add(g);
9 **return** $\mathcal{H}_p$

Algorithm 2 presents our approach to generate 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. Such ghost calls are not rare—we found 82 apps containing ghost calls among 10,043 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).

**Algorithm 2:** Library instance construction

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

### D. Feature Generation

As previously 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 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.

Many (security-related) mutations, e.g., the remote control vulnerability exposed in *Baidu moplus SDK* [20], 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 names of packages, classes, and even methods are commonly turned into meaningless strings (e.g., `/t`, `/a`, `/b`). To avoid being confused by this disturbance, 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 apply such process to 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 group instances by their signatures, i.e., the MD5 hashes of extracted features. Instances with the same signature are assigned to the same group. A *clustering threshold* is selected to decide if a group is large enough to represent a library. That is, a group of library instances

TABLE I: Numbers of apps collected for evaluation and their origins.

| Market | # of apps | URL |
|---|---|---|
| mumayi | 55,682 | www.mumayi.com |
| appfun | 47,090 | appfun.adwo.com |
| 520apk | 13,048 | www.520apk.com |
| lenovo | 151,426 | www.lenovomm.com |
| baidu | 30,275 | shouji.baidu.com |
| jifeng | 30,661 | www.gfan.com |
| yingyongbao | 5,184 | sj.qq.com |
| hiapk | 76,066 | www.hiapk.com |
| gezila | 11,030 | www.gezila.com |
| xiaomi | 63,494 | app.mi.com |
| yy138 | 5,073 | www.yy138.com |
| liqucn | 10,134 | www.liqucn.com |
| angeeks | 54,432 | www.angeeks.com |
| 3533 | 15,871 | www.3533.com |
| apk91 | 27,190 | zs.91.com |
| nduo | 8,965 | www.nduo.cn |
| 1mobile | 16,659 | www.1mobile.com |
| sougou | 27,795 | zhushou.sogou.com |
| anzhi | 401,578 | www.anzhi.com |
| anzow | 12,521 | www.anzow.com |
| zs2345 | 4,538 | zs.2345.com |
| 7xz | 4,871 | www.7xz.com |
| huawei | 6,804 | app.hicloud.com |
| 16app | 38,003 | www.16apk.com |
| apk3310 | 22,376 | apk.3310.com |
| appchina | 244,413 | www.appchina.com |
| *others* | 62,216 | (Appendix A) |
| **total** | 1,427,395 | |

is considered to represent a third-party library only if the number of instances in this group is greater than or equal to the threshold. Details about how the threshold is decided are presented in §IV-B.

We label each cluster with the topmost level package name. If a cluster contains instances with different names, the label is set to be the name carried by most instances. We further merges clusters with the same label. Such merged clusters indicate libraries with different mutations.

## IV. Evaluation

### A. Dataset

To evaluate LibD, we crawled 1,427,395 Android apps from 45 third-party markets. Although the official app market, Google Play, hosts over a million apps [2], it conducts rigorous reviews on submitted apps, including both static and dynamic analysis. Presumably, many malicious or vulnerable third-party libraries may get rejected during the review process [21], [22]. On the other hand, third-party markets usually do not have such review processes, and we expect to collect more diverse library instances. As we will show with experiment results, we successfully detected a large number of library mutations, many of which are obfuscated, with our current settings. In addition to well established third-party markets, we also crawled apps from popular Android forums. Table I lists the sources from which we collected apps and the number of samples crawled from each source.

### B. Clustering Threshold

As mentioned in §III-E, LibD needs a hyper parameter named the clustering threshold to decide whether a group of library instances is large enough to represent a third-party library. We investigated how different values of the clustering threshold can affect the number of detected libraries and then tried to set a reasonable threshold by referring to related work.

TABLE II: Third-party libraries detected with different clustering threshold settings.

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

Previously, different thresholds have been used to cluster libraries. For example, Wukong [7] set the threshold as 32 while Li et al. used 10 [9]. To search for the best option empirically, we iterated different threshold values from 10 to 50 and recorded the number of libraries detected with each of the thresholds. Recall from §III-E that LibD clusters library instances with the same feature into groups, and each group is considered to be a library mutation. If some mutations share the same topmost level package name, they are identified as different versions of the same library. Naturally, with a larger clustering threshold used, the number of detected libraries and mutations decreases.

Table II shows the detection results regarding different threshold settings. As can be seen, even with a relatively large value of the threshold, i.e., 50, LibD can detect a large number of libraries (over 2,000).

Comparing LibD with previous work, we report that LibD can detect more third-party libraries than the both whitelist and system API-based methods [4], [7]. We will present further discussions in §IV-D.

### C. Threshold Sensitivity

Since the analysis results of LibD depend on the clustering threshold as a hyper-parameter, we conducted a sensitivity analysis to investigate how different threshold values can impact the number of detected libraries and whether the stability of threshold values is correlated to factors, e.g., the sample size. For this purpose, we break down the numbers in Table II and present in Fig. 4 the statistics of each individual market included by our dataset. The decomposed results are grouped by the size of each market.

Overall, the quantities of detected libraries are mostly stable when the threshold is no less than 25, for all sizes of markets. After threshold drops below 25, there is a sharp increase of the numbers of positives for many markets. This phenomenon occurs in most size groups, indicating that the sensitivity of threshold values is unlikely to be dependent on the total counts of analyzed apps, as long as the sample size is of a market scale. Additionally, Fig. 4f compares the sum of numbers of detected libraries found in each market with the total count of positives found when processing all samples as a whole dataset. As can be seen, LibD reported significantly fewer libraries when analyzing the entire dataset, suggesting that many

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 | | |

libraries are shared across different markets. This is another indicator showing that the threshold is likely independent of the analyzed market and may be determined with a set of universally effective criteria.

According to the analysis above, we propose a method to determine the threshold with the goal of minimizing false positives. In previous work, Chen et al. [4] released a whitelist including 72 commonly used third-party Android libraries. We matched this list with libraries detected with different thresholds (Table II), recording the count of missed libraries in Table III. We found that until the threshold decreases to 10, LibD can exhaust *all* libraries in the whitelist. *Consequently, our subsequent experiments used 10 as the threshold value unless otherwise noted.*

In theory, applying obfuscation to the dataset is likely to have an impact on the determination of an optimal threshold, if this cannot be effectively nullified by the detection technique. Take the whitelist-based detection for example. When the names of some library instances are turned into random strings, they are no longer on the whitelist, shrinking the sizes of the clusters they should have belonged to. Consequently, some clusters may no longer be large enough to be recognized as a library. However, simply raising the threshold does not completely solve the problem, because it may lead to increased false positives. In general, tuning the clustering threshold only is not sound enough to achieve obfuscation-proof detection.

### D. Comparison with Other Work

*1) Comparing with LibRadar:* To test the effectiveness of LibD, we setup an in-lab examination by comparing LibRadar [8] and LibD. LibRadar is the successor of the library identification component of WuKong [7], a Android clone detection tool. In the original publication introducing WuKong, the authors reported that they detected more than 10,000 third-party library variants among 105,299 applications crawled from 5 different markets, using 32 as the clustering threshold. With the same threshold, LibD detected 16,074 variants in our dataset. Since the two datasets are different, the results are not directly comparable. Therefore, we designed new experiments to compare LibD with LibRadar, which is now accessible as an online service. Different from LibD, which takes the opcode sequences of basic blocks as the primary feature to build the profile of a library candidate, LibRadar uses a more coarse-grained feature, i.e., the frequency of different Android API calls. Also, LibRadar only considers the package inclusion relation when constructing library candidates, while LibD takes into account additional relations based on code semantics (§III-B).

To our best knowledge, there is no systematic approach to acquiring the ground truth about the presence of third-party libraries in Android apps, since the boundary of a library is essentially only known to developers. To obtain results that are close to ground truth, manual inspection on the samples is necessary.

TABLE IV: Comparison with LibRadar. Results validated with 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 |

For a convincing and feasible evaluation, we randomly collected 1,000 apps from our dataset as a subset and manually investigated the subset to get the ground truth. We identify third-party libraries according to the following conditions.

- If the library name represents a legal domain name, then it is considered to be a third-party library.
- If the package name is a subdomain of a legal domain name, we will then query search engines (e.g., Google) with the complete name and see whether the query leads to some Android library vendor. If so, we conclude that a new library instance is found.

