Revisiting the Impact of Common Libraries for Android-related Investigations

Li Li[a], Timothée Riom[b], Tegawendé F. Bissyandé[b], Haoyu Wang[c], Jacques Klein[b], Yves Le Traon[b]

[a]*Faculty of Information Technology, Monash University, Australia*
[b]*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*
[c]*School of Computer Science, Beijing University of Posts and Telecommunications, China*

Abstract

The packaging model of Android apps requires the entire code to be shipped into a single APK file in order to be installed and executed on a device. This model introduces noises to Android app analyses, e.g., detection of repackaged applications, malware classification, as not only the core developer code but also the other assistant code will be visited. Such assistant code is often contributed by common libraries that are used pervasively by all apps.

Despite much effort has been put in our community to investigate Android libraries, the momentum of Android research has not yet produced a complete and reliable set of common libraries for supporting thorough analyses of Android apps. In this work, we hence leverage a dataset of about 1.5 million apps from Google Play to identify potential common libraries, including advertisement libraries, and their abstract representations. With several steps of refinements, we finally collect 1,113 libraries supporting common functions and 240 libraries for advertisement. For each library, we also collected its various abstract representations that could be leveraged to find new usages, including obfuscated cases.

Based on these datasets, we further empirically revisit three popular Android app analyses, namely (1) repackaged app detection, (2) machine learning-based malware detection, and (3) static code analysis, aiming at measuring the impact of common libraries on their analysing performance. Our experimental results demonstrate that common library can indeed impact the performance of Android app analysis approaches. Indeed, common libraries can introduce both false positive and false negative results to repackaged app detection approaches. The existence of common libraries in Android apps may also impact the performance of machine learning-based classifications as well as that of static code analysers. All in all, the aforementioned results suggest that it is essential to harvest a reliable list of common libraries and also important to pay special attention to them when conducting Android-related investigations.

1. Introduction

Rapidly, Android has grown as a popular programming platform for developers and a worthwhile operating system for manufacturers. Of the 432 million smartphones sold in the last quarter of 2016, over 81% of them are actually equipped with Android, which is far beyond the occupation of other mobile operating systems such as iOS [1]. In addition to smartphones, Android now is also used for household and office devices such as tablets, personal computers, TV sets, fridges, washing machines, etc., run a diversity of applications.

Unfortunately, because these apps pervade all human activities, malicious or malfunctioning apps have become important threats that can lead to damages ranging from benign (e.g., app crashes) to critical (e.g., financial losses with malware sending premium-rate SMS, reputation issues with private data leaks, and potentially loss of human lives when apps will run on Android cars). These threats are further exacerbated in an ecosystem where thousands of apps written by hundreds of third-party developers are made readily available for download by users. GooglePlay, the official market for free and paid Android apps, now has over 3 million apps in various categories from productivity and messaging to games and social networking. Antivirus vendors, which regularly report on the status of malware spreading, have revealed that Android is now a target of choice for malicious attacks [2].

The research community has produced a large body of work for mitigating the emerged threats in the Android ecosystem, essentially to guard the security and privacy of users. For scalability and practicability reasons, a substantial number of the proposed approaches [3, 4, 5] rely on static analysis to parse the entire code shipped in the app package to find security problems in code instructions, to extract features for further processing or simply to compare apps in large repositories. Unfortunately, because Android development paradigm allows to easily include third-party code, in the form of libraries, a significant portion of an app is eventually irrelevant to certain analyses (e.g., piggybacking analysis). Common libraries embedded in app code thus constitute a significant barrier for the static exploration of applications code. There are indeed a number of research directions where tasks are hindered by the presence of common libraries in app code:

Repackaging detection. Techniques for comparing apps to detect repackaged apps by computing their similarities may provide inaccurate results when common libraries are pervasively used. In a preliminary study, Wang et al. [6] have found that over 60% of Android apps' code is contributed by common libraries. To increase accuracy in detection, most recent ap-

proaches have been considering filtering out such libraries, using heuristics.

Malware detection. Recently, researchers have been focusing on machine learning techniques as scalable means to identify malicious apps in large datasets. To that end, they usually extract static features from the code. Unfortunately, the presence of library code may create significant noise making it hard to discriminate benign features from malware-specific ones. To account for such noise, some approaches, such as MUDFLOW [7], assume that advertisement libraries, which are common libraries, are trustable. Thus, they simply ignore all results related to ad libraries, so as to focus on the real app code. In the case of MUDFLOW, 12 frequently used advertisement frameworks are considered. Although these 12 libraries are not representative, it does show the necessity to exclude common libraries for malware detection.

Code analysis. Besides the false positives that may arise due to over-approximation, static code analysis is also often challenged by computing power and memory requirements. In the case of FlowDroid [8], the state-of-the-art static taint analysis tool for Android apps, it was reported that the analysis time can be too high [7]. Let us refer back again to Wang et al.'s findings, where 60% of app's code are contributed by common libraries, which would thus indicate roughly that over half of the CPU and memory consumption could be wasted on irrelevant library code, threatening the performance of the analyzer.

The aforementioned cases constitute strong motivations for automatically identifying once a large set of **common libraries** from market-scale apps, which could then be used by other approaches to immediately take such libraries into account. A straightforward solution for achieving such a task is to build a comprehensive *whitelist* of common libraries. Wang et al. [6] claim to have collected more than 600 different common libraries to improve their repackaged app detection process. Other approaches [9, 10, 11, 12] build on top of limited whitelists collected using simplistic heuristics and containing between only 9 (AdDroid [10]) and 103 (Bootk et al. [11]) libraries.

In this paper, we investigate the use of common libraries in Android based on a dataset of around 1.5 million apps collected from the official Google Play market. In particular, we build and maintain a comprehensive whitelist of 1,113 Android common libraries that we share with the communities. Our approach identifies common libraries based on the assumption that they are used by many apps as such, i.e., without developer modification. We further label those libraries to distinguish between **advertisement libraries** (or ad libraries, a specific type of common libraries) and others, using heuristics defined from our manual investigations.

The initial goal of this work is to provide a comprehensive and publicly accessible *whitelist* of Android libraries and their unique representations. Beyond that, based on the collected dataset of libraries, we revisit several research directions related to the impact of common libraries w.r.t. various Android-related investigations. Notably, we explore the impact of common libraries for repackaged Android apps detection, machine learning-based malware detection, and static Android app analyses, etc. In particular, we aim at answering the following three research questions:

- **RQ1:** What is the impact of common libraries on the performance of repackaged apps detection?
- **RQ2:** What is the impact of common libraries on the performance of machine learning-based malware detection?
- **RQ3:** What is the impact of common libraries on the performance of static analysis of Android apps?

Finally, towards better understanding the usage of libraries in real-world Android apps, we further design and implement a library representation tool called LibRepresenter. The main function of this tool is to compute abstract representations of common libraries. Given a Java package name (or library), LibRepresenter will form an abstract string based on its structural information to represent it. The rationale behind this abstraction (or representation) is that it provides a means to directly search and compare libraries among a large set of Android apps while being resilient to simple obfuscations (e.g., renaming). To demonstrate the usefulness of this library representation, we further empirically explore the following two research questions.

- **RQ4:** Are libraries used differently by benign and malicious apps? If different, to what extent can the usage of libraries be used for learning anti-virus predictions?
- **RQ5:** Can abstract representations of common libraries be useful to identify their obfuscated counterparts?

Overall, in this work, we make the following contributions:

- An approach to automatically harvest common libraries from market-scale Android apps. In this work, we collect 1,113 common libraries from a dataset of around 1.5 million Android apps.
- A discriminative study of advertisement libraries, for which 240 common libraries are recognized as ad libraries.
- An empirical investigation and evaluation of the use of common libraries in Android apps. We show that there are indeed significant differences in the use of common libraries between benign and malicious apps. Besides, we also show that our harvested common libraries are indeed useful for other approaches, e.g., to reduce both false positive and false negative rates for piggybacked apps detection (i.e., a special type of repackaged apps where the repackaged version has been injected with some payloads), and to improve the time performance of static code analysers.
- A benchmark of library variants (i.e., the different versions, modified either by the app developers or by attackers, of the same library) and obfuscated library versions, that we make available online to the Android research community aiming at facilitating further researches towards performing better and securer libraries to app developers. This benchmark of library variants and obfuscated versions, along with the identified two *whitelist*s

of libraries as well as the various prototype tools implemented in this work are made available online to the Android research community at:
https://github.com/serval-snt-uni-lu/CommonLibraries.git.

This paper is an extended version of a conference paper published at the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering [13]. Compared to the conference version, this work has refined our previous approach by going one step further to represent common libraries with unique abstractions that could be later leveraged to identify library variants (different versions) as well as obfuscated library versions. To do so, we have designed and implemented a prototype tool named LibRepresenter, which represents, based on the structural information of Android app code, every package of an Android app into a unique abstraction that provides a means for analysts to identify library variants and pinpoint obfuscated library versions. To further demonstrate the usage of our collected common libraries, we present another static analysis tool called LibExcluder, which takes as input an Android app and outputs a new app version where the code from a given set of libraries are removed. Based on these two tools, we then added two new research questions. The first added research question explores the usability of LibExcluder in improving time performance of static code analysers. The experimental results demonstrate that excluding common libraries can indeed increase the performance (i.e., reduce the CPU and memory consumption) of state-of-the-art tools, where not only time consumption is reduced but also successful analysing rate is increased. The second research question explores the usefulness of our collected library representations for which we seek to identify obfuscated library versions based on the abstracted representation thanks to LibRepresenter. The empirical results show that our approach is promising to be leveraged to identify obfuscated library counterparts. Furthermore, we have improved the abstract, introduction, threats to validity, related work, conclusion sections with up-to-date discussions.

The rest of this paper is organized as follows: After the general introduction section, we discuss the closely related studies in Section 2. Then, we introduce two concrete examples to better explain the problems we attempt to address in Section 3. Next, we present our approach and its implementation details, including that of LibExcluder, in Section 4 and Section 5, respectively. In Section 6, we present our investigated data set and the overall results. We then empirically evaluate our findings in Section 7, followed by a discussion of limitations in Section 8. Finally, in Section 9, we conclude this paper.

2. Related Work

In this section, we discuss a batch of works that investigate the issues related to libraries and show that even if libraries are not harmful by themselves, they threaten the validity of other approaches. After that, we also summarise the works that are dedicated to the identification of Android libraries.

2.1. Problems of Libraries

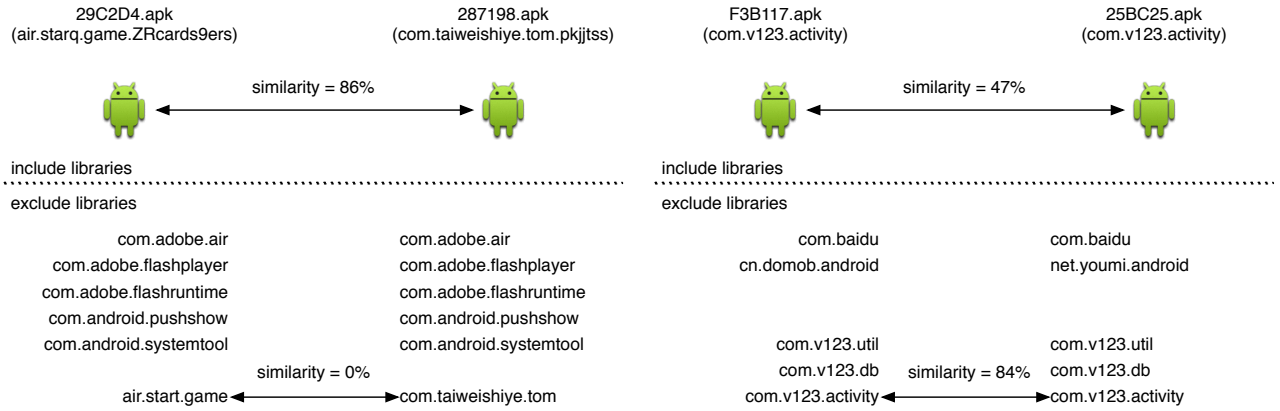
As reported by Hu et al. [14], Android libraries are currently suffering three threats: 1) the library modification threat, where normal libraries can be modified to be malicious. Our previous work has also confirmed this findings [15]. 2) the masquerading threat, e.g., a well-known malware family called *DroidKungFu* uses names such as *com.google.update* to pretend the services are provided by Google [16]. 3) the aggressive library threat, where some legitimate libraries have aggressive behaviours such as collecting users' email address.

