2016 23rd Asia-Pacific Software Engineering Conference

LibSift: Automated Detection of Third-Party Libraries in Android Applications

Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan and Lipo Wang
School of Electrical and Electronic Engineering
Block S2, Nanyang Technological University
Nanyang Avenue, Singapore 639798
{csoh004, ibktan, yauhen001, annamala002, elpwang}@ntu.edu.sg

***Abstract*—Android applications typically contain multiple third-party libraries and recent studies have shown that the presence of third-party libraries may introduce privacy risks and security threats. Furthermore, researchers have reported the importance of considering the third-party libraries for their program analysis tasks. A reason being that the presence of third-party libraries may dilute the features and affect the accuracy of their results. Existing literature typically employs a whitelist to exclude the third-party libraries from their analysis in order to achieve accurate results. However, these whitelists are generally incomplete and weak against the renaming obfuscation technique that is commonly employed in Android applications. In this paper, we propose LibSift, a tool to automatically detect third-party libraries in Android applications. LibSift detects third-party libraries based on package dependencies that are resilient to most common obfuscations. The evaluation results not only indicate that LibSift can detect third-party libraries accurately and effectively, but also show that LibSift can detect even the less popular libraries that are not detected by two of the state-of-the-art approaches.**

I. Introduction

To date, Android is the most widely used smartphone platform. Due to its popularity, the number of apps developed for Android platform is also booming at an rapid rate. At the time of writing this paper, Android's official app market, Google Play, hosts up to more than 2 millions apps [1]. Unfortunately, along with its popularity, Android platform is being increasingly targeted by malware authors, who create malicious apps. Sophos [2], reported that 2,000 new malware samples were being discovered everyday in 2014, up from 1,000 in 2013. The security threats caused by the malicious apps may leak the user's private data or even cost the user money [3], [4]. Apart from security threats, Android apps also suffer from repackaging threats where plagiarists repackage legitimate apps with additional malicious code or redirected revenues and publish them as their own [5].

Therefore, in recent years, a substantial amount of research is being dedicated to analyzing Android apps in attempt to mitigate these threats. The analysis of Android app poses a number of challenges. For example, due to the openness of Android platform, there are multiple Android markets that any third-party developer can publish their apps on any of these markets without much scrutiny. This has led to an overwhelming number of apps to be analyzed. Furthermore, the flexibility of Android OS allows the developers to easily incorporate third-party libraries (TPLs) into their apps to lighten the burden on the developers. As a result, one particular feature of Android apps is that they generally contain a number of TPLs. The usage of TPLs escalates the problem further, as the use of TPLs typically causes the app to include a significant amount of code that is irrelevant and may hinder the process of many program analysis tasks. Wang et al. [6] reported that TPLs generally contribute to more than 60% of the app's code.

There are generally three more prominent areas of program analysis tasks that are affected by TPLs. Firstly, in app clone detection the similarities of the code between the apps need to be measured. Since TPLs can be easily manipulated, not considering all the TPLs for the analysis may greatly affect the results. A recent study [7] has shown that the computation of code clone similarities in Android app is significantly affected by the consideration of including or excluding TPLs. Secondly, static taint analysis is a time consuming and computational intensive process. However, a significant portion of the resources are spent on analyzing irrelevant code when the TPLs dominates the app. Researchers have reported that they have encountered these problems. For instance, using a state-of-the-art static taint analysis tool, FlowDroid [8], DroidSafe [9] reported that for some apps the analysis cannot be finished within 2 hours and AsDroid [10] reported that FlowDroid ran out of memory on some apps. Since TPLs are typically used as it is without any modification, the TPLs can be separately analyzed and have the results be reused in the analysis of the apps that use them [11]. Lastly, in malware detection, the presence of TPLs may dilute the features used in the detection, thus reducing the contrast between benign and malicious samples and affecting the results of the detection. For example, advertisement libraries are excluded from analysis in MUDFLOW [12] as advertisement libraries are frequently used in Android apps and their dataflows become common and dilute the training data. Furthermore, DroidAPIMiner [13] reported that filtering out TPLs increases the difference of API usage between malware and benign apps.