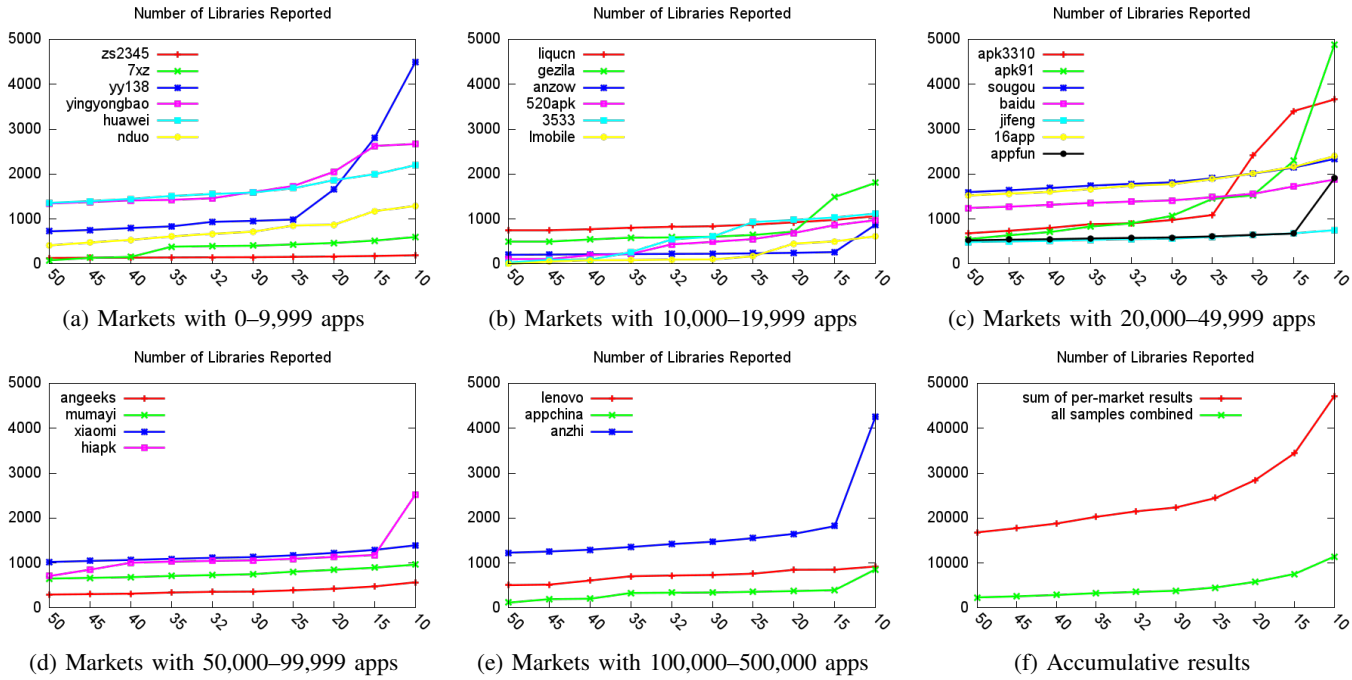
Following the strategy above , we acquired 2,613 libraries as the ground truth from the 1,000 apps. We then run LibD and LibRadar [8] over these apps. LibRadar [8] provides an online service to detect libraries. This enables convenient comparison with LibD. Note that LibRadar only provides the names of the detected libraries.

Table IV presents the performance of LibRadar and LibD. We also validate the detected libraries according to the ground truth. Overall, LibD identified 1,954 libraries, among which 1,456 libraries are true positives. LibRadar found 670 different libraries in total and 264 of them are true positives. In terms of false positive and negative rates, LibD can notably outperform LibRadar.

*2) Comparing with Li et al. [9].:* To avoid confusion caused by package and symbol name obfuscation, LibD relies on low-level program features like type hierarchies and opcode sequences to find and match similar app components. This may lead to overly restrictive similarity matching, leading to more false negatives. We therefore compare LibD with methods based on less fine-grained features of the apps.

The research conducted by Li et al. detected 1,113 libraries from about 1.5 million apps in Google Play [9]. Their approach operates at the package level and relies on package names to cluster library candidates in the first place. To counter the package name obfuscation problem, they excluded all packages with single character names from their detection. To discriminate libraries that happen to posses the same name, they refined each cluster by comparing the prototypes of methods in a package.

The implementation of the method developed by Li et al. is not available. The authors provided the names of the detected libraries which are verified through manual inspection. It is unclear what results their method would produce given our dataset. Therefore, we cannot assess the accuracy of the method in terms of false positive and false negative rates, meaning a comprehensive comparison like we did for LibRadar is not feasible. We hereby present a best-effort comparison by inspecting the difference between their results




(a) Markets with 0–9,999 apps (b) Markets with 10,000–19,999 apps (c) Markets with 20,000–49,999 apps

(d) Markets with 50,000–99,999 apps (e) Markets with 100,000–500,000 apps (f) Accumulative results

Fig. 4: Number of libraries reported by LibD with different thresholds

and the libraries detected from our whole dataset. Among the 1,113 library names reported by Li et al., 262 are not included by the results of LibD. Further inspection shows that 249 of these 262 libraries are indeed *highly correlated* to some of our results in the sense that there exist LibD-detected libraries with different but very similar package names and structures. For example, `com/comScore/exceptions` and `com/comScore/stramsense` are two libraries reported in their list, which did not appear in our results. However, LibD reported a library named `com/comScore`, which happens to be the union of `com/comScore/exceptions` and `com/comScore/stramsense`. We speculate that these 249 libraries were missed by LibD likely due to different heuristics used for recognizing library boundaries.

As for the remaining 13 library names missed by LibD, we cannot relate them to any of our results. Since no information other than library names are known to us for in-depth analysis, we can only roughly infer the reasons behind the differences. Note that the library names reported by Li et al. were obtained by analyzing apps crawled from Google Play, which is not available in China. It is known that many Android developers customize their products according to regions where apps are to be published. Since there are many third-party services that are not available in China but accessible from other regions, libraries providing these services are naturally absent from our dataset. For example, we noticed that several Google Play apps include a library named `com/android/psu`. By enumerating all package names existing in the dataset, we confirmed that packages with this name are not contained by any of our samples.

Overall, LibD can identify a reasonably large portion of the libraries detected by the method proposed by Li et al. The results indicate that although our method relies on fine-grained program features like type hierarchies and opcode sequences for similarity measurement, the matching process is unlikely to be overly restrictive, compared to methods based on higher-level features like package names and method prototypes.

### E. Resilience to Obfuscation

One of the major advantage of LibD over previous tools is its capability of detecting and identifying obfuscated libraries. To justify the claim, we evaluated LibD's resilience to obfuscation algorithms in two different aspects. Firstly, we tested LibD on a set of open source apps obfuscated by ProGuard [23], the official obfuscation toolkit of Android development. As for the second aspect, we zoomed in the comparison between LibD and LibRadar, which is presented above, and particularly studied the performance of the two tools in processing obfuscated app samples.

The open source apps were randomly sampled from F-Droid, an Android app market that contains only free software. With access to the source code, we can compile the apps with obfuscation explicitly enabled by ProGuard. Our experiments used the default obfuscation setting of ProGuard. With this setting, all packages, classes, and class members receive *new short random* names [24]. Advanced obfuscations like package hierarchy flattening was disabled. Resilience to obfuscation is demonstrated by observing LibD's accuracy for obfuscated and unobfuscated samples of the same apps. A total of 100 apps were sampled from F-Droid and manually investigated to obtain the ground truth about the libraries included. Table V shows the results of library detection. Due to the relatively small sample size in this experiment, we set the clustering threshold to 1, meaning every candidate library instance is reported to a library.

As can be seen from the results, the number of libraries included by F-Droid apps are notably low in contrast to apps that we crawled from the commercial markets. On average,

TABLE V: Library Detection for 100 F-Droid Apps

| | Unobfuscated | Obfuscated |
|---|---|---|
| **# of library instances** | 159 | 159 |
| **# of reported library instances** | 152 | 154 |
| **# of false positives** | 6 | 8 |
| **# of false negatives** | 13 | 13 |

an F-Droid app contains only 1.6 libraries, while apps in the original dataset contains over three libraries. Nevertheless, LibD accurately captured this difference and reported very small proportions of false positives and false negatives when the apps were free of obfuscation. After obfuscation, the number of reported false positives slightly increased, while the number of false negatives remained the same. Overall, the accuracy of LibD is not significantly affected by the obfuscation from ProGuard, showing its resilience to common Android anti-analysis techniques.

To additionally show that our tool is proficient at handling obfuscated Android apps, we compared LibD with LibRadar, which is also designed to be resilient to renaming-based obfuscation to a certain extent.

We randomly sampled 100 obfuscated apps from the original dataset (Table I) and the performance of LibD and LibRadar regarding this subset of apps. Among these 100 apps, LibRadar found 13 obfuscated libraries, while LibD found 14. Both LibRadar and LibD are able to cluster the obfuscated and obfuscated versions of the same libraries into the same clusters. The difference between the results are due to different features used for determining library similarity, which is not directly related to obfuscation resilience. 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 rate is that it build library signatures using the set of system APIs used by the code, which is not fine-grained enough to distinguish some similar but different libraries. Overall, the results indicate that the LibD's resilience to renaming-based obfuscation is at least as strong as LibRadar.

### F. 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 operating 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.

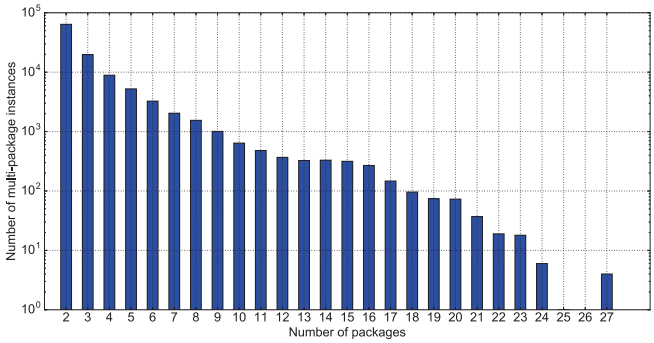



Fig. 5: Distribution of different multi-package library instances regarding the number of packages.