Other works [17, 3, 11, 18] done by us and by others, have also shown that some libraries frequently and aggressively collect (leak) users' private information. For instance, the most common leaked information is the device id, which is used by ad libraries to uniquely identify a user. These findings are in line with the investigation of Stevens et al. [19], in which the authors show that, through libraries, users can be tracked by a network sniffer across ad providers and by an ad provider across apps.

Recently, Derr et al. [20] have investigated the updatability of third-party libraries on Android and have empirically found that many Android apps access into outdated libraries and their developers do not attempt to update them in order to avoid ostensible re-integration efforts and version incompatibilities. They have also demonstrated that over 95% of vulnerable libraries can be easily fixed through a drop-in replacement of the vulnerable library with the fixed version. This evidence suggests that problematic libraries are likely to stay in the Android ecosystem that would continuously infect Android app users and most of those infections could be easily avoided. The library variants and obfuscated versions identified in this work is actually our first step towards providing a reliable benchmark for researchers and practitioners to investigate the updates of libraries and hence to invent intelligent techniques for automated library updates.

For advertisement library, Stevens et al. [19] argue that ad libraries usually require permissions beyond their real needs and some badly programmed libraries use Android's Javascript extension mechanism insecurely. AdRisk [9] focuses on detecting privacy and security risks posed by ad libraries. Most notably, it shows that some libraries even execute untrusted code from internet sources. Moreover, those untrusted snippets are fetched through an unsafe mechanism, which by itself has caused serious security risks. Gui et al. [21] have shown that free ad libraries actually come with a hidden cost for developers such as the rating of apps. As reported by Mojica et al. [22], ad libraries are indeed impacting the ratings of Android apps.

Although our work in this paper is not dedicated to identifying problems of libraries, our findings, the list of common (ad) libraries, can definitely benefit other approaches (e.g., API studies [23, 24, 25, 26, 27, 28]) by giving them a good starting point for thorough analysis. For instance, Mojica et al. [23] leverages Software Bertillonage [29] to investigate reuse among Android apps. They have found that most of the classes reused are actually from third-party libraries. With the help of our large comprehensive common library set, the precision of their results could be much improved.



(a) False Positive. (b) False Negative.

Figure 1: Two motivating examples show the importance of excluding common libraries in order to perform precise repackaged apps detection. Note that *F3B117.apk* and *25BC25.apk* are actually signed by different certificates although they share a same package name.

2.2. Research Works threatened by Libraries

Researchers have noticed Android libraries will definitely influence the results of app clone detection [6, 30, 31, 32], where most of them use a list of libraries as a whitelist. As an example, Chen et al. [30] leverage a whitelist containing 73 libraries in their approach, which is far away from being a complete whitelist of existing libraries, as shown by Wang et al. [6], over 600 distinct libraries have been identified. However, this list is not publicly available. Besides, compared to our findings in this paper, this list is also considerably incomplete.

For clone detection, it is important to detect and filter out third-party libraries, as the results may be doomed if the studied apps are dominated by common libraries. As an example, the very first approach presented by Linares et al. [33] shows that the results are statistically and significantly different between including and excluding third-party libraries. Not only for clone detection, but also for machine learning-based malware detection, the results are threatened by common libraries. MUDFLOW [7], as an example, uses a list of 12 well-known ad libraries as a whitelist, to exclude such features that fill in them. A later work, done by Li et al. [34], has also leverage that list in their machine learning-based malware detection.

Our work, in this paper, provides a comprehensive list of common libraries, that can be leveraged by other approaches and thus to significantly refine their results.

2.3. Identification of Libraries

Wang et al. [6] use an automated clustering technique to detect common libraries, in which they have found over 600 distinct libraries. Similarly, Ma et al. [35] and Li et al. [36] also leverage clustering based approaches to detect common libraries in Android apps. Our approach is in line with their assumptions on common libraries, however, we come up with a different approach and we also discriminate ad libraries from common libraries, for which they have not. Soh et al. [37] present a tool called LibSift aiming at detecting third-party libraries based on package dependencies which are resilient to common obfuscations. Another approach called AdDetect [38], identifies Android ad libraries through their semantics (e.g., the usage of Android components, or specific APIs) and then performs an ML-based classification to detect ad libraries. However, these approaches do not report any findings that can benefit the Android research community (e.g., library code may impact the detection of repackaged Android apps).

3. Motivating Example

We now motivate our work by discussing the impact of filtering out libraries from apps when performing repackaging detection. Repackaging is an operation that consists in taking an existing app, unpacking it, then modifying it by adding a (generally malicious) new payload and re-signing it, before distributing it as a new app. Like repackaged apps (where a payload is not necessarily added), repackaged apps are now pervasive in the Android ecosystem where they further constitute an easy way to build and distribute malware [39, 40, 41]. A typical approach for detecting repackaged apps consists in performing pairwise comparisons to identify the original app that was actually repackaged. However, in the process of computing similarity, libraries, which may account for a large portion of apps, can influence towards inaccurate results. We present two real-world examples of pairs of apps where the presence of libraries can lead to a mislabelling of a legitimate app as repackaged or a failure to flag a repackaged app as such.

3.1. Mislabeling Legitimate apps as Repackaged

We consider in Fig. 1a the case of two apps[1] collected from an Android market. The packages in their code structure are very similar when considering the common libraries that they integrate: one app has 86% of its code[2] that is also contained in

[1]Unique package names: *air.starq.game.ZRcards9ers* and *com.taiweishiye.tom.pkjjtss*.

[2]The percentage is computed based on method level, where more details will be given in Section 4.2.

the other app. However, considering the results of a prior investigation of a set of 1,169 known legitimate/repackaged app pairs where we found that most of the similarity degree ranges between 81% and 100%, we could set a threshold of 80% for identifying repackaging cases. This threshold has also been used by other studies [42, 43] and has been experimentally demonstrated to be reliable to highlight repackaged Android apps. This, unfortunately, would lead to a mislabeling in the above case. Indeed, a detailed analysis of both apps shows that they are actually using several common libraries (e.g., `com.android` and `com.adobe`). Excluding such libraries from the similarity computation, the similarity degree falls down to 0%, leaving no room for a false positive prediction.

3.2. Missing True Repackaged Apps

We now consider in Fig. 1b two apps which are known to be a legitimate/repackaged app pair. These apps share the main package called *com.v123.activity*. However, library *cn.domob.android* was replaced in the repackaged app with the library *net.youmi.android* to redirect the revenues of the legitimate app to another developer. Nevertheless, although these two apps are repackaged from one to another, their similarity degree is only at 47%, which would constitute a false negative in our detection scheme with a threshold of 80%. However, if the detection system identified first the common libraries and dismissed them during the pairwise comparison, the similarity degree would reach 84%, leading to a successful prediction.

Overall, the validity of pairwise comparison for repackaging detection could be threatened when substantial parts of app code are common library code. Thus, to limit both false positives and false negatives, library filtering is now more and more considered in state-of-the-art repackaging and repackaging detection approaches [6, 44, 30]. However, the whitelists that they leveraged is built based on manual investigations or automatically with limited datasets. Furthermore, these whitelists are seldom available to other researchers in the community.

4. Methodology

In this section, we provide details on the approach that we have devised to collect common libraries and their unique representations (abstractions).

Fig. 2 illustrates the general process of our approach, which is dedicated to harvest common libraries in Android apps, identify advertisement libraries among them, and represent the library with comparable abstractions. First, for our approach to make sense, we need a large and representative dataset of Android apps. Then, as a first step, we visit all the apps in the dataset and rank all packages in terms of the frequency of their appearance in apps. For the sake of simplicity, we assume that a package with the same name in several apps is a candidate library. Thus, *Step 1* outputs a ranked list of candidate libraries, where the highest ranked candidate library has the most recurring package name in the dataset. In the second step, we perform a more fine-grained pairwise comparison of candidate library code within apps. The objective of *Step 2* is to confirm as common library packages those recurring packages that have the same name and are very similar in their code. Next, in *Step 3*, we further investigate the harvested libraries to label those that are advertisement libraries and thus may be treated differently in some Android analysis approaches. Finally, in *Step 4*, we design and implement a prototype tool called LibRepresenter that leverages code structural information to represent app packages with an abstract string that could then be leveraged to support quick comparison of different libraries.

We now provide details on how each step works in the following four subsections.

4.1. Step 1: Candidate Libraries Extraction

We assume that common libraries are such software packages that are:

- used in a large number of apps – recurring packages have a very high probability of being common libraries.
- used by developers without modifications – their code must be similar across apps. Hu et al. [14] have found that over 80% of libraries are indeed used without modification in their dataset of 100,000 Google Play apps.

Building on those assumptions, and leveraging a large dataset, we extract all package names from Android apps and cluster them based on their frequency of occurrence in the dataset. Theoretically, packages that appear in at least two apps could be taken as candidate libraries[3]. To reduce the number of distinct packages considered as candidate libraries, and which must be further processed we consider two constraints:

- We only consider the first three segments[4] of package name or the entire name if there are less than 3 segments. With this constraint, we manage to limit the number of redundant sub-packages while still guaranteeing a large diversity in package names.
- We also exclude packages with names starting with *android.support*. Indeed, there are many sub-packages within this package and they are used pervasively in Android apps. Furthermore, since these are part of the Android framework, we do not consider them in our study[5].

4.2. Step 2: Common Libraries Confirmation

Because package naming is done in Java programming with limited constraints, any two packages may share the same name while being completely different in terms of code functionality. Also, the frequency of a package name may actually be contributed by repackaging operations, obfuscation activities (e.g.,

[3] Actually this may not be true if the apps are from the same developer. However, since we are performing experiments on a large set of apps, this small deviation will not impact our final results.

[4] In this paper, we use the term segment to describe each domain of different levels, e.g., for package *org.example*, we say it contains two segments, which are *org* and *example*.

[5] Nonetheless, we do consider *android.support.** as a common library in this work.

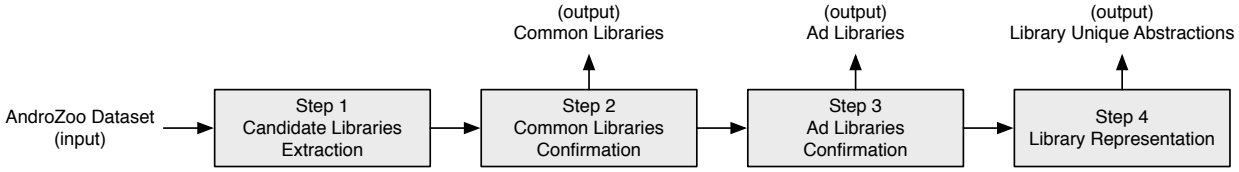


Figure 2: Approach overview.

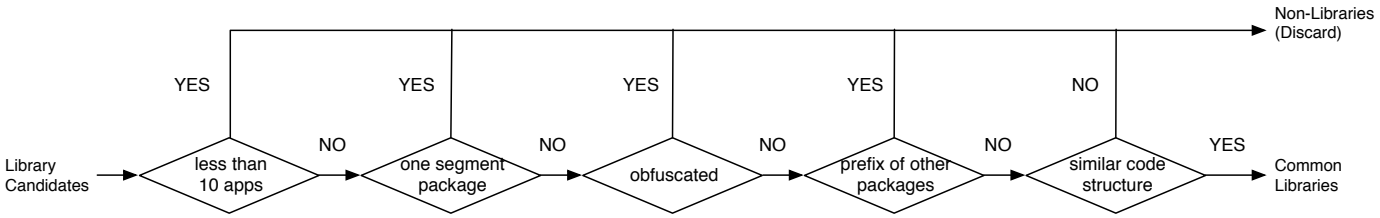


Figure 3: Refinement process for common libraries identification

a.a.a or *com.a* are recurrent in many obfuscated apps) or simplistic naming (e.g., *debug* or *mobile* package names). Thus, we must refine the list collected in the previous step with code similarity measurements to find actual code packages used as common libraries.

Beforehand, given the expensive property of pairwise comparison, we use heuristics to exclude from the candidate libraries outputted by *Step 1*, those packages which would be irrelevant. Our refinement process is shown in Fig. 3.

1) At first, we focus on those packages whose names appear in more than 10 apps to reduce the number of candidate libraries to the most relevant ones.