Apart from hindering many program analysis tasks, the usage of TPLs is also associated with more privacy risks and security threats [14], [15]. For example, popular TPLs may be masqueraded by malicious libraries to mislead users into thinking that it is a legitimate TPL. Furthermore, some TPLs, even the popular ones, are known to be aggressively

1530-1362/16 $31.00 © 2016 IEEE
DOI 10.1109/APSEC.2016.40

collecting the users' private data. By identifying the TPLs in the apps, we can effectively address these library-centric threats. Grace et al. [16] reported that most of the existing advertisement libraries collect private data and some run code fetched from the Internet. In another recent study, the authors have demonstrated that TPLs are likely a significant medium for the propagation of malicious code [17].

Consequently, multiple attempts had been made in the existing literatures to identify and exclude TPLs from different program analysis tasks [5], [13], [18], [19]. One of the most common techniques used to identify TPLs in Android apps is to use a whitelist and match the names of the packages in the app to the TPLs package name listed in the whitelist. However, due to the widespread interest and the dynamic nature of Android ecosystem, it is difficult to build a comprehensive whitelist of TPLs as there are too many of them. Furthermore, it is common for Android apps to employ obfuscation techniques that may, for instance, rename the packages, classes and methods. For example, a non-obfuscated app with a package *com.library.example*, after obfuscation, the package may become *com.a.a*. Thus, this common obfuscation technique will cause the whitelist approach to fail in detecting the obfuscated TPL.

Despite the importance of identifying TPLs in Android apps, there are only a few existing studies focusing on it. LibRadar [20] uses as feature the frequency of different Android APIs in each pacakge, and performs feature hashing on a large number of apps. Following that, strict comparison is enforced to perform clustering and identify TPLs that are frequently used. Li et al. [21] put together a large whitelist of popular TPLs by refining a list of package names that have frequent appearance in a large number of apps.

In this paper, we propose LibSift, an approach to identify TPLs in Android apps based on the package dependency graph (PDG) of the app. We evaluate the effectiveness and efficiency of LibSift on a real-world dataset of 300 Android apps. We further compare LibSift with two state-of-the-art TPLs detection approaches, LibRadar [20] and the whitelist from Li et al. [21].

Our proposed approach is inspired by PiggyApp [22] that is the most related study to our work, but it addressed a different problem. In PiggyApp, its goal was to identify piggybacked apps[1] that share the same primary module. In our approach, we focus on detecting TPLs in Android apps. Its main idea is that typically, the code of every Android app can be divided into primary module and non-primary modules (if any) by using module decoupling technique. The primary module defines the app's core functionalities and the non-primary modules are contributed by TPLs. The code within each module, primary or non-primary, is tightly coupled, whereas, the code between modules is loosely coupled or even standalone. Furthermore, dependency graph techniques have been proven to be resilient to multiple types of common obfuscation techniques, such as renaming, statement manipulation and program transformation [18]. Another advantage of our approach is that, on the contrary to the state-of-the-art TPLs detection approaches, our approach does not assume that all TPLs will be used by many apps. Thus, we are able to detect even the non-popular TPLs that do not appear in many apps. In fact, in our evaluation, we show that LibSift can effectively detect the less popular TPLs missed by both of the state-of-the-art approaches. In addition, for our approach, it is straightforward to update the list of TPLs when new library versions or new TPLs are detected.

Our main contributions in this paper are as follows:

- We propose a novel approach to automatically detect all TPLs used in Android apps based on package dependency graph and without first having to study a large number of apps.
- We implement a prototype of our approach, LibSift, and conduct extensive experiments to evaluate the effectiveness and efficiency of our approach to detect TPLs in Android apps.
- We compare our approach with two state-of-the-art approaches, LibRadar [20] and whitelist from Li et al. [21], and show that our approach can detect libraries that are not detected by them.

II. Proposed Third-party Library Detection

A. Overview

In the software context, a library is generally coded in a modular fashion, such that the code within the library is loosely coupled but highly cohesive. Coupling refers to the interdependencies between modules, while cohesion describes the degree of relationship between the functions that are within a single module. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large. TPLs written for Android apps follow the same principle. Android apps are generally written in Java programming language and thus possess Java's property of organizing code into packages. As a result, Android apps can be decoupled into individual modules, where the code within each module is tightly interwoven and the code between each module is independent or loosely coupled. Based on the above-mentioned observation, we propose to detect all TPLs used in a given Android app by making use of the natural partitioning of the Android apps and perform module decoupling at the package level.