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.

### G. 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 10, we report to find 5,141 multi-package libraries (8.4% of all the detected libraries in total).

We also use all the different library instances for evaluation, as they can reveal potential rare changes on the third-party libraries. Fig. 5 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 63,948 two-package instances (58.8% of all the multi-package instances). We manually analyzed 10 frequently occurred 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. 6, while there are 51,099 instances with only one structure, 67,147 instances actually contain more than two different structures

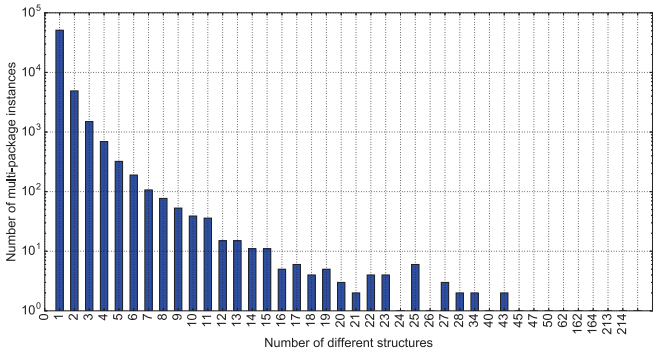



Fig. 6: Distribution of different multi-package library instances regarding the number of structures.

TABLE VI: Five shared packages and evolved libraries.

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

(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 VI presents 5 packages that are shared by at least two libraries. Given the observation that the first two 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 a total of 18,594 different kinds of first two segments in the outputs of LibD; we list the top ten in Table VII. Note that if the first two segments of two library names are identical, they are likely from the same developers. In other words, Table VII 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 [23]) replace the library name with several meaningless strings while preserve the original directory structures. Our experiment 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

TABLE VII: Top ten most commonly encountered initial 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 |

TABLE VIII: 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 |

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 10, LibD can detect 11,458 different libraries in total. With our manual 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 VIII 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 quite common in real-world Android applications.

*3) Library Mutations:* In this section we study the library mutations. In general, our experiment 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 IX: 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 |

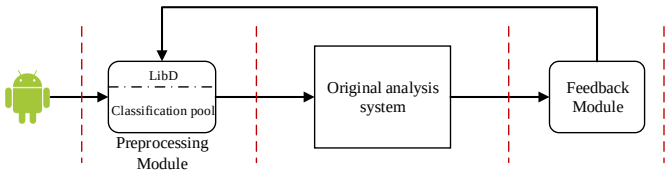



Fig. 7: Acceleration workflow overview.

Table IX 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-G). 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 IX) 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. Boosting Vulnerability Analysis with LibD

### A. Motivation

For most existing Android app analysis tools, the input apps are analyzed as a whole, namely the tools do not distinguish app-specific code and third-party code. Potentially, this is a huge waste of resources when the analysis is conducted at a large scale, since a third-party library can be shared by many different apps, while repeatedly analyzing the same library does not bring any additional benefits. Moreover, the amount of redundant analysis on the same chunk of code increases as more apps are analyzed and more third-party libraries are encountered. During the evaluation of LibD, we found that an Android app on average contains three third-party libraries, which reveals the opportunity to speed up large-scale Android app analyses.

To demonstrate that Android app inspection can be significantly boosted by pruning the analysis of redundant libraries, we propose to integrate LibD into an existing Android analysis system. For this purpose, the threshold for library detection is adjusted to *one* such that all repetitive code blocks can be identified (Section IV-B).

### B. Overview

Conceptually, a cache system is added to help an Android analyzer to memorize the analysis results about a previously encountered code block. To this end, we add a preprocessing module and a feedback module to the original analyzer. The modified analysis workflow is illustrated in Fig. 7.

The preprocessing module is used to generate and store signatures and analysis results for each code block from the input apps. LibD serves as the major component of the preprocessor and it dissects the input apps into different parts,

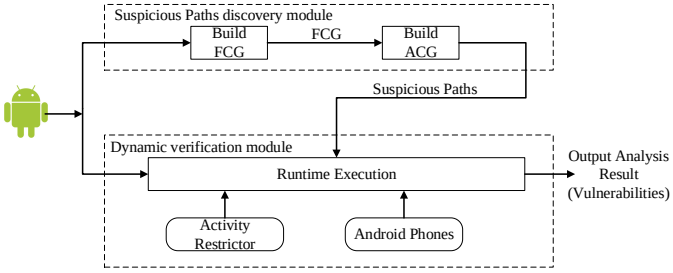



Fig. 8: The original SmartDroid analysis workflow.

with the signature of each part stored in the database. If the signature hits a record in the database, the preprocessing module will advise the analyzer to skip the analysis of the corresponding code block and the analysis result is directly retrieved from the database. If there is no match for the signature in the current database, the analyzer will perform analysis towards the code block. The result is forwarded from the feedback module to the preprocessor and the database will be updated with an new entry composing the newly-analyzed basic block and the analysis result.

### C. Analysis System Selection

To evaluate the acceleration scheme, we implemented it on top of SmartDroid, a well-developed Android vulnerability analysis system [17]. SmartDroid takes a two-step approach to identify potentially vulnerable Android apps statically and further verifying the detection results dynamically.

Before we describe how we boost the performance of SmartDroid with LibD, we first briefly introduce how SmartDroid works. Fig. 8 shows the workflow of the original SmartDroid system. The static analysis module of SmartDroid is responsible for pinpointing suspicious vulnerable code paths with user-provided data flow footprints as signatures. In this module, SmartDroid first extracts the functions from the app and builds the call graph. This graph is named as FCG (Function Call Graph). Based on the FCG, SmartDroid builds a higher level graph, ACG (Activities Call Graph), to represent the logic connections between different activities. SmartDroid then traverses all the traces in the ACG and compares them with predefined vulnerable traces. Once a trace in the ACG matches any of the predefined vulnerable traces, this trace will be marked as vulnerable. After the traversal is finished, SmartDroid constructs a set of potentially vulnerable paths. These paths are later verified, by a verification module of SmartDroid, through dynamic analysis on the Android emulator to check whether they are feasible and indeed vulnerable.

While SmartDroid is capable of identifying vulnerable apps, there are two limitations that potentially impede its overall analysis power towards large sets of Android apps. First, SmartDroid only provides very coarse-grained analysis results, i.e., whether an app is vulnerable or not. In other words, it cannot pinpoint which exact code blocks contain the vulnerability. In addition, when analyzing a large number of apps, SmartDroid will repeatedly analyze the same libraries in different apps, leading to a waste of computation resources.

Based on our research, SmartDroid can be enhanced to directly pinpoint vulnerable code blocks within a vulnerable

app. In this paper, a code block is defined as *a library instance or a code component consisting of one or multiple functions that provides a complete implementation of some specific functionality.* In our improved vulnerability detection scheme, the preprocessing module, which employs LibD to dissect the apps, is configured with a threshold of one to cover all code blocks in an app.

With this improved scheme, we expect that the analysis efficiency will be boosted. Recall that our analysis has shown that a lot of third-party libraries are integrated into the apps. Such repeated code blocks will cause the original SmartDroid system to repeat the vulnerability analysis towards the same code block multiple times, and the repeated analysis would waste considerable amount of computing resources and time.

As our library detection approach is able to rule out libraries from apps, the new detection scheme can avoid much of the redundant analysis. When an app is sent to SmartDroid, we try to prune the previously analyzed libraries and only deliver the fresh code components to the detection system.

However, this improvement over SmartDroid introduces a new problem that potentially affects the effectiveness of the detection process. As previously mentioned, SmartDroid verifies static detection results through dynamic analysis. To do that, SmartDroid will need to construct traces that can reach the vulnerable program points. After our modification, traces extracted by SmartDroid are confined within individual code blocks instead of whole apps. Due to the event-driven programming paradigm of Android apps, many execution paths cannot be statically captured. Therefore, there is a possibility that traces extracted from single code blocks lack certain prefixes or suffixes[1] and cannot be dynamically verified.

To estimate the impact of this problem, we randomly sampled 1,000 apps to get the proportion of apps whose vulnerabilities are enclosed by a single code block after an app is dissected by LibD. The analysis was conducted in the following steps:

- Use SmartDroid to statically analyze the 1,000 apps, targeting four kinds of vulnerabilities, i.g., DoS, WebView leak, SSL Hijacking, and FileCross. *Details about these vulnerabilities are presented in §VI-A.*
- For those that SmartDroid considers vulnerable, ask SmartDroid to further construct execution traces that can potentially trigger the vulnerabilities.
- Use LibD to dissect the 1,000 apps into code blocks, which are essentially library candidates.
- For each constructed execution trace, examine if it flows from one code block to another. This step was manually performed by four researchers familiar with SmartDroid.[2]

With the procedure described above, SmartDroid reported 1,390 vulnerable program points and LibD dissected the 1,000 apps into 3,330 code blocks. The manual inspection confirmed that all execution traces constructed by SmartDroid are contained within a single code block. Therefore, we expect that our acceleration scheme will not have a significant impact on the accuracy of SmartDroid.

[1]For some vulnerabilities, e.g., the DoS vulnerabilities, the program point that triggers the error is not where the failure manifests. In such occasions, a vulnerability cannot be verified until the execution reaches the manifest point.