2) Then, we remove such packages whose names contain only one segment. Although such short names are indeed likely to be redundant in several apps, they are not likely to be those of packages that will be distributed as common libraries. Indeed, to prevent package name collisions, one convention in Java package naming[6] recommends organisations/development teams to use their reversed Internet domain names (e.g., com.facebook) to begin their package names, which justifies our assumption that common libraries, intended for wide distribution, have package names with several segments.

3) Next, we undertake to exclude packages with obfuscated code. However, because there is currently no advanced approach for checking whether a package is obfuscated or not, we build on a naive approach based on observations that we inspect from several obfuscated apps: every package that contains a single letter segment (e.g., *d* of *com.idreamsky.d*) is considered as obfuscated.

4) To further reduce the number of candidates, we exclude such packages that are prefixes of other packages (e.g., we remove package *com.sansec* if package *com.sansec.AESlib* exists). The idea behind this decision is that on the one hand long packages would indicate more fine-grained examination while, on the other hand, short packages would increase the chance of being duplicated (by accident).

5)In this step, we perform package similarity analysis to discriminate common library packages from normal app code package. Given p, a package name, and A, a set of apps which include a package named p, our similarity analysis works in three steps:

- *1) Pairwise combinations of apps.* We consider all the pairwise combinations of apps with package name p. Recall that every considered package name p was selected as a candidate because it appears in at least 10 apps. Thus, for any given p, there are at least $\binom{10}{2} = 45$ pairs to compare. Google's ad package *com.google.ads* is the one for which A is the largest (247,394 apps), leading to over 30 billions pairs that require comparisons. For scalability reasons, we randomly selected for each case of a package name p, 10 pairs of apps, allowing us to assess whether this package name indeed represents a common package code across the apps.
- *2) Method Comparisons.* Analysis of a pair of apps is performed by computing the similarity between their methods. This similarity takes into account not only the signatures of apps but also their respective contents. Two methods, from two different apps, with the same signature are said to be *identical* only when their contents are the same. Otherwise, they are simply said to be *similar*. Such methods may exist between two packages of the same library in several cases: a method in one library package may be modified to insert malicious payload during repackaging operations; different obfuscation algorithms applied on different apps that include the same library may produce methods with the same signature but different contents. To limit the impact of obfuscation, we proceed to abstract the contents of methods by comparing the types of statements (e.g., "invoke") in the Jimple code, leaving out all names of variables/fields/methods. However, since obfuscation is not expected to modify SDK API methods, we also take into account

[6] `https://newcircle.com/bookshelf/java_fundamentals_tutorial/packaging`

the names of such methods. Eventually, the similarity of methods is computed as a simple text differencing.

- *3) Similarity Analysis.* In the last step, we finally perform pairwise similarity analysis for packages with the same name p. There are two thresholds, namely t_p and t_a, which are involved in the similarity analysis. First, we consider that two packages p_1 and p_2 correspond to the same common library p if p_1 and p_2 are identical or are at least similar up to a threshold t_p. Second, because of the known common phenomenon of repackaging in Android, which may nullify the package similarity (because they are probably from the same original app), we must dismiss cases where the similarity score of the pair of apps (app_1 and app_2) is higher than a threshold t_a. Note that the similarity between apps is computed at the method level (i.e., what percentage of methods are identical or similar between the apps?).

Algorithm 1 Package similarity analysis.

```
1:  procedure SIMILARPACKAGES(p, app1, app2, t_p, t_a)
2:      Input: p: package
3:      Input: t_p,t_a: thresholds for package-related, all methods
4:      Output: TRUE: p is similar between app1 and app2
5:      s ← new InnerStorage()
6:      MS_1 ← getMethodSignatures(app1)
7:      MS_2 ← getMethodSignatures(app2)
8:      for all ms_i ∈ MS_1 do
9:          if ms_i ∈ MS_2 then
10:             if content(ms_i,MS_1).equals(content(ms_i,MS_2)) then
11:                 store(s,p,ms_i,"identical")
12:             else
13:                 store(s,p,ms_i,"similar")
14:             end if
15:         else
16:             store(s,p,ms_i,"deleted")
17:         end if
18:     end for
19:     MS_2.removeAll(MS_1)
20:     for all ms_i ∈ MS_2 do
21:         store(s,p,ms_i,"new")
22:     end for
23:     total_pkg ← total(s.pkg)
24:     total_all ← total(s.all)
25:     simiscore_pkg ← max(s.pkg.identical/(total_pkg−s.pkg.new), s.pkg.identical/(total_pkg−s.pkg.deleted))
26:     simiscore_all ← max(s.all.identical/(total_all−s.pkg.new), s.all.identical/(total_all−s.all.deleted))
27:     if simiscore_pkg >= t_p && simiscore_all <= t_a then
28:         return TRUE
29:     else
30:         return FALSE
31:     end if
32: end procedure
```

To summarize, as illustrated in Algorithm 1, for similarity analysis, given a pair of apps (app_1, app_2), we compute four metrics (cf. lines 8-22): *identical* (i.e., the number of methods that are exactly the same, both in terms of signatures and implementation), *similar* (i.e., the number of methods having the same signature but with different contents), *deleted* (i.e., the number of methods that exist in app_1 but not in app_2), and *new* (i.e., the number of methods existing only in app_2). These metrics are good indicators for comparison and have been leveraged in state-of-the-art Android similarity tools, such as Androguard [45] to compute similarities among Android apps. Basically, the problem of identifying similar libraries can be transferred to counting the number of identical methods shared by given two Android apps. The more methods shared by two apps, the more similar these two apps should be. Given these metrics, we can compute the similarity between the pair (app_1, app_2) using Formula 1 (cf. lines 25-26).

$$similarity = max\{\frac{identical}{total-new}, \frac{identical}{total-deleted}\} \quad (1)$$

where

$$total = identical + similar + deleted + new \quad (2)$$

Note that we use the same formula to perform the similarity analysis of a given pair of packages (p_1, p_2), except that the metrics are computed by counting methods in packages rather than in apps (e.g. *identical* is the number of methods that are exactly the same in p_1 and p_2, *deleted* is the number of methods that exist in p_1 but not in p_2, etc.). This is actually the main reason (i.e., flexibility) that we decide to implement the similarity computation algorithm by ourselves, despite that there are already several existing ones proposed to the community (e.g., Androguard [45], WuKong [6], and the approach introduced by Mojica et al. [23]). Nonetheless, this choice should not impact our approach toward identifying common libraries in Android apps.

4.3. Step 3: Identification of Ad Libraries

A specific example of the type of widespread common libraries in Android is advertisement libraries. Such libraries are indeed used pervasively as they constitute one of the main ways for app developers to be rewarded for their development effort. Ad libraries are also often inserted during repackaging to redirect revenues. Their presence in an app also often lead antivirus products to flag them as adware. Recent approaches for Android security analysis are now processing ad library code in a specific way to reduce false positives. For example, MUDFLOW [7] simply does not report any potentially sensitive data leaks through ad libraries, as they might be legitimate. To that end, they have leveraged a limited whitelist of 12 libraries. In this context, we propose to further mine our collected set of common libraries to identify a large set of ad libraries which could be leveraged to improve the results of Android analyses. To that end, we consider a basic method of detection based on the library name and a more semantic method based on the characteristics of ad libraries, where these two methods are supposed to be complementary to one another.

4.3.1. Keywords matching

We note that ad library package names generally contain keywords that include the term *"ad"*. Widespread examples of such packages are *com.google.ads* and *com.adsdk.sdk*. Unfortunately, simply matching "ad" in the package name would lead to a substantial portion of false positives as several library package names have "ad" in their segments which are common words (e.g., shadow, gadget, load, adapter, adobe). Thus, to work around this limitation, we collect all English words containing "ad" from SCOWL[7] (accounting for a total of 13,385 words, including the aforementioned common words such as shadow, addition, radio, adobe, etc.), and dismiss packages containing such words as potential ad libraries. Note that many ad libraries, including famous ones such as *cn.domob* and *com.mopub* do not contain "ad" keyword. Therefore, we additionally resort to a semantic method, detailed in the next subsection, based on the characteristics of ad libraries to supplement this keyword matching approach.

4.3.2. Ad features investigations

We consider samples from a list of ad packages summarized by Grace et al. [9] and manually investigate how ad libraries differentiate from other common libraries and infer a set of features whose presence in a package would justify the tag of ad library.

- *1) Internet usage:* All investigated libraries unsurprisingly require access to the Internet to remotely upload to a server some viewing statistics and update ad contents. Thus, apps integrating add libraries also require permission `android.permission.INTERNET`. Given this fact, we can already exclude a number of common libraries, which appear in apps without Internet access. However, given that an app may request the *INTERNET* permission for its own needs, we cannot immediately state that a common library in such an app is an ad library. Instead, we must investigate whether the code of such an app indeed declares uses Internet-related APIs. To that end, we leveraged the whitelist of such APIs, originally shared by PSCout [46], to produce candidate ad libraries among the common libraries.
- *2) Components declaration:* Our manual investigations have also revealed that ad libraries often contain components, mainly Activities, for facilitating users' ad-related interactions (e.g., switching to a new full-screen ad page when users click on an advertisement banner). As a concrete example, MoPub[8] is an advertisement library targeting both Android and iOS. To integrate this library in their apps, developers must declare four components in their apps' *manifest* file. One component, in particular, *MraidVideoPlayerActivity* is necessary for video ads to work properly. Thus, when a library package is associated with a declared component, we flag it as a potential ad library.
- *3) Views declaration:* In Android, advertisements are generally set to be visualized, which form in Android programming imply the use of view gadgets (i.e., classes extended from `android.view.View`). Thus, we check whether there are *View*-based classes under a common library to flag it as a candidate ad library.

4.4. Step 4: Library Representation

Our identified common libraries are based on Java package names that can be easily manipulated by name-based obfuscation techniques (e.g., to protect legitimate code by app developers or to protect tampered code by attackers) or simply by modification of human beings (e.g., by attackers to inject vulnerable code into the original code base). As demonstrated by Li et al. [47, 42], attackers are well motivated to tamper common libraries so as to perform repackaging attacks on Android apps. Indeed, by just tampering a single popular library, all the apps that access into that library could be contaminated.

Both obfuscation and modification operations produce different variants of the original library. Because of security reasons (e.g., to detect vulnerable apps), there is a strong need to identify different library variants. Indeed, by investigating the changes among the different variants of the same library, app analysts can observe promising insights that can eventually be leveraged to locate vulnerabilities in Android apps or understand the reasons and intentions behind the obfuscation/modification operations.

Towards identify library variants as well as obfuscated library versions, in this work, we design and implement another research-based tool called LibRepresenter. Given a Java package name (or library), LibRepresenter represents it via an abstract string that is formed based on its structural information. The rationale behinds this abstraction (or representation) is that it provides a means to directly search and compare libraries among a large set of Android apps. Because of obfuscation, some structural information is modified and hence cannot be leveraged to directly form the unique representation. To this end, LibRepresenter provides two types of abstractions: one with limited information that is supposed to be used for pinpointing obfuscated library versions while another one with more comprehensive information that can then be leveraged to identify library variants.

The representation of common libraries conducted by LibRepresenter is mainly in two steps:

- *1) Structural Tree Construction.* Given an Android app, LibRepresenter builds a tree to model its structural information, hereinafter we call this tree as *structural tree*. In particular, for every method presented in the app, we model all the segments of its Java packages as well as its class and method name as nodes and the logic sequence of those segments, class/method names as edges. As an example, Fig. 4 illustrates an example of simplified structural tree that is constructed based on the *ActivityMessenger* class[9], which is available in app *ServiceCommunication1* of the DroidBench repository. Method names such

[7] Spell Checker Oriented Word Lists: http://wordlist.aspell.net

[8] https://github.com/mopub/mopub-android-sdk

[9] https://github.com/secure-software-engineering/DroidBench/blob/master/

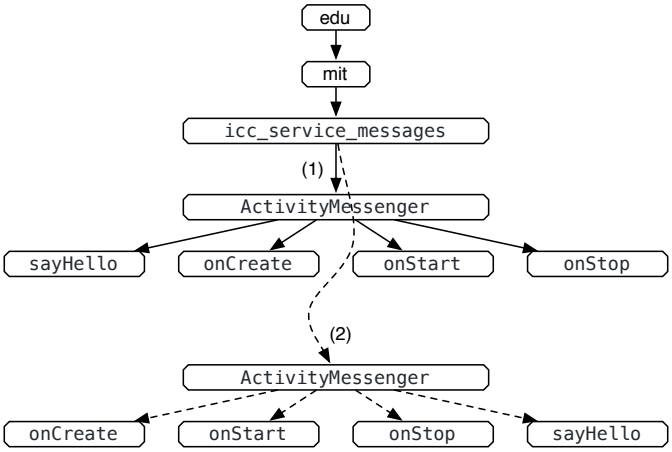


Figure 4: An example of a simplified structural graph (without considering the dashed edges) and the process of rearranging nodes (replacing edge (1) with edge (2)).

as *onStart* and *sayHello* are always presented in a structural tree as leaf nodes and are always children nodes of such nodes that are formed based on class names (e.g., *ActivityMessenger*).