In our approach, we use a module decoupling technique similar to the technique that was introduced in PiggyApp [22]. PiggyApp performs module decoupling to identify the primary module of the apps and uses it to detect Piggybacked apps. Despite addressing different issues, since our technique is similar in nature, it is notable to highlight the differences between our techniques. Firstly, package homogeneity[2] is considered as a dependency relationship with high importance in PiggyApp. Whereas in LibSift, package homogeneity is not consider as a dependency, but as a criteria to merge the

[1]Piggybacked apps refers to a type of repackaged app which involves the injection of rider code into the original app.

[2]We consider two packages as homogeneous if they form a parent-child or sibling relationship

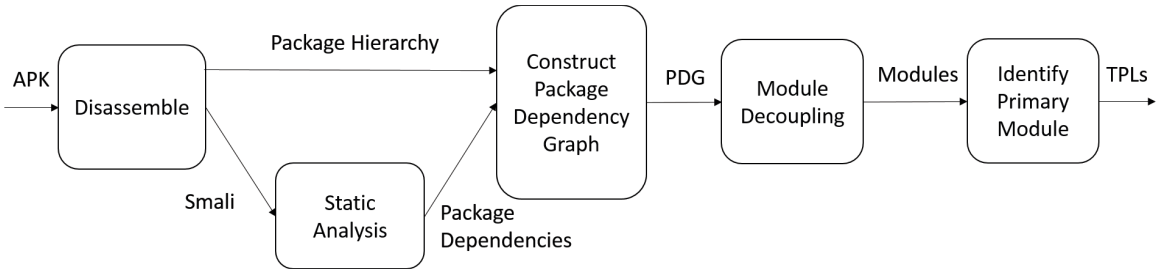

Fig. 1. Overview of LibSift

packages into a module. Secondly, in the module decoupling process, PiggyApp performs agglomerative clustering to cluster the packages with dependencies weight greater than a pre-defined threshold, starting from the most tightly coupled pair and stop when there are no clusters with dependencies weight greater than threshold left. However, in the case of LibSift, we perform module decoupling using a different algorithm. For instance, we check for package homogeneity before merging the packages and if more than one sibling packages belong to a module, we assign all sibling packages sharing the same parent to that module. Finally, the method we use to identify the primary module is also different. In PiggyApp, the primary module is identified as the module that provides the main activity or the module that handles the most activities of the app. However, LibSift identifies the primary module based on the app's package name or as the module that have dependencies with most number of other modules. More details on our approach are provided in the following (Section II-B - II-E).

Figure 1 shows the overall architecture of our approach to detect TPLs in Android apps. LibSift consists of four main processes, disassembling, constructing PDG, module decoupling, and identifying primary module. The disassembly process is necessary to reveal the information required for the next process. The construction of the PDG involves the analysis of the bytecode information to extract dependencies between different packages. For module decoupling, we cluster the packages of the given app into separate modules based on its PDG. Lastly, if the app contains more than one module, we identify the primary module from the set of modules.

B. Disassemble

The Android operating system uses Android application package (APK) for distribution and installing. When the app developer compiles and builds the app, all the app code are compiled into a single dex file, thus, losing the clear separation between the TPLs and the core app code. Following that, the dex file along with the necessary resources are packaged into an APK. Fortunately, during the building process the hierarchical information of the packages within the app is preserved and we can use it along with package dependency information to detect the TPLs used by the app.

In order to do so, given an APK, we first disassemble it using apktool [23] to reverse engineer the classes.dex file to smali files, which also reveals the package hierarchy information within the app. During the disassembly process, the bytecode in the dex file is converted into smali [24] intermediate representation in the form of multiple smali files which is analogous to the class files of the Java programming language. Furthermore, these smali files will be placed in their respective folder based on their package hierarchy. As such, we can now analyze the smali files and extract the package dependencies information.