[2]All participants were affiliated with Chinese Academy of Sciences, three of whom are not the authors of this manuscript.

### D. Acceleration

In order to accelerate SmartDroid, we add a preprocessing module and a feedback module to the original SmartDroid system. The preprocessing module is used to first dissect the integrated apps into code blocks. After that, each block is analyzed separately by the original part of SmartDroid and assigned a label $L_{sec}$ indicating the analysis result. A label is either $SECURE$ or the vulnerability type detected in this block. An empty label value $NULL$ indicates that the block is not analyzed yet.

After SmartDroid finishes analyzing the code blocks and verifying the results through dynamic analysis, the feedback module will update the analysis results of these blocks stored in the preprocessing module. Next time the same code block is encountered, the preprocessing module will skip the analysis and reuse the previously obtained results. Algorithm 3 and 4 describes the processes of these two modules, respectively.

**Algorithm 3:** Preprocessing Algorithm

**Input:** Android app data set $S_{app}$
**Output:** Classification pool $S_{cbs}$ and bidirectional block-to-app mapping $M$

$S_{app} \leftarrow$ Android apps data set;
$S_{cbs} \leftarrow \varnothing$;
$M \leftarrow \varnothing$;
**foreach** $app \in S_{app}$ **do**
  $S_{inst} \leftarrow Func_{inst}(app)$; /* $Func_{inst}$ invokes LibD to dissect the app*/
  **foreach** $block \in S_{inst}$ **do**
    **if** $block \notin S_{cbs}$ **then**
      add new relations $(app, block)$ and $(block, app)$ into $M$;
      Add $block$ into $S_{cbs}$ with its $L_{sec}$ as $NULL$;
**return** $(S_{cbs}, M)$

In Algorithm 3, LibD is employed to dissect an input app into blocks. We also build the mapping between apps and blocks, which is a many-to-many relation, i.e, each app contains more than one code block, and the code block instances, especially the third-party libraries, can also be shared among different apps. This mapping makes it much more convenient to trace the apps from the instances and vice versa. We then insert these fresh instances into our classification pool. For each newly extracted instance, we first check if it has been recorded already. If a block instance is never analyzed before, we create a fresh record in the pool. The output of this algorithm is a set of blocks $S_{cbs}$ and the bidirectional mapping $M$.

Algorithm 4 illustrates the updating process of the classification pool, where the analysis results from Smartdroid are used to updated the security labels of blocks in the classification pool. As the analyzed code accumulate, and the per-app analysis process will speed up since repeatedly appearing code

**Algorithm 4:** Feedback Algorithm

**Input:** classification pool $S_{cbs}$ with unanalyzed blocks
**Output:** updated $S_{cbs}$

```
foreach w ∈ S_cbs do
    if L_sec of w is NULL then
        sec_label ← Func_smartdroid(w);
        update L_sec of w to be sec_label;
return updated S_cbs
```

blocks will not be re-analyzed and the previously generated results will be reused.

### E. Implementation

Fig. 9 presents the modified SmartDroid system. The preprocessing and feedback modules are used to instrument the inputs and outputs of the SmartDroid system.

According to Algorithm 3, preprocessing module uses LibD to dissect Android apps into code blocks and inserts each of the block into a classification pool. This pool is divided into three areas: the secure block area, the vulnerable block area, and the unanalyzed block area. As shown in Fig. 9, SmartDroid is directed to only focus on the unanalyzed blocks. For this purpose, we modified the SmartDroid system to change its input format. In the original SmartDroid system [17], the input is a complete Android app, after our modification the input becomes individual code blocks.

The general workflow of our modified system is mostly identical to that of the original system. We first transmit the unanalyzed blocks to the static analysis module of SmartDroid. The system will collect all the suspicious paths in the code blocks by comparing all the paths of the ACG with the predefined vulnerable traces. After that, these suspicious paths are sent to the dynamic verification sub-module. This sub-module executes each of the suspicious paths to verify if these paths can trigger the flaws in reality. If one path is able to make any of the known flaws happen during the verification, this block is considered as vulnerable and its corresponding record would be updated in the following feedback module.

### F. Acceleration Evaluation

We deployed our modified detection system on the OpenStack Platform and leveraged 100 virtual machines to analyze apps in parallel. In general, we evaluate our acceleration scheme by measuring (1) the vulnerability discovery accuracy and (2) the acceleration efficiency in terms of the processing time.

*1) Accuracy:* Essentially, four kinds of widely existing flaws, DoS, WebView leaks, SSL Hijacking and FileCross, are studied in this step. The details of these vulnerabilities are presented in Section VI. Considering SmartDroid as a well-developed tool for vulnerability detection, we take the detection result of the original system towards our app data set as the baseline to assess the accuracy of our modified system; we measure the false positive and false negative rate in this step. Table X reports the comparison result. Here, we list the vulnerable apps detected by both systems. Note that since the same vulnerability assertion approach is deployed in both systems, for a benign app, our modified system should not mark it as "vulnerable". In other words, we expect that no false positives should be reported. Our evaluation result is consistent with this intuition, as listed by Table X.

Comparing with the detection results of the original SmartDroid system, we report that most of the false negative rates are negligible (half of them are zero), and the modified SmartDroid system is still well-performing in detecting vulnerable Android apps.

To understand what factors have led to the errors, we sampled some of the false negatives and studied them case by case. For each vulnerability class, we randomly selected five false negatives. For classes with less than five false negatives, we took all available cases. As such, a total of 38 false negatives were analyzed in depth. Through the analysis, the main cause of false negatives is the potentially incomplete execution traces fed to the dynamic verification module, as discussed in §V-C. Among the 38 cases we analyzed, 33 are due to the lack of necessary prefixes, i.e., the vulnerabilities could not be triggered, and 5 are due to the lack of suffixes, i.e., the failures failed to manifest.

*2) Acceleration Effects:* To evaluate the processing speed increase of the modified system, the accumulative time consumption as well as the number of extracted suspicious paths are studied in this section.

The time consumption is a key criteria to measure the efficiency of the system. In general, the original SmartDroid system takes five minutes to analyze one app on average. The static analysis of suspicious path extraction takes around one minute while the dynamic verification step takes about four minutes. While the dynamic verification step cannot be optimized, as aforementioned, the modified system shall effectively reduce the number of suspicious paths in each app that need to be verified.

Since the modified system "caches" the analysis results, we expect that the analysis speed would be constantly reduced by the accumulation of analyzed apps. We fed the modified system with in total 1,427,395 Android apps and measured the processing time after analyzing certain amount of cases. The results are presented in Table XI, which are consistent with our expectation.

As reported in Table XI, the original and the modified systems spend about the same amount of time in analyzing the first 100 apps. However, the modified system becomes about 6 minutes faster when 1,000 apps have been fed. This "gap" keeps growing as more apps are analyzed. In total, the original system takes more than 27 days to finish all tasks. That is, the modified system saves almost one month to analyze the data set comparing with the original analysis system.

We also evaluated the efficiency of the modified system by training from the 1,427,395 apps data set and perform cross data set validation. To this end, we additionally collected 370,507 apps which are not in the original dataset used for evaluation We then randomly selected 1,000 apps from new dataset and run both analysis systems to record the total analysis time. On average, the original system can process 15 apps per minute while our modified system can handle

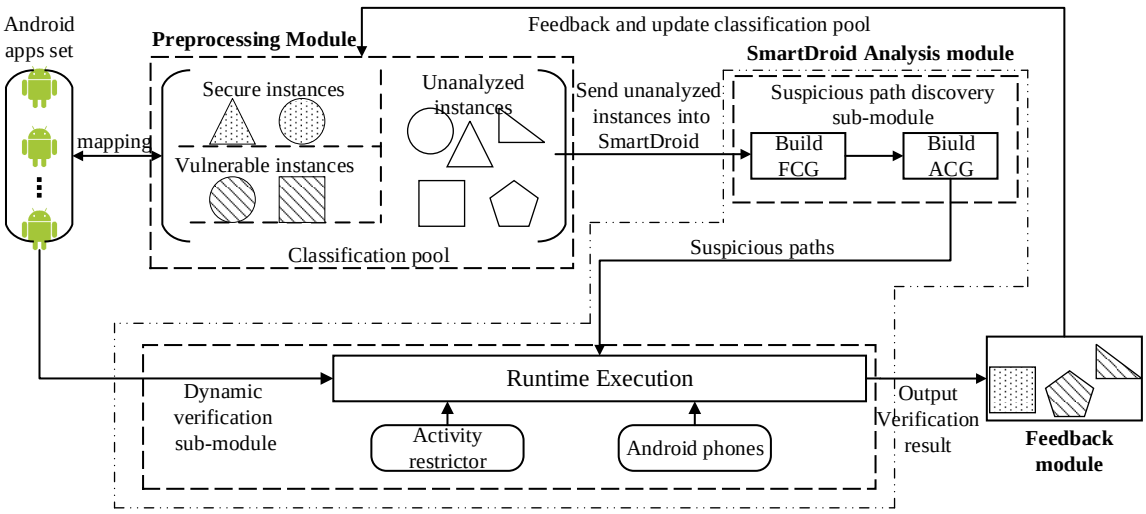



Fig. 9: Workflow of the modified SmartDroid system.