Since we want to represent library code with unique abstraction strings, we need to rearrange the constructed structural tree to avoid potential inconsistencies, e.g., the sequence of integrated nodes. To this end, for each non-leaf node, we rearrange its all children nodes in a way that all these nodes are presented, from left to right, following alphabetical order. As shown in Fig. 4, after the rearrangement, the *onCreate* node has been moved to the leftmost place while the *sayHello* node is moved to the rightmost place.

- *2) Library Abstraction.* After the construction and rearranging of the structural tree, LibRepresenter represents each node (or library) based on all its children nodes. All specifically, these children nodes are visited and recorded in a breadth-first manner and the final abstraction string is made up of two parts: (1) concrete part, where node names are leveraged; and (2) abstract part, where declared modifiers for leaf nodes (i.e., every leaf node represents a Java method) while the number of children nodes for non-leaf nodes is leveraged. Listing 1 presents the representation of package *edu.mit.icc_service_messages*, yielded by LibRepresenter.

  The combination of these two parts can be leveraged to identify library variants that have small changes out of the whole code structure while the abstract part per se can be leveraged to identify possible obfuscated versions. So far, advanced obfuscated techniques that could change the code structure of libraries are not addressed by LibRepresenter. Nevertheless, as demonstrated by Wang and Rountev [48], most Android apps are actually obfuscated by the default obfuscator ProGuard that usually performs name changes only. As a result, although only simple structural information is leveraged, we would expect that LibRepresenter could still be useful for many Android apps towards identifying obfuscated library versions. It is worth to mention that, so far, LibRepresenter only records basic attributes to represent the code structure of Android apps. It is however quite easy to extend the current implementation to further include more attributes (e.g., the number of statements for each method, a.k.a. leaf node) and thereby to improve the accuracy of detecting library variants and obfuscated versions.

4.5. Implementation details

We implement our approach through several languages such as Java and shell/Python scripts. In *step 1*, we leverage *Apktool*[10] to disassemble Android apps. Given an Android app, we extract the prefixes of paths of *smali* files (a format used by *Apktool* to represent Android apps' code) to represent its packages. Then, we cluster all the packages of investigated apps together and rank them through their repeated times. The packages whose size are greater than a given threshold are selected as library candidates.

The code similarity analysis in *step 2*, the ad library conformation in *step 3* and the library representation work in *step 4* are all implemented in Java. More specifically, both of them leverage Soot [49, 50] to achieve their functionality and work in the *Jimple* code level, where Soot is a framework for analysing and transforming Java/Android apps while Jimple is an intermediate representation of Soot. The transformation from Android Dalvik bytecode into Jimple code is powered by Dexpler [51], which currently is available as a plugin in Soot.

5. LibExcluder

So far, based on the process described in the previous section, we are able to harvest a set of common libraries. In order to facilitate the usage of identified common libraries, we design and implement a research-based tool named LibExcluder. The objective of LibExcluder is to remove library code from a given Android app and therefore presenting to existing state-of-the-art approaches a new app version where library code no longer exists. Without any modification (i.e., being non-invasive), existing approaches such as FlowDroid [8] and IccTA [3] can benefit from our work by performing a library-free analysis.

Fig. 5 illustrates the working process of LibExcluder, which takes as input two artefacts: a given Android app and a whitelist of common libraries, and outputs a new app version, which is generally as same as the inputted one except that some code that belongs to any library configured in the whitelist are excluded. The implementation of LibExcluder relies on four main steps: First, it unpacks the given Android app and builds a call

eclipse-project/InterComponentCommunication/ServiceCommunication1/src/edu/mit/icc_service_messages/ActivityMessenger.java

[10] https://ibotpeaches.github.io/Apktool/

```
concrete part: (icc_service_messages,
(ActivityMessenger$1,<init>/[],onServiceConnected/[...],onServiceDisconnected/[]),
(ActivityMessenger,<init>/[...],bindService/[],getSystemService/[],
onCreate/[...],onStart/[...],onStop/[...],sayHello/[...],setContentView/[],unbindService/[]),
(BuildConfig,<init>/[]),
(MessengerService$IncomingHandler,<init>/[...],handleMessage/[...]),
(MessengerService,<init>/[...],getApplicationContext/[],onBind/[...]),
(R$attr,<init>/[]),(R$id,<init>/[]),(R$layout,<init>/[]),(R$string,<init>/[]),(R,<init>/[]))

abstract part: (10,(1,65537/0/0),(1,65537/0/0),(1,65537/0/0),(1,65537/0/0),(1,65537/0/0),
(1,65537/0/0),(2,1/1/2,65536/1/1),(3,0/0/0,1/1/3,65537/0/2),(3,1/1/0,1/2/1,65536/1/0),
(9,0/1/0,0/1/0,0/1/0,0/3/0,1/1/4,4/0/1,4/0/2,4/1/1,65537/0/1))
```

Listing 1: The real representation of package *edu.mit.icc_service_messages*, which contains more elements comparing to the simplified structural tree shown in Fig. 4.

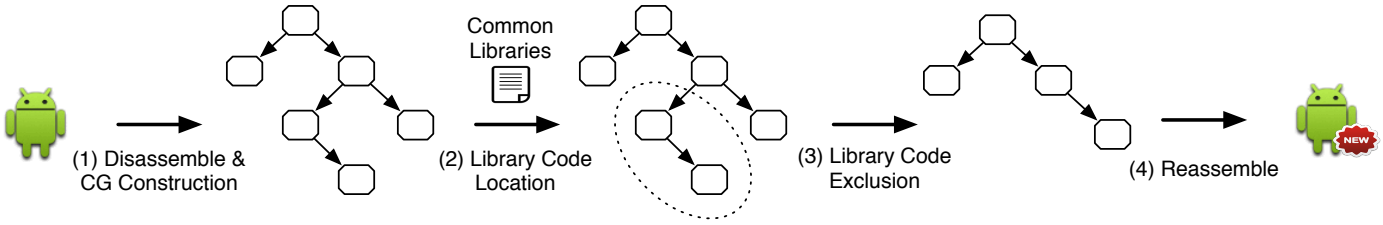


Figure 5: LibExcluder Overview.

graph (CG) based on the unpacked bytecode. Then, it goes through all the libraries configured in the whitelist and attempts to locate all the library code. Next, after identifying all the library-related code, LibExcluder then removes them from the constructed call graph. In practice, in order to make the remaining code self-compilable, the removed library code is set to phantom to ensure that the modified code is still analyzable by static analyzers. Finally, in the last step, LibExcluder reassembles the changed code to a new Android app, which now does not contain any library code. Instead of analyzing the original app, existing analyzers, without any modification, can now focus on the new version to perform a library-free analysis.

LibExcluder is implemented following the same code instrumentation strategy introduced by other approaches such as IccTA [3] and DroidRA [52]. Unfortunately, it shares the same drawbacks as well. For example, we cannot guarantee that the newly generated app version will be executable. Nevertheless, the goal of LibExcluder is not to generate executable Android app, but to generate an app version that is capable of supporting static analysis approaches. As shown in Section 7, LibExcluder is indeed capable of improving the performance of existing static analyzers.

6. Dataset and Results

In this section, we first disclose our evaluated data set in Section 6.1 and then we present our overall findings including both common libraries and also ad libraries in Section 6.2. Finally, we present further statistics on the libraries in Section 6.3 and Section 6.4.

6.1. Dataset

Our data set is made up of 1,455,516 (around 1.5 million) apps that are collected from the official Google market (*Google Play*) over several months. This dataset has already been applied to large-scale experiments on Android researches such as malware detection [41, 53, 34] and repackaged apps detection [15]. We have sent all the apps into VirusTotal to check whether they are malicious or not. Among the 1,455,516 apps, 311,490 (nearly 21%) of them are flagged by at least one anti-virus product hosted on VirusTotal while 65,079 (nearly 4%) apps are flagged by at least five anti-virus products.

6.2. Overall Results

Table 1: Summary of our investigation results.

Type	Number
#. of packages (total)	7,710,505
#. of packages (distinct)	676,674
#. of packages ($N_{shared\_apps} > 10$)	19,725
#. of packages (one segment)	613
#. of packages (obfuscated)	1,461
#. of packages (prefix of others)	919
Size of final set of candidate common libraries	**16,732**

Table 1 illustrates the overall results of our investigation on a data set of around 1.5 million apps. In total, we collect 676,674 distinct package names, where we filter out 656,949 package names that are used by at most 10 apps, leading to a set of 19,725 package names. We further dismiss 2,993 from consideration by applying our library refinement process. Those

2,993 package names are composed of 613 one segment packages, 1,461 obfuscated packages and 919 packages that are prefixes of other packages. Finally, we perform pairwise similarity analyses for 16,732 packages. For each package, we randomly select 10 pairs of apps to do the comparison. As long as there are positive results, we consider it as a common library and vice visa.

6.2.1. Results of Common Libraries

Table 2: Results of common libraries with different thresholds: t_p for package-level and t_a for app-level. Common libraries are select if and only if their package-level similarities are bigger than t_p while their app-level similarities are smaller than t_a.

$t_p \backslash t_a$	0.1	0.2	0.3	0.4
0.9	1,113	2,148	3,173	4,072
0.8	1,363	2,564	3,715	4,685
0.7	1,573	2,898	4,117	5,144
0.6	1,735	3,179	4,452	5,509

Our common libraries selection is actually depending on the two thresholds introduced in Section 4: t_a for app-level similarity and t_p for package-level similarity. The precision of our results is positively correlated to t_p while negatively correlated to t_a. Indeed, the bigger t_p is, the higher the probability that a given candidate library is an actual common library, giving the assumption that libraries are not modified when they are used among apps. On the other hand, the smaller t_a is, the lower the probability that the compared two apps are repackaged from one to another. Recall that if two apps are repackaged from one to another, the similarity of packages would become meaningless, as in this case, most packages would be the same, without being necessarily common libraries.

Table 2 illustrates the results of common libraries with different thresholds. The final number of common libraries range from 1,113 to 5,509. To better refer to our results in the remainder of the paper, we name $CL_{p,a}$ the set of Common Libraries that are selected with the thresholds t_p and t_a. For example, $CL_{6,4}$ stands for a "loose" set of common libraries we harvest with $t_p = 0.6$ and $t_a = 0.4$. $CL_{9,1}$ stands for a more precise set of 1,113 common libraries we harvest with $t_p = 0.9$ and $t_a = 0.1$, which although the number of libraries is smaller than $CL_{6,4}$, contains potentially less false positives (i.e., more precise than $CL_{6,4}$). Therefore, in this work, we consider $CL_{9,1}$ as the default common library set and recommend users to leverage this set for their future investigations.

6.2.2. Results of Ad Libraries

We then distil ad libraries from the previously harvested common libraries. We start from the $CL_{6,4}$ library set and performs two types of refinement: 1) ad-related keywords matching and 2) ad characteristic-based investigation. The refinement results are presented in Table 3.

Ad-related keywords matching. By following the process described in Section 4.3, we were able to automatically harvest 275 ad libraries.

Table 3: Results of ad libraries.

Description	#. of Libraries
Ad-related keyword matching	275
Ad characteristic-based investigating	822
Merge (conservative ad libraries)	1050
Manual confirmation (keyword matching)	222
Manual confirmation (characteristic investigating)	137
Merge (precise ad libraries)	240

Ad characteristic-based investigating. We have observed three characteristics that ad libraries may have in Section 4.3. Fig. 6 shows the results of our investigation. Among the 5,509 libraries in $CL_{6,4}$, 1,248 of them request the *INTERNET* permission, 1,560 have declared *View* gadgets and 1,388 have declared components. The intersection results are also illustrated in Fig. 6. In this work, we take the intersection of all the three characteristics as potential ad libraries, leading to a set of 822 ad libraries.