C. Package Dependency Graph

The PDG shows the degree of dependencies between the packages of an app. Note that our computation of PDG does not consider control dependencies, but instead focuses on dependency relationships, such as class inheritance, method calls, and member field references. With the dependencies and package hierarchy information, we construct a $PDG = (N, E, W)$. Where N is a set of nodes and each node $n \in N$ represents a package in the app (note that we only consider packages with smali files directly under them). $E \subseteq (N \times N)$ is a set of edges and each edge $e(n1, n2) \in E$, where $n1, n2 \in N$, connecting any two nodes represents that there is a dependency between these two nodes. Lastly, W is a set of labels representing the weights of the edges and each weight $w \in W$ is the degree to which the nodes connected by the edge $e(n1, n2)$ are dependent on each other. As different relationships represent a different degree of dependencies, we assign them with different weights. Except for package homogeneity, we use the weight assignments suggested in PiggyApp, which are 10, 2, 1 representing class inheritance, method calls, and member field references, respectively. In the following section (Section II-D) we will explain that we view package homogeneity not as a dependency, but as an additional criteria for the packages to be clustered as a module.

An example of the PDG for a Korean language learning app is as shown in Figure 2. In this particular app, there are 15 sub-packages that contain smali file directly under them. Some of the sub-packages that do not have dependencies with any other sub-package are depicted as nodes without edge.

D. Module Decoupling

In our approach, we perform module decoupling using package level semantic information. Each module consists of one or more packages, and every package in the module has

a parent to child or sibling relationship to at least one other package in the same module. However, we argue that sharing the same parent in the package hierarchy does not necessarily means that they are related. For example, the following two popular libraries, *com.google.ads* and *com.google.analytics*, they share the same parent *com.google* and may be both considered as libraries from *Google*, but in actual fact, they are two separate libraries that can be used independently. Hence, it is more beneficial to report them as separated libraries.

As such, based on the PDG, we cluster the packages into modules only if the following conditions are met: 1) Package homogeneity 2) Total weight between the two packages is greater than the pre-defined threshold and 3) If package p1 and package p2 are sibling nodes from the same module then all the other sibling nodes of p1 and p2 are from the same module. As mentioned above, simply sharing the same parent module does not means that they are related. However, if there are high dependency between some sibling nodes, then it is likely that all other sibling nodes belongs to the same module. The algorithm we use to perform module decoupling is shown in Algorithm 1.

Algorithm 1: Module Decoupling Algorithm

Input: PDG - Program Dependency graph of the app
threshold - Pre-defined threshold
Output: M - Set of Modules of the app

```
foreach edge e in PDG do
    if weight w > threshold then
        (n1, n2) ← get_nodes(edge)
        if package_homogenity(n1, n2) == TRUE then
            M ← merge(n1, n2)
foreach node in the PDG do
    if sibling_nodes in a module then
        M ← merge(node, M)
return M
```

The PDG of the same app in Figure 2 after the module decoupling process is shown in Figure 3. As evident in Figure 3, the 15 sub-packages of the app are successfully merged into 5 modules. The sibling nodes such as *android.support.v4.view* and *android.support.v4.widget* are merged into the parent node *android.support.v4*. Furthermore, since *android.support.v4* is already declared as a module, all child nodes under *android.support.v4* in the package hierarchy are merged into this module. As the packages can only be merged when package homogeneity is true, the TPLs *com.facebook.android* and *android.support.v4* are not merged with the primary module *com.bravolang.korean*.

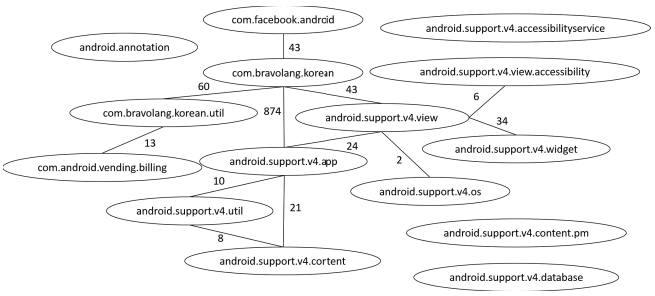

Fig. 2. Package Dependency Graph Example

E. Identify Primary Module

The primary module is named as such because it provides the primary function of the app and it is where the core app code resides. For certain Android app security analysis, especially Android app clone detections, it is important to identify the primary module for similarity analysis as the other modules are TPLs that can be easily replaced with other TPLs of similar functions.