TABLE X: Comparing the detection results of the modified SmartDroid with those of the original.*

| Type of Vulnerabilities | Vulnerable Apps Detected by SmartDroid | Vulnerable Apps Detected by the Modified System | False Negatives (False Negative Rate) |
|---|---|---|---|
| DoS | 254,209 | 254,135 | 74 (0.03%) |
| WebView leaks | 3,099 | 3,099 | 0 (0.00%) |
| SSL Auth Flaw | 44,512 | 44,506 | 6 (0.02%) |
| FileCross | 18,490 | 18,490 | 0 (0.00%) |
| DoS & WebView leaks | 1,522 | 1,522 | 0 (0.00%) |
| DoS & SSL Auth Flaw | 1,692 | 1,635 | 57 (3.37%) |
| DoS & FileCross | 2,173 | 2,173 | 0 (0.00%) |
| SSL Auth Flaw & WebView leaks | 92 | 92 | 0 (0.00%) |
| SSL Auth Flaw & FileCross | 50 | 47 | 3 (6.00%) |
| WebView & FileCross | 12,858 | 12,753 | 105 (0.82%) |
| Dos & SSL Auth Flaw & WebView leaks | 46 | 46 | 0 (0.00%) |
| DoS & SSL Auth Flaw & FileCross | 18,218 | 18,218 | 0 (0.00%) |
| DoS & WebView leaks & FileCross | 38,865 | 37,979 | 886 (2.28%) |
| SSL Auth Flaw & WebView Leaks & FileCross | 2,181 | 2,102 | 79 (3.63%) |
| DoS & WebView Leaks& SSL Auth Flaw & FileCross | 10,496 | 10,490 | 6 (0.06%) |

**The false negatives are counted assuming the results reported by the original SmartDroid are correct. No false positives were reported.*

TABLE XI: Comparison of the processing time.

| # of Analyzed Apps | Total Time Consumption of the Original System | Total Time Consumption of the Modified System |
|---|---|---|
| $\leq$ 100 | 5 min | 6 min |
| $\leq$ 1,000 | 61 min | 55 min |
| $\leq$ 10,000 | 511 min | 483 min |
| $\leq$ 100,000 | 5,772 min | 4,434 min |
| $\leq$ 1,000,000 | 51,365 min | 31,234 min |
| $\leq$ 1,427,395 | 72,504 min | 32,789 min |

TABLE XII: Comparison of the analyzed suspicious paths.

| | Suspicious Paths Analyzed in the Original System | Suspicious Paths Analyzed in the Modified System |
|---|---|---|
| # of Suspicious Paths in Total | 44,289,364 | 6,317,167 |
| # of Average Suspicious Paths in Each App | 120 | 17 |

100 apps per minute, namely 5.5 times of efficiency increase. Note that in this new experiment, we did not feedback the analysis result to the preprocessing module.

Overall, 19,269,833 blocks were dissected from our data set of 1,427,395 apps. Our study shows that 17,048,455 blocks are reused during the evaluation. On average, an app contains 13 blocks and most of them (11 blocks) are shared among different apps. Therefore, we expect that most app analysis tasks can be accomplished by only analyzing small amount of blocks in each app.

In addition to the processing time study, we also evaluate the number of suspicious paths that need to be analyzed in both systems. Table XII shows the suspicious paths that are labeled in both systems. In total, there are 44,289,364 suspicious paths in our 1,427,395 apps data set; each app contains 120 suspicious paths on average, but most of them are shared among different apps. As aforementioned, most dynamic verification in the original system repeatedly process already-analyzed paths. On the other hand, after we optimized the system by caching the fine-grained analysis results, the number of suspicious paths has been reduced to 6,317,167. For each app, there are only 17 suspicious paths for analysis on average. As a result, our modification reduces the analysis efforts to 14.2% by squeezing out the redundancies.

## VI. Improving Vulnerable Library Detection

Usually, the code containing programming flaws and making a library vulnerable is only a small part of the entire library. For example, the vulnerable part of Baidu Moplus SDK [20] is only a single-line code snippet. As such, the vulnerable and benign versions may be very similar in terms of both the

structure and functionality, making it non-trivial to identify the vulnerable versions.

One of the design goals of LibD is to find the sweet spot for the sensitivity of library signatures. On the one hand, we would like the signature to be sensitive enough to reflect subtle changes made to the library code such that the different versions of the same library can be effectively distinguished. On the other hand, we also want to avoid designing overly fine-grained signatures to keep the computation of signatures efficient enough for large-scale analysis. In this section, we demonstrate that LibD can be used to detect the vulnerable variants of the same Android library among millions of apps.

### A. Inspected Vulnerabilities

To evaluate if our approach is sensitive enough to identify common programming flaws in vulnerable third-party libraries, we choose four widely-spread types of Android vulnerabilities, including deny of services (DoS), WebView information leaks, man-in-the-middle (MITM) SSL hijacking, and the "FileCross" problems affecting Android browsers.

- *DoS*. An attacker using Deny of Service flaws can cause a running computer or server to crash, for example, by exploiting the overflows in memory. On the Android platform, the type of attacks can crash the smartphone. With a proper use of the integer underflow (e.g., CVE-2017-14496), an adversary is able to deploy a remote DoS attack. In the CVE database[3] platform, the DoS vulnerabilities account for 18.6% of all the uploaded vulnerabilities. This type of flaws pervasively exists in the different versions of the Android platform.
- *WebView Information Leaks*. WebView is an important gadget in the Android Framework that is responsible for rendering web contents. In some Android versions, the WebView component is buggy and may cause information leaks when programmed in certain patterns (e.g., CVE-2014-6041). We summarized these potentially vulnerable code patterns and try to identify them in our data set.
- *SSL Hijacking*. SSL hijacking has become a common security flaw in mobile apps in recent years [25]. The root cause of this flaw is that mobile apps fail to check the SSL certificates of the servers they communicate with, thus vulnerable to man-in-the-middle attacks when the mobile devices are connected to untrusted networks. In our evaluation, we focus on detecting inappropriate implementations of HTTPS communications in Android apps.
- *FileCross*. Android browsers support the URI scheme of `file://`. FileCross refers to the vulnerabilities that exploit the file access URL to inject malicious JavaScript code into the file system and steal on-device data [26].

### B. Detecting Vulnerable Apps

Again, we employ SmartDroid [17] as the vulnerability detector to obtain the ground truth about whether an Android app is vulnerable or not. As previously mentioned, SmartDroid can only detect vulnerabilities at the granularity of apps but not third-party libraries. Our modification of the original analysis system improved the situation by narrowing the vulnerable scope from the entire app to code blocks.

[3]http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224

TABLE XIII: Vulnerable Libraries

| # types of vulnerabilities | # Vulnerable libraries |
|---|---|
| DoS | 2,659 |
| WebView leaks | 2,141 |
| SSL Auth Flaw | 2,020 |
| FileCross | 311 |
| DoS & WebView leaks | 27 |
| DoS & SSL Auth Flaw | 865 |
| DoS & FileCross | 1,428 |
| SSL Auth Flaw & WebView leaks | 23 |
| SSL Auth Flaw & FileCross | 22 |
| WebView & FileCross | 867 |
| DoS & SSL Auth Flaw & WebView leaks | 12 |
| DoS & SSL Auth Flaw & FileCross | 233 |
| DoS & WebView leaks & FileCross | 42 |
| SSL Auth Flaw & WebView Leaks & FileCross | 35 |
| DoS & WebView Leaks& SSL Auth Flaw & FileCross | 116 |
| **Total** | 10,801 |

### C. Evaluating Vulnerable Library Detection

The result of the modified vulnerability analysis system shows that our third-party library detection approach is sensitive to vulnerable libraries. All the vulnerable libraries are real and may threaten massive apps at the same time.

As is shown in Section IV, we found a considerable number of different versions of the same library, while only some of them are indeed vulnerable. In total, we detected 10,801 vulnerable third-party libraries with a clustering threshold of 10 (see §IV-C for threshold selection). The vulnerable libraries account for 17.7% of all libraries, indicating that third-party libraries are not as secure as assumed by some previous research. Threats of defective third-party libraries needs to be thoroughly considered.

In our case study, we noticed that a library can be affected by multiple vulnerabilities. Table XIII shows the statistics about this phenomenon. According to our results, 34.0% of the vulnerable libraries contains more than one vulnerability. For these libraries, the most common vulnerability combination is DoS and FileCross. This is an alarming fact since the two defects can easily form a realizable attack sequence. Attackers can first exploit FileCross to deploy and execute malicious JavaScript code on a victim device; the malicious JavaScript can then easily trigger the DoS attack.

### D. Further Analysis

To obtain a deeper understanding on the characteristics of detected vulnerable libraries, we further manually inspected the top 200 most popular ones, ranked by the number of apps affected. Table XIV and Table XV list the separated rankings of the top 10 most popular obfuscated and unobfuscated vulnerable libraries, respectively, showing library names, the count of apps including them, and the types of vulnerabilities reported. The 200 analyzed libraries cover all those listed in Table XIV and Table XV. For each manually inspected library, we randomly picked an instance from each corresponding cluster.