In the next step, we merge the aforementioned two ad libraries sets, leading to a set of 1,050 ad libraries. In the remainder of the paper, we name this set AD_{1050}.

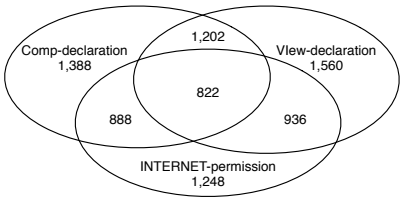


Figure 6: Investigation results of different characteristics for ad libraries. Besides the 822 ones presented in this figure, we harvest in total 1,050 ad libraries, where 275 of them are collected via an ad-related keyword matching approach.

Manual confirmation. As far as we know, AD_{1050} is currently the largest set of ad libraries existing in the community. However, because we start from $CL_{6,4}$, mainly to start with the biggest set (minimizing the miss of libraries), AD_{1050} may contain false positives. To this end, we perform a fast but aggressive manual refinement, where only clear ad libraries[11] are taken into account. As a result, 240 libraries are confirmed as ad libraries[12], hereinafter we refer to this set as AD_{240}. These 240 ad libraries are found to be accurately labelled, forming a golden set of ad libraries. We argue that a golden set of ad libraries is important, which plays as a core base that makes it possible for other approaches to also yield reliable datasets/results. Indeed, if a non-ad library is considered to be an ad library, all the findings observed based on the ad library would be consequently false alarms.

[11] We search each library on Google and manual go through the top 10 results. We consider a package is indeed an ad library as long as its corresponding web pages have explicitly claimed that it functions advertisements.

[12] This does not mean the remaining 810 libraries are not ad libraries (i.e., not false positives). It is likely that our fast refinement approach is not enough to reveal all the library packages.

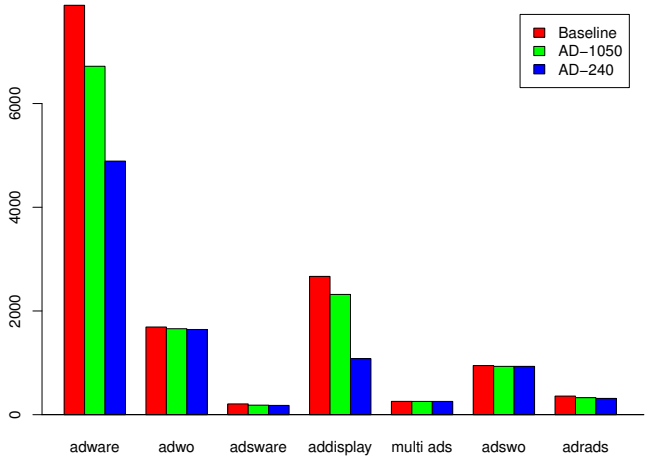


Figure 7: Investigation results of comparing our ad libraries to the adware results of VirusTotal.

Completeness of our harvested ad libraries VirusTotal is a free service that hosts about 60 antivirus products for analyzing suspicious files, including Android apps. Along with entirely malicious apps, VirusTotal is also able to identify adware and provide information on the labels. However, AV labels are not homogeneous, and there is no standard for naming malware and adware. After manually inspecting several results of VirusTotal, we have observed seven keywords (adware, adsware, addisplay, adswo, adwo, adrads, and "multi ads") that are commonly leveraged by VirusTotal AV to tag adware.

In this study, we first select a set of apps that are flagged by VirusTotal as adware, and then we inspect whether those apps could have been tagged as adware based simply on package matching with our harvested libraries. In this study, we consider 10,000 randomly sampled apps which are flagged by at least one antivirus product of VirusTotal (the flagged labels are not necessarily for adware). Among the 10,000 apps, 8,120 (81.2%) of them are flagged as adware following the keywords described above. Based on the two ad sets that we have harvested before, we are able to flag 5,045 of them for AD_{240} and 6,916 of them for AD_{1050} as adware, giving a completeness of 62% and 82%, respectively.

Fig. 7 presents the fine-grained results, categorized through different ad-keywords. Our harvested ad libraries perform almost perfectly for five keywords out of the total seven keywords. However, the performance on "adware" and "addisplay" keywords are less stable, indicating that our harvested ad libraries are still missing some less widespread libraries.

We now evaluate the recall of VirusTotal in terms of the usage of advertisements by comparing with our findings. In order to evaluate the recall of VirusTotal, we have to collect a set of apps that are indeed adware. To this end, we manually select and confirmed 30 ad libraries from our findings to perform the experiments. These 30 libraries are actually used by 33,475 apps in our data set, for which we send all of them to VirusTotal.

Among the 33,475 apps that we examine, 19,695 (59%) of them are flagged as goodware by VirusTotal, meaning one of the anti-virus products hosted by VirusTotal have flagged them as such. For the remaining 13,780 apps, VirusTotal has flagged them as malicious, where 11,713 (85%) of them are flagged as adware, while the remaining 2,067 apps are flagged as malware (non-adware). In this work, we take the 19,695 goodware and the 2,067 malware as false negatives, resulting in a recall of 35%, illustrating at the moment the results of VirusTotal are far from good. In other words, there is still a huge space for VirusTotal to improve, in order to perform a sound analysis.

Table 4: Investigation results of the 30 ad libraries we select. Besides, we have listed the popularity of top used 10 libraries (out of the selected 30 libraries).

Seq.	Name	Apps	VirusTotal	Recall
1	com.adsdk.sdk	9439	2589	27%
2	com.adfonic.android	6469	1658	26%
3	com.adwhirl	6052	1317	22%
4	com.jirbo.adcolony	4033	1332	33%
5	com.applovin.adview	5895	2103	36%
6	com.jumptap.adtag	3576	941	26%
7	com.purplebrain.adbuddiz	2685	882	33%
8	com.huntmads.admobadaptor	2488	515	21%
9	com.tapit.adview	2371	583	25%
10	com.adwo.adsdk	1949	1886	97%
Sum	$(1 \rightarrow 10)$	30,301	10,225	34%
Sum	$(11 \rightarrow 30)$	5,807	3,206	55%
Total	$(1 \rightarrow 30)$	33,475	11,713	35%

Table 4 shows more details of the investigation results, in which we have shown the popularity of the top 10 used ad libraries (out of the 30 libraries we select previously). Respectively, we also investigate the recall of VirusTotal for each ad library. In most cases, the recall of VirusTotal is less than 40%, which is in line with the overall results. However, for package *com.adwo.adsdk*, 97% apps that contain it have been successfully flagged, showing that it is not impossible to aggressively identify ad packages from Android apps.

The aforementioned evidence suggests that, although our harvested ad libraries are currently the largest publicly accessible set, it is not yet complete enough to cover all ad packages. Nevertheless, we believe that our harvested libraries still constitute a significant set for other to boost analysis.

6.3. Popularity of common libraries

Fig. 8 lists the top 20 common libraries and indicates, for each, the number of apps in which they are used. The top used library is *com.google.ads*, which is used by 247,394 apps (nearly 17%) of our data set. Moreover, the results suggest that developers often use libraries which are proposed by popular (well-known) companies such as Google or Facebook.

> The most used common library in Android apps is *com.google.ads*, an adverstisement library included in nearly 17% of apps.

6.4. Proportion of Library code in app code

We then look into the percentage of Android apps code which comes from libraries. To this end, we consider the libraries present in $CL_{9,1}$ and a set of 10,000 apps randomly

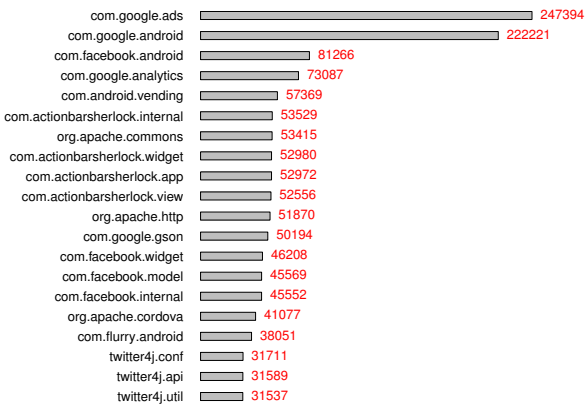


Figure 8: Popularity of the top 20 common libraries in our investigation, and the number of apps in which they are used.

selected from our initial set of apps. For each app, we compute the size of the $CL_{9,1}$ libraries ($size_{lib}$, in bytes) presented in the app and the size of the whole app ($size_{app}$). We finally compute the portion p of the use of common libraries through $p = size_{lib}/size_{app}$. The experimental results vary from 0 to 0.99, giving a median value of 0.41. Among the 10,000 apps, 4,293 (42.9%) of them have used more code in libraries than in their real logic ($p >= 0.5$). These results show that Android apps are indeed using common libraries pervasively.

> 42% of our sampled app packages contain more common library code than specific app code. On average, 41% of an Android app code is contributed by common libraries.

6.5. Representation of Identified Libraries

Given that the time needed to analyse the apps, the bandwidth required to collect them and the storage capacities to store the data put limitations for the experiments, we could not run LibRepresenter on all the apps available in Androzoo. Nevertheless, we run LibRepresenter on 200,000 Google Play apps that are randomly selected from our original dataset. These 1,051 libraries are leveraged and are spread with 21,150 different versions (i.e., variants). Fig. 9 plots the distribution of the number of variants among the involved libraries. The median and the average number of variants are 5 and 30, respectively. Overall, 881 (or 84%) libraries have at least two variants spread in the Android ecosystem.

Table 5 depicts the top 10 libraries ranked based on their number of variants discovered using LibRepresenter. The first column presents the library name while the following two columns present the number of variants as well as the actual number of usages, respectively. Generally, the more usages a library gets, the more variants it may spread in the Android ecosystem. We further perform a correlation study between the number of mutants and the number of total usages by computing the Spearman's *rho* (also known as Spearman's rank correlation coefficient), which is a non-parametric measure aiming at assessing

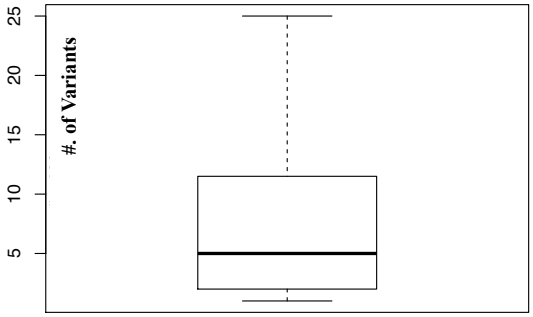


Figure 9: Distribution on the number of library variants.

statistical dependence between two variables using a monotonic function. The experimental result (*rho* = 0.57) statistically confirms our observation: the number of variants is moderately correlated with the number of library usages.

Table 5: Top 10 libraries ranked based on their number of variants appeared in the randomly selected 200,000 apps.

Library	# Variants	# Usages (Apps)
org.apache.http	4120	45175
com.badlogic.gdx	3288	6659
com.handmark.pulltorefresh	3180	7286
com.android.vending	1643	49676
com.google.common	804	10913
com.markupartist.android	539	1976
org.osmdroid.views	442	2487
com.androidquery.callback	420	8192
com.heyzap.sdk	409	2315
net.robotmedia.billing	348	936

Furthermore, we go one step deeper to manually check if the listed 10 libraries are open sourced. Our hypothesis is that a large number of variants appearing to those libraries could be the reason that they are open sourced, which provide various opportunities for app developers to leverage at any point in the progress of the library. Through manual investigation, we confirm that our hypothesis is true where nine out of the top 10 libraries (except *com.heyzap.sdk*) is open source.

> Around 84% of Android libraries have at least two distinct variants. Generally, the number of variants is correlated with its total usage (e.g., by how many apps) and open sourced libraries trend to have more variants.

7. Empirical Investigations

We now empirically explore the five research questions presented in the general introduction section of this paper.

7.1. RQ1: Repackaging Detection

Recall that in Section 3, we have shown that repackaging detection approaches are likely to yield false positives and false negatives if code contributed by common libraries are not taken

into account. We provide more empirical evidence of such threats.

For our experiments we rely on a set of pairs of apps that we have collected in the similarity analysis of Step 2 and regrouped into two categories: the first category, *FNData*, contains 761 pairs of apps where the similarity score for each pair is below 50% while the second category, *FPData*, includes 1,100 pairs of apps where the similarity score of each pair of apps is over 80%. Given the previously justified threshold of 80% for deciding on repackaging (cf. Section 3), we assume that all pairs in *FPData* are repackaged pairs while those in *FNData* are not. We now explore again the similarity scores of the pairs when excluding from each app the common libraries (in CL_{91}) they may include.