Based on our observation, in most cases, the primary module can be simply inferred from the app's package name or is a subset of the app's package name. Note that in this paper, app's package name refers to the unique package name declared in the AndroidManifest.xml that uniquely identifies the app on the device and package name refers to the name of the packages in the app. Using the same Korean language learning app as an example, its app's package name is *com.bravolang.korean*, and the core app code written by the developer is all enclosed within the *com.bravolang.korean* sub-package, therefore the primary module can be correctly identified as *com.bravolang.korean*. This common practice is due to the fact that Android app development following the Java programming language package naming convention, that suggest that developers use their reversed Internet domain name to begin their package name to avoid conflict with other apps. Moreover, using each period in the package name as a path separator, all the code by the same developer would be placed together in the path hierarchy.

However, in some cases, we are unable to infer the primary module just from the app's package name. One of the main reason is due to the use of obfuscation which renamed the packages. In these cases, we identify the primary module base on the reasoning that the primary module contains the core code of the apps that is in charge of communicating with the TPLs to access their resources. Furthermore, TPLs are designed to work independently and are not dependent on other TPLs to function, this means that non-primary module should not have dependencies on other non-primary modules. Therefore, the primary module can be identified based on

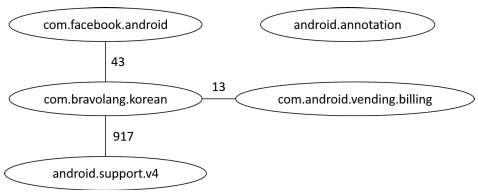

Fig. 3. Package Dependency Graph After Module Decoupling

the highest number of other modules it has dependencies on. Based on the PDG, for each module, we can identify its dependencies on other modules and compute the total number of other modules it has dependencies on.

In summary, to identify the primary module, we first use the app's package name and match it with the modules that we have identified. If a match is found, the matching module will be the primary module. However, when we fail to identify the primary module through matching the package name, we will identify the module that communicates with the most number of other modules as the primary module. The algorithm we use to identify the primary module is shown in Algorithm 2.

Algorithm 2: Identify Primary Module

Input: M - Set of Modules of the app
app_package_name - unique package name of the app
Output: primary_module

```
foreach module m in M do
    if match_pacakgeName(module, app_package_name) == TRUE then
        return module
return highest_dependent_count(M)
```

III. Evaluation

We have implemented a prototype of LibSift in Python code to perform preparations, module decoupling and identification of primary module for the detection of TPLs in Android apps. In the preparation step, we first disassemble the APK of the given app, using apktool [23]. Our python script then automatically extract the packages from the app. Following that, we parse the smali code to identify the dependencies between the packages and build a weighted PDG. Finally, we cluster the packages into modules based on Algorithm 1 (refer to Section II-D) and identify a primary module using Algorithm 2 (refer to Section II-E). For the evaluation of LibSift, we first evaluate the accuracy of LibSift on a set of real-world Android apps. Next, we evaluate the performance of LibSift. Finally, we compare LibSift with two state-of-the-art TPLs detection approaches. The dataset used in our experiment consists of 300 real-world apps collected from the top popular apps from different categories in Google Play store. All evaluation experiments are performed on a Linux machine with 2.6 GHz Intel core CPU and 32 GB of RAM.

A. Module Decoupling and Validation

Based on our experiments, we empirically determined that a cut-off threshold = 15 is a suitable value to accurately decouple the modules in Android apps. A threshold that is too high will tend to fail to identify and group related packages belonging to the same module that do not have high dependencies. On the other hand, a threshold that is too low is prone to falsely group unrelated packages with low dependencies into the same module.

TABLE I
Module Decoupling Results Summary

Description	Number of modules
Total number of modules	5,460
Number of obfuscated modules	802
Minimum number of modules in an app	1
Maximum number of modules in an app	76
Average number of modules in an app	18.2
Standard deviation of modules in an app	13.77

A summary of our module decoupling results is presented in Table 1. The results stated are from the 300 apps dataset and based on threshold = 15. The total number of modules detected by LibSift in all 300 apps is 5,460. The breakdown of the numbers of obfuscated and non-obfuscated modules detected by LibSift are 802 and 4,658 respectively. The minimum, maximum and average number of modules per app, detected by LibSift, are 1, 76 and 18.2 respectively. The standard deviation for the number of modules per app is 13.77. This shows that most Android apps do indeed use multiple TPLs.