We designed a systematic protocol to manually inspect the library instances covered by Table XIV and Table XV. For

the obfuscated libraries, the two research questions to answer are: *"what are their identities?"* and *"what are their main functionalities?"*. We developed the answers mostly through reverse engineering. The typical procedure for revealing the true identity of each library instance is to dump all string literals used by the library code and see if any of them indicates library name. For example, many libraries produce logs during execution and prepend their names to each log entry. Regarding the second research question, for most obfuscated libraries, it is easy to learn their functionalities from the Internet once the library name is revealed. Rarely, for those that lacks online information, we needed to additionally analyze their decompiled code. For unobfuscated libraries, our primary goal was to confirm their identities and investigate whether they were ever published by a third-party library developer and provided to other app vendors. If so, we would collect all historical versions of that library published by its developer and see if the instance in our dataset matches any of these versions.

Two authors participated in the manually analysis. In order to avoid biases caused by different personal reverse engineering experiences, the two authors were asked to start with a small subset of 20 libraries and cross-validate results from each other. All conflicts about the results the were resolved with face-to-face discussions before the two participants proceeded to inspect the whole dataset. During the discussion, we found that for most sampled libraries, the functionality can be made clear through online information once their identities were confirmed. The conflicts were mostly caused by the carelessness of one of the manual analysis participants. Therefore, we did not design a similarity measure for deciding the functionality but totally relied on qualitative methods. More details about the manual analysis process can be found in Appendix B.

Through the manual analysis, we have summarized two interesting empirical findings. The first one is that the renamed libraries are not always constructed by random characters. Instead, some of the modifications are manually done. We found two cases of this phenomenon. The first case is a pair of libraries named `/com/wendyapp/wps` and `/com/lovepop/flystart`. If solely judging from their names, we consider that they are instances of different libraries that likely provide different functionalities. But after inspecting the decompiled code, we found that their code structures are almost identical. Even the positions and frequencies of the used APIs are the same. So we conclude that at least one of the names of the two libraries was obfuscated. In order to pick out the original library name, we check both their names with DNS. It turns out that the original name of this library is `/com/lovepop/flystart`. This is an ads library used for recommending magical pop-up greeting cards for weddings. Based on this knowledge, the other name (`/com/wendyapp/wps`) is highly likely to be an obfuscated one and we do not find any results on this name as an web domain. Indeed, this name does not look like the other obfuscated library names with random sequence of characters. The second case is the pair of `/com/charry/android` and `/com/flurry/android`. Note that this pair is among the top 10 most popular obfuscated libraries (Table XIV). This indicates that although manually renamed libraries are rare, they could have a considerable impact in the app markets.

The second empirical finding is that the renaming-based obfuscations are mostly applied to finance-related libraries, such as ads libraries. In particular, we found that 7 of the 10 libraries in Table XIV can be related to Google's advertising library, suggesting that Google's library might be the prototype of a large number of obfuscated ads libraries in the markets we analyzed and the vulnerabilities in Google's library were inherited by those obfuscated ones.

Typically, an ads library will credit the developer when an ad is successfully presented by the hosting app and clicked by a user. For example, the Google ads library will issue Google money once the ads in the insert app are pressed on the screen. But not all the obfuscated Google ads libraries, especially the manually obfuscated libraries, give money to Google as the original library is designed to. Once the modification of a library has changed the destination of the financial flow from Google to another third-party account, this modified library would steal money that was supposed to be sent to Google. This modified library should be regarded as malicious. We manually analyze these libraries in the table and confirm that Library `/com/nainaidu/ads` is malicious. As for the other modified or obfuscated libraries, we did not find clear clues indicating that they are depositing funds to other account, but they are still suspicious. In our deduction, the adversary should have found some DoS or WebView related vulnerabilities in some specific versions of the Google ads library and modified both the names and corresponding code inside to make sure that these modified versions would not be covered when Google update its ads library.

Table XV shows the ten most commonly unobfuscated libraries. All these libraries are developed by large Internet enterprises. According to our result, they are pervasively spread in apps provided by third-party software markets in China. Once a vulnerability is exploited in any instance of these libraries, it is likely that the impact will be quickly radiated to a massive number of apps. It may be surprising that these libraries are not reported to be obfuscated by LibD, which means there are no other libraries that share the same signature with them. With the above analysis results, we can conclude that these libraries are not tampered with by malicious parties. Therefore, the vulnerabilities discovered in Table XV are surely introduced by the original developers of these libraries.

In Section III-D1, we introduced the RPC vulnerability residing in the library of Baidu moplus SDK [20], which is also set as target of the modified SmartDroid system. After analyzing all the libraries and apps, we found that 2,012 different apps in our dataset are tainted by 15 different versions of the library containing this flaw.

## VII. Limitations of LibD

Compared with previously work, LibD has advanced the state of the art in several aspects. Yet, the technique has limitations and some of them can be potentially addressed.

TABLE XIV: Obfuscated Vulnerable Libraries

| Rank | #Vul-Library (Count) | # Number of affected apps | Vulnerabilities |
|---|---|---|---|
| 1 | /org/gg/music/ ..... /com/google/ydd/ **Total Num:** 20 | 58,976 | DoS, WebView |
| 2 | /com/tencent/a/ /com/tencent/b/ **Total Num:** 2 | 24,012 | DoS, WebView |
| 3 | /com/gg/sda/ ..... /cof/gootle/adz/ **Total Num:** 14 | 17,706 | DoS, WebView |
| 4 | /com/baidu/location/ /com/a/a/ **Total Num:** 2 | 17,679 | DoS, WebView |
| 5 | /com/sdf/ads/ ..... /com/google/ads **Total Num:** 17 | 16,323 | DoS, WebView |
| 6 | /com/nainaidu/ads ..... /com/go2/ads/ **Total Num:** 12 | 13,300 | DoS, WebView |
| 7 | /com/xxgg/abs/ /com/google/ads/ ..... /com/tgxsw/ads/ **Total Num:** 8 | 9,524 | DoS, WebView |
| 8 | /com/google/ads/ ..... /cn/google/ads/ **Total Num:** 10 | 9,447 | DoS, WebView |
| 9 | /com/charry/android/ ..... /com/flurry/android/ **Total Num:** 2 | 4,471 | DoS, WebView, SSL Auth Flaw, FileCross |
| 10 | /com/ccccc/android/ /com/millennia lmedia/android/ **Total Num:** 2 | 4,310 | DoS, WebView |

TABLE XV: Unobfuscated Vulnerable Libraries

| Rank | Library Name | # Number of affected apps | Vulnerabilities |
|---|---|---|---|
| 1 | /com/tencent/mm | 43,731 | DoS, WebView |
| 2 | /com/tencent/mm | 33,491 | Dos, WebView |
| 3 | /com/tencent/mm | 30,100 | DoS, WebView |
| 4 | /m/framework/network | 27,854 | SSL Auth |
| 5 | /com/tencent/connect | 20,433 | DoS, FileCross |
| 6 | /com/baidu/location/ | 17,973 | DoS, WebView |
| 7 | /com/tencent/weibo/ | 1,750 | DoS, FileCross |
| 8 | /com/amazon/inapp/ | 17,001 | DoS, WebView |
| 9 | /com/tencent/qqconnect/ | 15,057 | DoS, WebView |
| 10 | /com/tencent/mm/ | 13,798 | DoS, WebView |

A number of these limitations originate from the fundamental technical challenges in Android app analysis, which we call the general limitations because they are shared by a large number of Android app analysis techniques. The others are more related to the current design and implementation of LibD, thus called specialist limitations.

### A. General Limitations

*1) Packers:* During our evaluation, we noticed that some samples cannot be correctly unpacked or decompiled. This prevents further analysis of LibD since we need the opcodes to compute the signatures. Investigations showed that these apps had been processed by the so-called Android "packers". Generally speaking, packers are tools that can transform APK files into malformed shapes while preserving the functionality of the apps. Packers are similar to code obfuscators in many aspects, but a packer is not limited to work on program code. According to the recent literature [27], there are various ways for a packer to render an APK file unanalyzable, including injecting false metadata into APK manifests, intentionally setting the encryption flag in the APK header without actually encrypting the file, tampering with the magic numbers of the DEX files, and inserting corrupted DEX objects into the APK files.

Many tools have been developed to nullify the mischievous effects brought by packers [27]–[29]. With some engineering effort, these tools can be integrated into LibD as preprocessors to handle packed samples. Indeed, this can increase the processing time of LibD; however, based on our observation, only a small portion of the apps in our dataset are indeed packed. It is possible that the ratio of packed apps is higher in malware samples, but detecting third-party libraries in malware is not yet a prevalent application.

*2) Native Code:* Many Android developers, including library developers, choose to publish part of their products in the form of native ARM machine code instead of DEX bytecode. The native code part of the app interacts with the rest of components through the Java Native Interface (JNI). There are several benefits of developing Android apps in native code. In the early age of Android, performance is possibly the primary motivation, since native code is typically more efficient than managed DEX bytecode. In recent Android versions, however, the performance problem has become much less of a concern due to the significantly improved Android runtime environment and app compilation model. Starting in Android 7.0, a hybrid combination of ahead-of-time, just-in-time, and profile-guided compilation strategies are employed to improve the performance of code written in Java and Kotlin. This makes publishing apps in native code for execution speed not as attractive as before. Nevertheless, it is still a common practice among mobile app vendors since many of them develop apps for different platforms and C/C++ is the currently the only portable choice available to all mainstream mobile OSes. Security can be yet another advantage of native code, since it is considered to be more difficult to reverse engineer than bytecode.