False positives elimination. Among the 1,100 pairs of apps in *FPData*, 1,029 (93.5%) remained similar above the 80% threshold. 71 (6.5%) pairs of apps now have similarity scores below 80%, and can no longer be supposed to be repackaged pairs. We manually verified 10 of pairs and found that they are indeed not repackaged.

False negatives re-classification. Among the 761 pairs of apps in *FNData*, 110 (14%) have higher similarity scores, among which 2 pairs are now beyond the threshold of 80% which would allow to re-classify them as repackaged pairs. We have manually verified and confirmed that these two pairs of apps are piggybacked pairs: one pair was previously used in our motivating example section (Fig. 1b).

> The appearance of common libraries infects the accuracy of repackaged app detection approaches. If common libraries are not taken into account, it is possible that both false positive rate and false negative rate for repackaged apps detection approaches can be reduced.

7.2. RQ2: Machine Learning for Malware Detection

We investigate the case of machine-learning based approaches for Android and study the impact of ignoring or taking into account common libraries on the accuracy of prediction. We consider a case study based on MUDFLOW [7] and its dataset. This dataset contains sensitive data leaks information for 15,096 malicious apps and 2,800 benign apps. MUDFLOW is a relevant example as the authors have originally foreseen the problem with libraries and thus attempted to exclude a small set of ad libraries in their experiments. With our large harvested dataset of common libraries, we investigate the performance gap that can be achieved by excluding more known libraries.

MUDFLOW performs machine learning to mine apps that use sensitive data abnormally. More specifically, MUDFLOW takes each distinct type of sensitive data leak (from pre-defined *source* to *sink*) and performs a one-class classification to detect abnormal apps. One-class classification is realistic in their experimental settings with their imbalanced dataset (they have much more malicious apps than benign apps).

Since our goal is not to replicate MUDFLOW (along with its sophisticated library-unrelated parameters), but to evaluate the impact of excluding common libraries for machine learning, we propose to implement a slightly simplified approach for our experiments. Unlike MUDFLOW, which constructs a training set based on benign apps and then applies it to predict unknown apps, we simply perform 10-fold cross-validation in our evaluation. It is, therefore, worth to emphasise that our approach serves only to demonstrate the impact of excluding libraries on the performance of machine learning-based malware detection approaches. As we are working on the same imbalanced data, we also choose one-class classification.

We have performed four types of experiments, which are detailed below:

- **E5:** We evaluate on all the 15,096 malicious apps. The feature set is made up of distinct sensitive data leaks. Instead of taking into account *source* and *sink* methods, each data leak is represented through the *source* and *sink* categories (e.g., methods like *Log.i()*, *Log.e()* are represented as category *LOG*).
- **E6:** This experiment has similar settings as in **E5**, except that such sensitive data leaks that are contributed by the 12 ad libraries considered by MUDFLOW are excluded.
- **E7:** This experiment has similar settings as in **E6**. In this case, however, the excluded set of libraries is the most constrained set of 1,113 libraries harvested in our work.
- **E8:** This experiment has similar settings as in **E7**. In this case, however, the excluded set of libraries is constituted by libraries selected based on a more loose definition of libraries. The excluding set contains 5,509 libraries, which may include a number of false positives.

The results of these four experiments are shown in Table 6. Comparing **E7** to **E5**, the accuracy is indeed increased, which suggests that the presence of common libraries code could confuse machine learning-based classifier. However, the accuracy remained the same between **E6** and **E5**, suggesting that the MUDFLOW *whitelist*, which contains 12 libraries, is too small to impact the final results. Interestingly, with our largest set of libraries, the accuracy of **E8** decreases slightly comparing to that of **E7**. This suggests that the precision in common library identification is important: excluding non-library code will eventually decrease the overall performance for machine learning.

Table 6: Results of our machine learning based experiments performed on the data set provided by MUDFLOW [7].

Seq.	#. of Features	Excluding Libs	Accuracy
1	109	0	81.85%
2	109	12 (MUDFLOW [7])	81.85%
3	109	1,113 ($t_p = 0.9$,$t_a = 0.1$)	83.10%
4	108	5,509 ($t_p = 0.6$,$t_a = 0.4$)	83.01%

The appearance of common libraries infects the accuracy of machine learning based Android malware classifiers. If common libraries are ignored, the accuracy of the ML-based malware detection approach could be increased. Correspondingly, if developer code (e.g., non-library code) is excluded, the eventual precision would be decreased.

7.3. RQ3: Static Analysis Performance Improving

Static analysis can be time-consuming, which could simply result in timeout or out of memory for its analysis. As an example demonstrated by Avdiienko et al. [7], with a server running 730 GB of RAM and 64 Intel Xeon CPU cores, their static approach sometimes could not even analyze a single app in a day. Therefore, under a practical assumption where the analyzing time and resources are limited, static approaches generally need to compromise on their analyses in order to strike a good balance on their results. One possible compromise to fast static analysis is to avoid the analysis of unnecessary code, e.g., by focusing only on the main app code while ignoring common library code. To achieve this purpose, we believe that our collected common libraries could be leveraged as a whitelist to restrict the analysis of library code.

Towards verifying this assumption, we perform an experimental investigation that empirically compares the results of two experiments, where one launches a static analyzer on a set of selected Android apps while the other launches the same static analyzer on sliced versions of the selected apps where common libraries are excluded using LibExcluder. To this end, we select a state-of-the-art tool called FlowDroid [8], a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analyzer for detecting privacy leaks in Android apps, to be the static analyzer and 50 Android apps that are randomly selected from a set of top-ranked apps including Whatsapp, Booking, Facebook, etc. Fig. 10 plots the distribution of the size of the selected 50 apps, where the median and mean size are 6.2 MB and 11.2 MB, far exceeding the overall average size of available Android apps. The maximum size, given by app *com.popcap.pvz2cthdhwct*, even reaches 186 MB, demonstrating that the selected apps are not toy apps for out experiments.

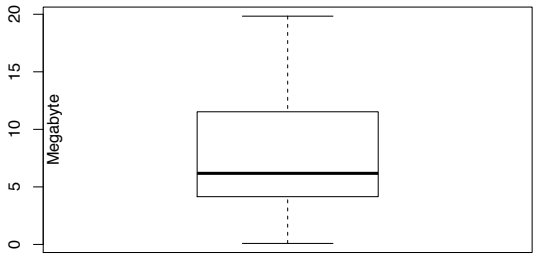


Figure 10: Distribution on the Size of Selected Apps.

We launch FlowDroid on all the 50 original Android apps (hereinafter referring as **Experiment 1**) and on their sliced versions (hereinafter referring as **Experiment 2**) where common libraries are excluded using LibExcluder. In order to ensure a fair comparison between these two experiments, we have performed our experiments under the same settings: (1) The path algorithm is set to be context-sensitive, which allows the analysis to keep tracking the calling context when a target function call is analyzed [54]; (2) The analysis is conducted without code elimination so as to avoid potential bias, as static analyzers may also intelligently exclude irrelevant common library code; (3) The static analysis is constraint to a single thread, which is important to ensure a fair comparison on the final consumed time of each experiment; (4) The maximum heap-size (i.e., memory) a single run can reach is 20 GB; (5) The maximum time a single run can sustain is set to be 20 minutes, which is lower than some state-of-the-art work (e.g., 24 hours are used by Avdiienko et al. [7]) but we believe is fair enough to fulfill our experiments. Indeed, our main purpose is to investigate the performance improvement when common libraries are not considered and consequently to verify a possible approach (i.e., excluding library code) that would allow existing static analyzers to finish within limited resources (e.g., time limits). In order to mitigate the impact of random errors, each experiment has actually been launched for three times and the average consumed time is considered.

Fig. 11 illustrates the number of successfully analyzed apps between the aforementioned two experiments. FlowDroid is able to finish for 16 unmodified apps whereas the number of successfully analyzed apps increases to 31 when the libraries are excluded. The success rate has increased for almost 100% cases when library code is not considered.

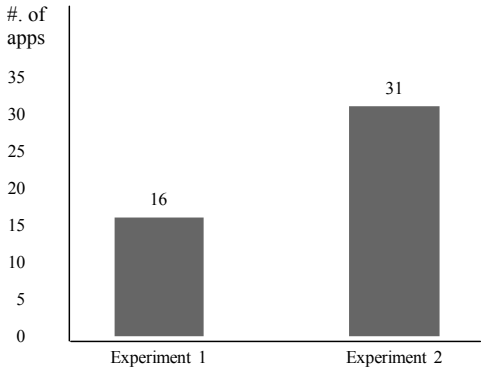


Figure 11: The Number of Successfully Analyzed Apps between Experiment 1 (Original App) and Experiment 2 (i.e., Library Code is Excluded).

Table 7 enumerates the experimental results (e.g., the time required to finish the analysis and the number of leaks identified) of selected apps that have been successfully analyzed in both Experiments 1 and 2. Columns 2-5 show whether the Android apps have accessed native code, dynamic loaded code, crypto code, and reflective code, where most of the apps have somehow accessed at least one of the aforementioned type code, suggesting again that the selected apps are complicated ones (i.e., our experiment is not based on toy apps but on real-world Android apps). Column 6 presents the number of classes that

are excluded by LibExcluder. Columns 7-11 illustrate the consumed time and detected leaks respectively for Experiments 1 and 2.

As shown in columns 7-9, Experiment 2 generally spends less time than Experiment 1, which is also illustrated by the time improvements sub-column (i.e., column 9), showing that excluding common libraries is promising to improve the time performance of static analyzers. In columns 10-11, the experimental results further show that unfortunately excluding common libraries may slightly impact the detection of privacy leaks, although time performance is improved. Nevertheless, under finite time and resources constraints, excluding common libraries allows more apps to be successfully analyzed by static analyzers. Although only partial results might be yielded, it is certainly better than the situation where no results are yielded (but simply timeout exceptions).

Observant readers may have noticed that there are several corner cases that have negative time improvements after excluding common libraries and have more leaks uncovered although less code is actually analyzed due to library code exclusion. For app Plants VS. Zombies (package: *com.popcap.pvz2cthdhwct*), the negative improvement could be explained by some random error, as there is in fact no library code excluded and the total time consumed is relatively small. However, this explanation cannot be simply applied to app *com.snda.wifilocating*, for which a significant number of classes are excluded while more time is spent. We, therefore, contact the authors of our applied static analyzer FlowDroid for potential hints. We also resort to the authors of FlowDroid for explaining the experimental results of app *com.huawei.phoneservice*, where less code is analyzed while more privacy leaks are uncovered. The possible explanation answered by the authors of FlowDroid indicates that FlowDroid's over-approximation strategy may cause the aforementioned corner cases. Because the library code is missing, FlowDroid has to safely assume that all the tainted data flowing into library code would remain tainted, which may cause more paths explored and consequently more time spent or more privacy leaks identified. Nevertheless, the small number of corner cases are specifically related to static taint analysis. Other general static analysis approaches would not be impacted and thus our finding, where excluding common libraries could improve the performance of static analysis, is still held.

As indicated in Fig. 11, 15 apps cannot be analyzed by FlowDroid in Experiment 1. Towards understanding whether these failures are due to finite time constraint, we resort to replicate Experiment 1 by increasing the timeout from 20 to 60 minutes (hereinafter referring as **Experiment 1'**). Table 8 present the replicated results for Experiments 1', where four additional apps are successfully analyzed by FlowDroid, demonstrating that excluding common libraries could be a promising alternative instead of increasing the analyzing time for improving the performance of static code analysis approaches.

> It is a possible alternative to exclude common libraries for improving time performance of static code analysis. Together with the aforementioned two experiments, all of these case studies suggest that library code can indeed mislead Android analysis, and our harvested set of common libraries is promising to be used to improve the performance of state-of-the-art approaches.

7.4. RQ4: Benign vs. Antivirus-flagged Apps

With this research question, we are interested in investigating the differences in library usage between benign and antivirus-flagged apps. To this end, we randomly select 20,000 apps, 10,000 benign and 10,000 flagged by at least one anti-virus product hosted by VirusTotal[13].

Among the 20,000 apps, 5,424 benign ones and 8,580 flagged ones use at least one common library. In total, 892 out of the 1,113 $CL_{9,1}$ common libraries are used. Figure 12 shows the boxplot with the number of common libraries used by each app in both categories. The median value for benign apps is 1 whereas the median value is 3 for antivirus-flagged apps. We confirm that this difference is significant by performing a Mann-Whitney-Wilcoxon (MWW) test. Benign apps thus use fewer libraries than anti-virus flagged ones.