Conceptually, the key steps in our method is also applicable to native code, while the technical details make analyzing native code a problem vastly different from what we have tackled in this paper. It is known that slicing machine code into functions and constructing the control flow graphs is extremely challenging [30]–[32]. Also, there are no or little package structures or inheritance information in native code, making our signature generation less rigorous. Most importantly, it is unclear whether the hash of opcode sequences is a suitable feature at the native code level, since the number of machine instructions can be much larger and the hash may be too sensitive to perturbations caused by compilers. We are currently unaware of a solution that is directly applicable to our problem.

### B. Specialist Limitations

*1) Package-Based Detection:* Similar to most existing library methods, LibD considers packages as the minimal units of a library. In case a package of a library is cloned into the main package of application, LibD will not be able to distinguish the library code from the application's own code inside that package. As such, LibD still need improvements if it is to be employed to analyze apps in adversarial settings, where the package structures of libraries are no longer authentic.

One way to address this problem is to refine the granularity of our current app dissection algorithm. For example, we can identify library candidates as a group of classes instead of packages. Indeed, an adversary can counterattack this improved method by further reorganizing the classes structures [33], but that will significantly increase their cost of operation. On the hand, performing large-scale library detection at the class level will be much more costly than the current package-based scheme. Intensive research and novel methods are required to make the idea practical.

*2) Advanced Obfuscation:* Obfuscation has been widely adopted in Android development. While LibD is able to counter common obfuscation algorithms like package and symbol renaming, there exist more advanced obfuscations that LibD cannot handle. Indeed, the signature we build for Android apps captures some of the semantic features of the code, yet it is mostly syntactic. If the obfuscation alters code syntactic structures intensively, it is likely that the performance of LibD will decline.

ProGuard [23], the official obfuscation toolkit provided by the Android SDK, is considered one of the most popular obfuscators. As mentioned in §IV-G2, ProGuard mostly provided renaming-related obfuscations. Typical objects that are renamed include package, class, and even methods. By applying ProGuard to the compiled DEX code, sensible names of the programming elements in the app are turned into randomly generated 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-G2). However, the latest version of ProGuard (published after the majority of our research was accomplished) started to provide unprecedented obfuscation techniques, e.g., package structure flattening[4]. We speculate that LibD is not capable of handling the new obfuscation at this point.

We also noticed that some third-party obfuscation tools can perform even more advanced obfuscation techniques, e.g., code encryption, control flow flattening, and code virtualization [34]–[36]. Handling these obfuscations is far beyond the capability of syntax-based similarity detection, namely they can very likely impede LibD. One way to accurately analyze deeply obfuscation apps is to employ dedicated deobfuscation techniques. Nevertheless, deobfuscation is still an open problem and has been actively researched [37], [38]. We do not discuss the details of this line of work in this paper.

*3) Functionality-Oriented Analysis:* We have presented two applications for which LibD is adequate. However, some analyses put more emphasis on tracing the functionality of the libraries. In such cases, our method may be too sensitive to divergences between different implementations of the same program functionality, leading to clustering results that do not meet client demands well.

Although this is indeed a limitation of our method, we consider it inevitable since there is unlikely a "silver bullet" effective against all scenarios with respect to the library detection problem. One potential method to improve the current design is to make the sensitivity of the signature configurable through user-provided parameters and let clients set the appropriate settings based on their needs.

[4]https://www.guardsquare.com/en/proguard

## VIII. Other Discussions

### A. Setting Thresholds

As explained earlier (§III-E), LibD identifies libraries according to a predefined threshold, and we set the threshold by validating a broad set of candidates regarding an existing work (§IV-B). Although our experiments report promising results given the threshold as 10, 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-D1). 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.

### B. 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 [39]–[41]. Our experimental result has also demonstrated efficient and precise detection of Android third-party libraries (§IV-D1) using syntactic features.

On the other hand, we have also observed some program semantics-based similarity analysis work [42]–[54]. 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, some of the existing semantics-based similarity work may not scale well [48], [49], [52]. Given the high scalability as a requirement for Android third-party library detection, we consider it may not be feasible to directly adopt previous techniques in our context. We leave it as future

work to integrate more scalable semantics-based methods into our research.

## IX. Related work

### A. Third-Party Library Identification

Early work on third-party mobile library identification mostly focuses on advertising libraries. Book et al. [55] and Grace et al. [56] 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 [57] and PEDAL [5] start to employ machine learning methods to provide more accurate and comprehensive results, but they still target advertising libraries only. AdRob [58] 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 [7] is an Android app clone detector which filters out third-party libraries before computing app similarity. 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 [8] 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. [59], [60] 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. [9] investigated the usage patterns of third-party Android libraries. Another study by Chen et al. [10] 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. Code Clone Detection

The problem of library detection is closely related to the more generalized clone detection problem. Many techniques have been proposed to find pairs of code fragments that share the same provenance. Different from library detection, general code detection typically searches for similarity matches at a pre-defined level of code structures, e.g., functions, files, and whole programs. In contrast, library detection needs to identify the boundary of similarity comparison without much prior knowledge.

Code clone detection techniques work on various syntactic and semantic features of programs. The similarity can be computed based on tokens [40], [61]–[64], parsing trees [41], [65], [66], and dependence graphs [4], [67], [68]. In addition to traditional structure levels, recent methods have started working on more semantics-related abstractions. For example, McMillan et al. [69] proposed to model software functionality with the patterns of standard library API invocations. In their method, Java APIs are grouped by their affiliations with classes and packages and assigned different weights based on their likelihood of occurrence in code repositories. Additionally, APIs correlated to each other in terms of functionality (i.e., compression and IO) are particularly considered as subgroups. These factors are coalesced to form a similarity measure called Latent Semantic Indexing (LSI). Compared to our method, LSI is more semantics-centric, thus more suitable for identifying applications that implement similar functionality but not necessarily with the same provenance. Another recently proposed clone detection method takes one more step further and inspects code semantics similarity at an even finer granularity. Luo et al. [48] used symbolic execution to abstract code semantics with logic formulas. The semantics equivalence of different code segments is then proved by asserting the equivalence of the resulting formulas. This method is very resilient to various software obfuscation algorithms. However, since the method is designed for pairwise similarity comparison, the cost of applying to millions of applications all together is unlikely to be affordable.

It is known that different clone detection methods engineered different features and similarity measures to capture software relevance, while each feature has its own strengths and weaknesses. For improved generality of code detection, researchers have considered combining multiple features and measures together to build more rigorous portraits for code segments. For example Davies et al. [70] developed the Software Bertillonage framework which utilized count-based (e.g., number of API calls), set-based (e.g., the set of classes in a package and the set of methods in a class), and sequence-based (e.g., sequence of method definition in a class) features simultaneously. Although the multi-dimensional features can boost the performance of clone detection in many cases, the method is not very resilient to obfuscations and thus does not work well in adversarial settings. Simple obfuscations like global symbol and package renaming could thwart Bertillonage.

Latest progress in code clone detection [71] employs deep learning techniques, which shifts the part of the burden of feature engineering from detector designers to automated probabilistic learners, i.e., recurrent neural networks.

### C. 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 [4], [7], [16], [72]–[75]. 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. [76] and Stevens et al. [77] extracted advertising libraries from popular Android apps and studied the privacy leakage problems residing in these libraries. Jin et al. [78] discovered that some third-party libraries providing HTML5 support for mobile developers can be easily exploited by code injection attacks. SMV-HUNTER [25] analyzed the man-in-the-middle SSL/TSL vulnerabilities in Android apps and third-party libraries. Li et al. [79] 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.

## X. Conclusion

In this paper, we present a novel technique for 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. We exploit LibD on SmartDroid, a real vulnerability detection system to detect the vulnerable library versions. The evaluation result shows that LibD can find 10,801 vulnerable library instances and accelerate the analysis speed by 5.5 times.

## Acknowledgments

This research was supported in part by the National Natural 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] 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

[2] "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/.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] "smali: smali and baksmali," https://github.com/JesusFreke/smali.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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/.

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

[22] Android, "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.

[23] "ProGuard," https://www.guardsquare.com/proguard.

[24] "ProGuard manual — usage," https://www.guardsquare.com/en/products/proguard/manual/usage#obfuscationoptions.

[25] 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.

[26] D. Wu and R. K. C. Chang, "Analyzing android browser apps for file: // vulnerabilities," *CoRR*, vol. abs/1404.4553, 2014. [Online]. Available: http://arxiv.org/abs/1404.4553

[27] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID '15, 2015, pp. 359–381.

[28] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed Android applications," in *Proceedings of the 25th European Symposium on Research in Computer Security*, ser. ESORICS '15. Springer, 2015, pp. 293–311.

[29] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 358–369.

[30] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Security Symposium*, ser. USENIX Security '14, 2014, pp. 845–860.

[31] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Security Symposium*, ser. USENIX Security '15, 2015, pp. 611–626.

[32] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '17, 2017, pp. 388–398.