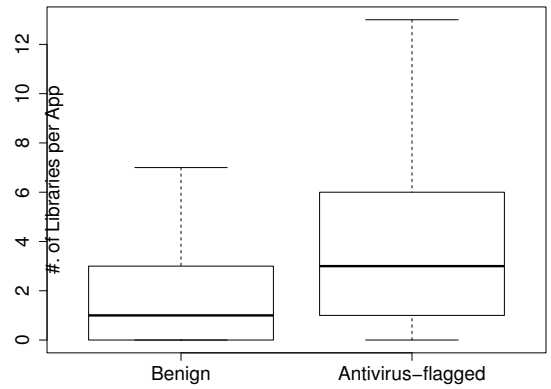


Figure 12: Library Usage between benign and antivirus-flagged apps.

We further study whether this is related to advertisement libraries. Among the 240 libraries in AD_{240}, 98 of them are used by 1,332 benign apps and 3,209 antivirus-flagged apps. Figure 13 shows the boxplot which suggests that antivirus-flagged apps contain more ad libraries than benign apps.

In addition to the quantitative comparison, we investigate whether the appearance of the collected common libraries can be used to discriminate antivirus-flagged apps[14] from benign apps through machine learning-based malware classification. To this end, we leverage *RandomForest* algorithm [55] to perform 10-fold cross-validation on the 20,000 apps considered above. Each app is represented by a feature vector where the

[13] https://www.virustotal.com

[14] In this work, we consider those antivirus-flagged apps as potentially malicious.

Table 7: Experimental Results of Successfully Analyzed Apps in Experiments 1 and 2.

App Package Name	Native Code	Dynamic Code	Crypto Code	Reflection Code	Removed Classes	Time (in seconds) Experiment 1	Time (in seconds) Experiment 2	Time (in seconds) Improvement	Leaks Experiment 1	Leaks Experiment 2
com.inmobi.oem.core.services	0	1	1	1	126	28	27	3%	5	5
com.secdroid.secretcodetester	0	0	0	1	810	4	3	33%	0	0
com.happyelements.AndroidAnimal	0	0	0	1	9	3	3	0%	0	0
com.google.android.apps.maps	1	0	1	1	45	23	23	0%	3	3
com.sdu.didi.psnger	1	1	1	1	1712	21	13	61%	5	4
com.huawei.openalliance.giftpackage	0	0	0	1	168	4	3	33%	0	0
com.facebook.katana	1	0	1	1	17	33	33	0%	0	0
ru.mail.mailapp	0	0	1	1	789	56	43	30%	36	22
com.huawei.phoneservice	1	1	1	1	644	249	245	1%	416	463
ru.sberbankmobile	1	1	1	1	1822	152	24	533%	74	41
com.huawei.svn.hiwork	1	0	1	1	573	9	7	28%	2	0
net.zedge.android	1	1	1	1	1414	152	90	68%	39	18
com.popcap.pvz2cthdhwct	0	0	0	1	0	4	5	-20%	1	1
com.vmall.client	1	1	1	1	1021	437	420	4%	315	294
com.snda.wifilocating	1	1	1	1	535	850	1045	-18%	168	161
com.sohu.tv	1	0	0	1	219	34	13	161%	21	6

Table 8: Experimental Results by giving 60 minutes as timeout for Experiments 1.

App Package Name	Experiment 1	Experiment 2
com.tripadvisor.tripadvisor	2476 (s)	70 (s)
com.contextlogic.wish	1622 (s)	26 (s)
com.dianping.v1	1312 (s)	32 (s)
com.tuniu.app.ui	2716 (s)	73 (s)

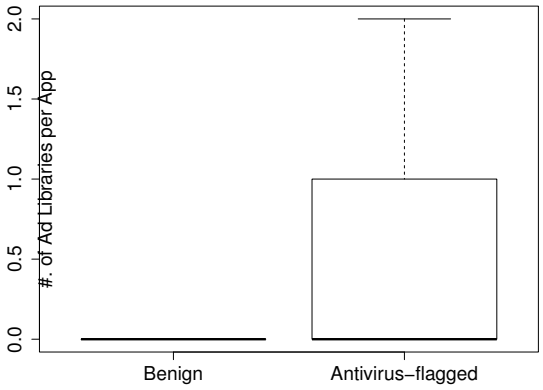


Figure 13: Ad Library Usage between benign and antivirus-flagged apps.

package name of each common library is taken as a feature. Recall that we have collected 892 libraries from those 20,000 apps. Therefore, our machine learning-based experiments contain 892 features. Table 9 illustrates the results of our machine-learning-based malware detection in 4 different settings, which are

- **E1:** Machine-learning experiments using the entire set of 20,000 apps.
- **E2:** Some apps do not contain any of our harvested libraries, leading to empty feature vector which can lead to mis-classifications. Thus, we conduct an experiment taking into account only apps which include at least one of our collected common libraries.
- **E3:** We replay the experiment E2 where we ensure that there is no class imbalance: we randomly select 5,424 apps from the 8,580 antivirus-flagged apps.
- **E4:** Similarly to experiment E3, we repeat E2 with a balanced dataset by oversampling the Benign set, using the Synthetic Minority Oversampling TEchnique (SMOTE) [56].

The results of all experiments showed in Table 9, indicate a good discriminative power of library features for machine-learning based detection of anti-virus (AV) flagged apps.

Table 9: Results of our machine-learning-based detection of AV-flagged apps.

Exp.	Benign set	AV-flagged set	Precision	Recall	F-Measure
E1	10,000	10,000	0.841	0.835	0.835
E2	5,424	8,580	0.861	0.861	0.861
E3	5,424	5,424	0.862	0.860	0.860
E4	8,580	8,580	0.875	0.873	0.873

Benign apps use significantly less common libraries than AV-flagged apps. The combinations of libraries in apps can be discriminated between benign and AV-flagged apps and hence can be leveraged by anti-virus products to predict new suspicious apps.

7.5. RQ5: Library Obfuscation

This research question considers the second type and attempts to investigate if our collected library representations are capable of identifying those libraries' obfuscated counterparts. In particular, among the 200,000 selected apps, we find that 58,388 of them are obfuscated (they contains packages having a segment with only one character). Since we also want to identify obfuscated library versions, which clearly will not match any of our collected library representations, we represent all the available packages. As a result, every obfuscated package is represented by a unique string abstraction that is quickly comparable to other abstractions and is searchable among a large set of abstractions.

Table 10: Top 10 obfuscated libraries ranked based on their number of variants.

Library	# Variants	# Usages	Obfuscation
com.android.vending	51	2,145	a.a.a,b.wsaft,com.a.a,com.a.b,com.android.a,com.android.b,com.android.c com.b.a,com.b.b,com.c.a,com.havos.g,com.m,com.r,i.e,k.k,k.k.mixjam,l.l,l.l.l ,u.best
com.google.gdata	24	454	a.b.a,com.a.a,com.a.b,com.b.a,com.b.b,com.b.c,com.c.a,com.d.a,com.google.a, com.google.b,com.google.c
com.android.gallery	20	34	cn.x6game.a,com.A.A,com.a,com.a.a,com.android.a,com.b,com.c,com.d,com.d.b,com.e com.pacsplus.a,com.seasgarden.a,com.slacker.f,myobfuscated.r,myobfuscated.w net.simonvt.a,pl.wp.b,rcs.akbd.f,vn.foodmob.j,x.pandafishing
com.nineoldandroids.util	19	896	a.a.b,a.b,com.a.b,com.b.b,com.c.b,com.d.b,com.e.b,com.f.b,com.g.b,com.nineoldandroids.a com.nineoldandroids.b,whyareyoureadingthis.A,whyareyoureadingthis.F whyareyoureadingthis.G,whyareyoureadingthis.y
com.kakao.sdk	19	41	b.a.a,com.a.a,com.a.b,com.aquafadas.a,com.aquafadas.c,com.aquafadas.f,com.aquafadas.g, com.babywhere.a,com.divmob.c,com.easy3d.a,com.easy3d.b,com.ipc.f,com.lewisj.a,com.netgate.r, com.spindle.c,com.ubermedia.e,com.viettel.a,com.viettel.c,myobfuscated.s,sg.radioactive.x
oauth.signpost.commonshttp	15	39	c.a,com.a,com.c.ctv,com.cleanmaster.h,com.cleanmaster.j,com.cleanmaster.k,com.e.sjs, com.glassdoor.a,com.ijinshan.a,com.smschatheads.l,com.softek.a,com.spindle.f,com.spindle.h, com.spindle.k,com.tecace.a,com.xiaomi.a,me.kiip.j
oauth.signpost.signature	14	56	a.a.a,a.a.d,a.a.e,b.a.d,b.a.e,c.a.d,c.a.e,com.prosegur.a,com.ubermedia.d,d.a.d,myobfuscated.r
org.apache.http	11	110	a.a.a,a.a.b,b.a.a,org.a.a,org.apache.b,org.b.a
com.squareup.otto	11	159	com.a.a,com.b.a,com.b.b,com.c.a,com.d.a,com.e.a,com.e.b,com.f.a,com.f.b,com.g.b,com.h.a, com.squareup.a,com.squareup.b
com.google.common	11	59	com.a.a,com.a.b,com.b.a,com.d.a,com.google.a,com.google.b,com.google.c

Among the 58,388 obfuscated apps, we eventually collect 1,624,835 package abstractions, from which we identify that 5,045 abstractions share the same representation of our collected library representations. Those 5,045 abstractions are account for 4,165 distinct apps and for 84 unique libraries. Table 10 highlights the top 10 obfuscated libraries among the 84 identified ones. This ranked list is quite different from the one presented in Table 5, where the rank is computed based on the number of library variants without considering obfuscation. Nevertheless, out of 10 libraries shown in Table 5, the fact that only three of them have been presented in Table 10 demonstrates that there is probably no correlation between the popularity of common libraries and the obfuscation-rate of library-featured apps.

Overall, this result demonstrates that our pre-collected library representations could be leveraged as a benchmark to identify obfuscated library versions. Furthermore, it also suggests that 5,045 usages of libraries would have been missed if obfuscation is not taken into account by a library-related study.

> Our approach (i.e., LibRepresenter and our pre-collected library representations) is promising to identify obfuscated library counterparts.

8. Discussion

8.1. Precision of Identified Libraries

We recall that, in this work, common libraries are collected based on their appearances in Android apps while accounting for the similarities among the apps that use them. For instance, libraries identified in the $CL_{9,1}$ set have been used by x apps (with $x >= 10$). Given a library $l \in CL_{9,1}$, although the similarities of the x apps are less than 10% (i.e., they are unlikely to be different versions of the same app), the code structures of l are very similar among the x apps who have used l (i.e., at least 90%). Nevertheless, it is still not immediately possible to estimate the accuracy of the collected set of libraries. We thus resort to a supplementary dataset of around 4 million apps to cross-assess the usage recurrence of the libraries identified in the $CL_{9,1}$ set. Our experiments reveal that over half of the libraries in $CL_{9,1}$ are leveraged in at least 100 apps, with a median recurrence number of 110. This result demonstrates that the libraries collected in this work are likely to be true libraries.

Furthermore, we go one step deeper to manually verify the $CL_{9,1}$ set so as to understand how precise is our approach towards identifying common libraries. To this end, we send the packages that have a recurrence number less than 100[15] in the $CL_{9,1}$ set to several code repositories such as Github, GoogleCode, and Maven and check its usage. Since it is still difficult for a human to decide if a given package is an actual library, we resort to the following approach to confirm common libraries: Given a candidate package, if it appears to be associated with an open-source project, since every app developers could access into the project, we consider it as a common library. Otherwise, if it is not associated with any open-source project, we check if it appears to be imported (via the import Java keyword) in at least three different projects and is different from the main package name (i.e., usually starting with the domain name, in reverse order, of the company who have developed the project), we consider it as a common library as well. Among the 1,113 packages presented in the $CL_{9,1}$ set, our manual verification confirms that at least 984 of them[16] are indeed libraries, giving a precision of 88%. This result further suggests that our methodology, based on library appearances and similarities, is quite reliable towards automatically identifying common libraries.

[15] They are less likely to be common libraries compared to such ones that have a recurrence number over 100.

[16] We have also made this library set publicly available online.

8.2. Additional Findings

Our investigation into libraries have additionally revealed several interesting findings on the use of libraries:

Malware writers often name their malicious components after famous and pervasively used libraries from reputed firms: e.g., the *DroidKungFu* malware family spreads malicious payload within a package called *com.google.update*. Our similarity analysis allowed to detect such fraud by further investigating outliers.

Well-used libraries are often used as the compromising point for malicious apps. Indeed, among 500 hundred repackaged malicious apps, which are built via 1) unpack benign apps, 2) injecting malicious payloads and 3) repack the modified code back to a new app, our investigation reveals that 65 out of the 500 (i.e., 13%) apps have been compromised where their leveraged *unity3d* is modified [57]. Table 11 enumerates the compromised details of those modifications related to *unity3d* library.

Table 11: Compromised Cases Related to Library *com.unity3d.player*.

Modification	App (#.)
com.unity3d.player/com.gamegod	12
com.unity3d.player/com.google.ads	7
com.unity3d.player/com.basyatw.bcpawsen	5
com.unity3d.player/org.fmod	4
com.unity3d.player/com.pbera.cuo	4
com.unity3d.player/com.tpzfw.yopwsn	3
com.unity3d.player/com.geseng	3
com.unity3d.player/org.apache.http	2
com.unity3d.player/com.ranway	2
com.unity3d.player/com.pmpm.pm	2
com.unity3d.player/com.nknk.nk	2
com.unity3d.player/com.naiiwt.toolon	2
com.unity3d.plugin/com.muzhiwan.embed	1
com.unity3d.player/ctl4ever.pu.com	1
com.unity3d.player/com.wohse.zuwreo	1
com.unity3d.player/com.usoety.toein	1
com.unity3d.player/com.tooswon.usoan	1
com.unity3d.player/com.tn.dq	1
com.unity3d.player/com.nnduBWhN.p	1
com.unity3d.player/com.niu	1
com.unity3d.player/com.mediocres.libary	1
com.unity3d.player/com.kuguo.pushads	1
com.unity3d.player/com.gamelosd	1
com.unity3d.player/com.elm	1
com.unity3d.player/com.db.pe	1
com.unity3d.player/com.db.cw	1
com.unity3d.player/com.bodys.sh	1
com.unity3d.player/com.asdpaw.foivnaw	1
com.unity3d.player/com.apkmania	1
Total	65

App developers may modify common libraries for achieving their specific purposes. For example, as demonstrated in Listing 2, the library code of *com.googlecode.android*, which provides a range of date picker widgets based on the principle of sliding bars[17] has somehow accessed into the main app code (i.e., com.cozi.androidfree.*). Actually, through a lightweight static analysis approach[18], we have found that app developers have modified the code of 57 libraries. It is worth to mention that those 57 candidate libraries could also possibly be false positive results, because normally library code is self-containing and hence should not call into main app code. However, if we simply flag those 57 candidate libraries as non-library code, we could also introduce false negative results. Nevertheless, to understand why developers want to modify library code and thereby to mitigate potential false positive and false negative results are out of the scope of this work. We thus keep it as our future work.

8.3. Threats to Validity

Because there is no convention for specifying that a code package represents a library, identifying Android common libraries is challenging. We were able to perform our study by mining about 1,5 million apps collected over several months. Our study however presents a few threats to validity:

Threats to internal validity. Currently, our approach is not fully aware of obfuscation, which may lead to incomplete results. However, our findings are based on a large datasets of apps to reduce the influence of obfuscated apps. Besides, our findings could be leveraged in settings where for example ad-libraries are represented by features which are resilient to obfuscation (e.g., called SDK API methods). More specifically, Wang et al. [48] have proposed a promising approach to identify possible obfuscators for Android. The identification of obfuscated libraries is however outside the scope of this work. Our common library set could be expanded by the research community with obfuscated versions.

The current implementation of LibExcluder aims to exclude libraries for static code analysis, where the remaining code might not be compilable (the drawback shared from the code instrumentation strategy borrowed from other approaches such as IccTA [3] and DroidRA [52]). There is no guarantee that the newly generated app version will be executable. Nevertheless, the goal of LibExcluder is not to generate executable Android app (or even compilable code), but to generate an app version that is capable of improving the performance of static analysis approaches.

Threats to external validity. The primary threats to external validity are related to our selection of Android apps. The experiments mainly rely on the AndroZoo repository, which may not be representative of all Android apps available in the mobile ecosystem. Nonetheless, this threat is mitigated by the fact that AndroZoo is the largest repository available for the research community.

[17] https://code.google.com/archive/p/android-dateslider/

[18] Given a candidate package p and an app app containing p, we first leverage Soot to build a call graph (cg) of app. Then, for any method m belonging to p, we recursively check its invoked methods. If m reaches a method belonging to the main app code, which can be recognized based on the unique app package name, we consider candidate p has been modified by app developers.

```
1 //The package name of this app is com.cozi.androidfree
2 public class com.googlecode.android.widgets.DateSlider.ScrollLayout extends
      android.widget.LinearLayout{
3  private void setTime(long, int, int, int, boolean){
4   $i6 = $r0.<com.googlecode.android.widgets.DateSlider.ScrollLayout: int minuteInterval>;
5   $r10 = virtualinvoke $r10.<java.lang.StringBuilder: java.lang.StringBuilder append(int)>($i6);
6   $r11 = virtualinvoke $r10.<java.lang.StringBuilder: java.lang.String toString()>();
7   staticinvoke <com.cozi.androidfree.util.LogUtils: void
      v(java.lang.String,java.lang.String)>("ScrollLayout", $r11);
8 }}
```

Listing 2: Example of Library Code Calls into App Code.

It is known that app information related to release date (usually taken from the last modified date of the Android app) is not reliable. For example, the release time can be manipulated by attackers. Nevertheless, our previous work (namely MoonlightBox, a tool to uncover release-time inconsistencies of Android apps [58]) has shown that such inconsistency only affects 7% of Android apps.

Threats to construction validity. So far, LibRepresenter represents common libraries based on their structural information. For highlighting obfuscated library versions, our approach only works for such cases where the structural information of libraries is not changed. Fortunately, most obfuscated Android apps are transformed by Proguard[19], the official obfuscation tool, which is designed for renaming only. In other words, the structure information would not be changed and thereby our approach should work for majority obfuscated Android apps. As discussed in Section 6.5, we could not run LibRepresenter on all the apps available in AndroZoo so that we have excluded obfuscated apps during our library identification process. Nevertheless, our results should not be significantly impacted as on one hand, the number of obfuscated apps is quite small (around 10% of candidates are excluded only), while on the other hand, a given library is likely to be presented in our big dataset in its obfuscated and non-obfuscated form. Even if the obfuscated version is overlooked, the non-obfuscated form should still be identified.

Although we have conducted several steps, including manual verification, to refine the identification of common libraries, the final results are not fully validated. The manual verification process is also subject to errors, which usually happen in manual inspection. In the future work, we plan to work out a robust framework so as to exhaustively evaluate our findings.

In this work, we have set a threshold at 10 for selecting common libraries. The bigger the threshold is, the smaller the size of potential libraries. Through experiments on randomly selected 20,000 apps, we observe that the decreasing of the number of potential libraries is much faster than that of actual libraries (by comparing the results with the library set obtained in this work), while increasing the threshold. By increasing the threshold from 10 to 20, the number of potential libraries drop 42%, while the number of actual libraries only drop 24%. Nonetheless, although our experiments reveal promising results, it may introduce false negative results, as conceptually a "package" shared by two different Android apps could be a library.

Moreover, in this work, we have also set a threshold at 80% for flagging repackaged Android apps, which may not be adequate as well, resulting in false positive results. Therefore, more investigations are certainly needed to pinpoint a foolproof threshold regarding real-world Android apps. Ideally, more experiments in altering the threshold could be conducted to investigate the stability (or sensibility) of our approach. We plan to investigate this direction in our future work.

Since Android evolves very fast, the set of libraries, including both common libraries as well as ad libraries, may become outdated in the future. Nonetheless, our methodology and the scripts, prototype tools, should be reusable to re-characterise such lists of libraries. Hence, as our first attempt, we present to the community a simple online service[20] aiming to maintain the state-of-the-art library set. So far, because of resource limitation (with limited memory size), only a small number of apps are integrated. As our future work, we commit to expand this service to all the AndroZoo apps and continuously increase the set along with the increase of AndroZoo apps.

9. Conclusion

We have presented our process for collecting a set of 1,113 common libraries and 240 ad libraries from a dataset of about 1.5 million Android apps. We empirically illustrate how these two library sets can be used as *whitelist*s by Android analysis approaches to improve their performances. More specifically, we have shown that two approaches, namely repackaging detection and machine learning-based malware detection, can indeed benefit from our harvested libraries. Moreover, with the help of LibRepresenter, we have additionally collected the abstract representations of those libraries and empirically demonstrated that our collected library representations could be leveraged as a benchmark to flag the appearance of common libraries including obfuscated ones, which normally cannot be identified by whitelist-based matches.

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

[20] http://115.146.85.168/Libraries/

Acknowledgments

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments that have led to important improvements in several parts of the manuscript. This work was supported by the Monash-Warwick Alliance Catalyst Fund (2018/2019), by the European Union, under the SPARTA project, by the Fonds National de la Recherche (FNR), Luxembourg, under projects CHARACTERIZE C17/IS/11693861 and Recommend C15/IS/10449467, by the University of Luxembourg, under the VulFix project, by the National Key Research and Development Program of China (grant No.2017YFB0801903), and by the National Natural Science Foundation of China (grants No.61702045, No.61772042 and No.61873069).

References

[1] The Verge. 99.6 percent of new smartphones run android or ios 101. `https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016`, 2017. Accessed: 2017-08-10.

[2] Symantec. istr 20 - internet security threat report, Apr. 2015. http://know.symantec.com/LP=1123.

[3] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[4] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.

[5] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[6] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun 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 (ISSTA)*, pages 71–82. ACM, 2015.

[7] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *ICSE*, 2015.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[9] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[10] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[11] Theodore Book and Dan S Wallach. A case of collusion: A study of the interface between ad libraries and their apps. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 79–86. ACM, 2013.

[12] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013.

[13] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[14] Wenhui Hu, Damien Octeau, Patrick Drew McDaniel, and Peng Liu. Duet: library integrity verification for android applications. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 141–152. ACM, 2014.

[15] Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting Malicious Code from Piggybacked Android Apps. In *Technique Report*, 2016.

[16] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[17] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *TrustCom*, 2014.

[18] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[19] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. Citeseer, 2012.

[20] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. 2017.

[21] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William GJ Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2015.

[22] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Theodore Berger, Steffen Dienst, and Ahmed E Hassan. Impact of ad libraries on ratings of android mobile apps. *Software, IEEE*, 31(6):86–92, 2014.

[23] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, 2012.

[24] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[25] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[26] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94. ACM, 2014.

[27] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, 2015.

[28] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[29] Julius Davies, Daniel M German, Michael W Godfrey, and Abram Hindle. Software bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[30] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 175–186. ACM, 2014.

[31] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, pages 37–54. Springer, 2012.

[32] Jonathan Crussell, Clint Gibler, and Hao Chen. Scalable semantics-based detection of similar android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[33] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys 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*, pages 242–251. ACM, 2014.

[34] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*, 2015.

[35] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[36] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Proceedings of the 39th International Conference on Software Engineering*, pages 335–346. IEEE Press, 2017.

[37] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. Libsift: Automated detection of third-party libraries in android applications. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 41–48. IEEE, 2016.

[38] Arun Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.

[39] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[40] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.

[41] Kevin Allix, Quentin Jérome, Tegawende F Bissyandé, Jacques Klein, Radu State, and Yves Le Traon. A forensic analysis of android malware–how is malware written and how it could be detected? In *COMPSAC*, 2014.

[42] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017.

[43] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2017)*, 2017.

[44] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[45] Anthony Desnos. Android: Static analysis using similarity distance. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5394–5403. IEEE, 2012.

[46] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[47] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology*, 32(6):1108–1124, 2017.

[48] Yan Wang and Atanas Rountev. Who changed you?: obfuscator identification for android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 154–164. IEEE Press, 2017.

[49] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[50] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 13–24. IEEE Press, 2017.

[51] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[52] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[53] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 2014.

[54] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[55] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[56] Nitesh V. Chawla et. al. Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[57] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. Automatically locating malicious packages in piggybacked android apps. In *The 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft 2017)*, 2017.

[58] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.