[33] C. Foket, B. D. Sutter, and K. D. Bosschere, "Pushing java type obfuscation to the limit," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 6, pp. 553–567, Nov 2014.

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

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

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 756–765.

[44] Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 925–943, 2015.

[45] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 21st International Symposium on Software Testing and Analysis*, ser. ISSTA '12, Jul. 2012.

[46] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '14, Nov. 2014, pp. 66–77.

[47] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, Dec 2016.

[48] 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.

[49] 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.

[50] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *Proceedings of the 30th IFIP SEC 2015 International Information Security and Privacy Conference*, ser. IFIP SEC 2015. Springer, May 2015.

[51] Z. Tian, T. Liu, Q. Zheng, F. Tong, D. Wu, S. Zhu, and K. Chen, "Software plagiarism detection: A survey," *Journal of Cyber Security*, vol. 1, no. 3, pp. 52–76, 2016.

[52] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, 2017.

[53] J. Ming, D. Xu, and D. Wu, "MalwareHunt: Semantics-based malware diffing speedup by normalized basic block memoization," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 167–178, Aug 2017. [Online]. Available: https://doi.org/10.1007/s11416-016-0279-x

[54] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 253–270. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] B. S. Baker, "A theory of parameterized pattern matching: Algorithms and applications," in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, ser. STOC '93, 1993, pp. 71–80.

[62] ——, "On finding duplication and near-duplication in large software systems," in *Proceedings of the 2nd Working Conference on Reverse Engineering*, ser. WCRE '95, 1995.

[63] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul 2002.

[64] A. Aiken, "MOSS: A system for detecting software plagiarism," http://theory.stanford.edu/œaiken/moss/, 2013.

[65] W. Yang, "Identifying syntactic differences between two programs," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 739–755, Jun. 1991.

[66] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the 16th International Conference on Software Maintenance*, ser. ICSM '98, 1998, pp. 368–377.

[67] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06, 2006, pp. 872–881.

[68] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. ACM, 2008, pp. 321–330.

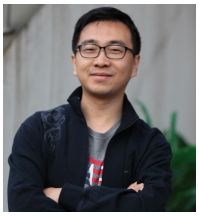
[69] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 364–374.

[70] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Determining the provenance of software development artifacts," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1195–1237, 2013.

[71] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98.
[72] 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.
[73] ——, "Attack of the clones: Detecting cloned applications on Android markets," in *European Symposium on Research in Computer Security*, 2012, pp. 37–54.
[74] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Proceedings of the 6th International Conference on Trust and Trustworthy Computing*, M. Huth, N. Asokan, S. Čapkun, I. Flechais, and L. Coles-Kemp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 169–186.
[75] 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.
[76] 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.
[77] 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.
[78] 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.
[79] 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.

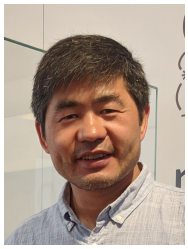
**Menghao Li** received his MSc in Software Engineering from University of Science and Technology of China and his PhD in Information Security from University of Chinese Academy of Sciences. He is currently an assistant processor with the Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences. His main research interests are software analysis, security assessment and vulnerability detection.

**Pei Wang** received his BSc in Computer Science and Technology from Peking University, his MASc in Electrical and Computer Engineering from University of Waterloo, and his Ph.D. in Information Sciences and Technology from The Pennsylvania State University. Pei works on a broad range of research topics including computer security, software engineering, and programming language. He will join Baidu X-Lab as a Senior Security Researcher as of September, 2018.



**Wei Wang** is an engineer in the Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include software security, software analysis, mobile app security, and programming languages. He received his MSc degree in Communications and Information Systems from Beijing Jiaotong University in 2007. Before joining Institute of information engineering, he worked in the Institute Of Computer Science & Technology of Peking University.

**Shuai Wang** is a Postdoctoral Scholar at ETH Zurich. He received his Ph.D. in Computer Security from Penn State University, and B.S. in Electronic and Information Science and Technology from Peking University. Shuai Wang is broadly interested in computer security and specializes in software security, binary code analysis, and low-level security techniques. Shuai Wang will join the Computer Science and Engineering Department at the Hong Kong University of Science and Technology as an Assistant Professor in 2019.



**Dinghao Wu** is the PNC Technologies Career Development Associate Professor in the College of Information Sciences and Technology at the Pennsylvania State University. He is currently Visiting Professor at EPFL. His research is in software systems, including software security, software protection, software analysis and verification, software engineering, and programming languages. His research has been funded by National Science Foundation (NSF), Office of Naval Research (ONR), and Department of Energy (DOE). He received his Ph.D. degree in Computer Science from Princeton University in 2005. He was a research engineer at Microsoft in the Center for Software Excellence and later the Windows Azure Division from 2005–2009. He received the NSF CAREER Award, George J. McMurtry Junior Faculty Excellence in Teaching and Learning Award, and College Junior Faculty Excellence in Research Award.

**Jian Liu** received the B.S. and M.S. degrees in 1997 and 2000 respectively from Yunnan University, and the Ph.D. degree from Institute of Software, Chinese Academy of Sciences in 2005, all in computer science. He is now an associate research professor at Institute of Information Engineering, Chinese Academy of Sciences. He held a visiting professor position in the School of Information Technology and Electrical Engineering, University of Queensland, Australia, in 2010. His current research interests include system and software security, mobile security, web security, program analysis, testing and model checking. He is a member of IEEE.

**Rui Xue** Rui Xue is a research professor with the State Key Laboratory of Information Security, Institute of Information Engineering, CAS. He received his MSc and PhD in Mathematics both from Beijing Normal University. His main research interests is information security, especially in Cryptography. For more information and details, please visit http://people.ucas.edu.cn/~xuerui/.

**Wei Huo** received the PhD degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2010. He is currently an associate professor in Institute of Information Technology, Chinese Academy of Sciences. His research interests include program analysis and software security. He is now working on building a large collaborated software security analysis platform.

**Wei Zou** received his BSc in Software from Nanjing University, PRC, his MSc in Software from the Institute of Computing Technology, Chinese Academy of Sciences(CAS). He was with Peking University (as professor). He is currently a professor with the University of CAS and with the Institute of Information Engineering, CAS. His main research interest is software security.

## APPENDIX

### A. Apps Collected from Minor Third-Party Markets

Table XVI lists the detailed breakdown of the origins of the remaining 62,216 apps in Table I.

| Market | # of apps | URL |
|---|---|---|
| zhuole | 12930 | Sjapk.com |
| appsapk | 2848 | appsapk.com |
| padh | 2461 | padh.net |
| a67 | 3520 | a67.com |
| eoemarket | 4178 | www.eoemarket.com |
| Neteaseapp | 1935 | m.163.com |
| anzhuo.com | 3196 | www.anzhuo.com |
| anuran | 395 | soft.anruan.com |
| xiazaiba | 1384 | www.xiazaiba.com |
| 2265 | 561 | www.2265.com |
| Feifan | 5270 | android.crsky.com |
| Mozhuo | 3764 | apping.cc |
| mm10086 | 3056 | mm.10086.cn |
| d.cn | 11536 | android.d.cn |
| android155 | 2271 | android.155.cn |
| xunzai | 454 | www.xunzai.com |
| mz6 | 1385 | www.mz6.net |
| zhuannet | 107 | zhuannet.com |
| PChome | 965 | download.pchome.net/android |

TABLE XVI: Origins of apps collected from minor third-party markets

### B. Manual Library Analysis Details

When manually analyzing the top 200 vulnerable libraries detected by the accelerated SmartDroid, we employed qualitative methods to answer the research questions about the identities and functionalities of the inspected libraries. By qualitative methods, we meant that the participants made decisions based on unstructured information, along with their own programming experience and understanding. No quantitative measures were designed to guide the analysis.

The method for determining a library's functionality can be summarized as follows.

1) The participants first try to infer library identities and functionalities through package names. Note that some obfuscated libraries only obfuscate their class and method names but not package names, which allows us to still make the inference. For example, it is straightforward to infer that `/com/baidu/location/` is a location library from the Chinese Internet giant Baidu.
2) Similarly, if a library obfuscates its package names but not method names, we may be able to infer its identity and functionality.
3) If the first two steps failed, we harvest "interesting" string literals in the code of a library and use them as keywords for Google search. There are no standard criteria for deciding what strings are "interesting." Typical targets include hard-coded string literals for logging and debugging, as mentioned in the previous paragraph. In most cases we can get a hit in the search engine, e.g., a website introducing the package writer and its products, from which we can learn the information we need for that library.
4) Rarely, we cannot retrieve useful information from the search engine. In such cases, the participants have to look at the decompiled code of the library. Typically, the functionality was inferred by looking at the Android Framework APIs called in the code.

During the cross validation process, there was one single conflict between the two participants out of 20 cases. The reason is that one of the participants drew the conclusion too eagerly by just looking at the package names. The library in this case is `/org/gg/music`. The participant concluded that the library is an unobfuscated music library developed by a party named "gg" without searching the Internet for its detailed information. The second participant, however, further analyzed the dumped string literals after failing to identify this library. The strings suggested that this library is a mutation of `/com/google/ads`. After the discussion, the first participant refined his analysis method to avoid similar mistakes in the subsequent analysis.