To evaluate the module decoupling accuracy of LibSift, we manually analyze the 300 apps and verify that the modules are correctly decoupled. The results show that our approach correctly decoupled 296 apps. Therefore, the accuracy of LibSift is 98.67%. In the rare occasions where the apps are incorrectly decoupled, we found out that it is due to the packages in the primary module not having any dependencies on each other. This causes the primary module to be separated into multiple modules and thus, falsely identified as TPLs, when they should be part of the primary module.

B. Primary Module

To evaluate the accuracy of LibSift in identifying the primary module, we manually verify the primary module of the 296 apps that are decoupled correctly. Note that if there are more than one potential primary module, LibSift reports all of the potential primary modules. In this case, since LibSift is unable to pin point the primary module, even though the correct primary module is among the list of potential primary modules, we consider those apps with multiple potential primary modules reported as wrong primary module identification. A summary of the the primary module identification statistics is presented in Table 2. The results show that 283 apps have their primary modules identified correctly. Therefore, the accuracy of LibSift's primary module identification is 95.61%. The reason LibSift fails to identify the primary module is two-fold. Firstly, the name of the packages for the primary module is obfuscated or different from the package name and therefore, cannot be identified by simply matching them. Secondly, in some rare cases, the TPL represents the core of the app and is used to provide communications between different TPLs.

Out of the 296 apps that are decoupled correctly, the primary module of 23 apps cannot be identified from its package name. Therefore, LibSift attempts to identify the potential primary module by looking for the module that has dependencies

TABLE II
PRIMARY MODULE IDENTIFICATION SUMMARY

Description	Number of apps
Identified from app's package name	273
Unable to to identify from app's package name	23
Based on dependencies with number of other modules	
Only 1 module with highest count	16
Primary module identified correctly	10
More than 1 module with highest count	7
One of the module with highest count is primary	3

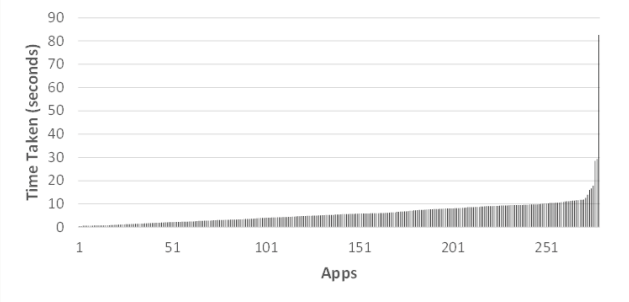

Fig. 4. Time Taken by LibSift to Process Each App

with the most number of other modules. For 16 apps out of the 23 apps, there is only 1 module in each app that has dependencies with most other modules. Among these 16 apps, 10 of them have their primary module identified correctly. Whereas for the other 7 apps out of the 23 apps, more than one potential primary modules are identified in each app as there are multiple candidates with dependencies on an equal number of modules. Out of these 7 apps, 3 of them contain the true primary modules.

C. Performance

After dissembling the APKs, the total time taken for LibSift to detect TPLs in all 300 apps is less than 27 minutes. Figure 4 shows the breakdown of time taken to detect TPLs in each app, sorted in ascending order. Only 1 app took more than 1 minute to process and about 90% of the apps took around 10 seconds or less. Note that these are achieved without prior studies of a large number of apps. To further improve the efficiency of LibSift, since LibSift does not require to cluster a large number of apps, it can be easily parallelized by distributing the workload to multiple machines.

D. LibSift vs. LibRadar and Whitelist

In this section, we compare our approach with two state-of-the-art approaches. One of them is the LibRadar [20] TPLs detection tool which is based on API features of packages in the apps. It uses a clustering based approach to cluster the hashes of the packages in a large dataset of apps and the packages that are clustered into large clusters, greater than a pre-defined threshold, are identified as TPLs. Another approach is by far the largest set of TPLs whitelist collected by Li et al. [21]. The list of TPLs is harvested from a large dataset of Android apps by extracting the names of the packages and clustering them based on the frequency of occurrence and followed by a series of refinements.

Figure 5 presents the results of TPLs detection for LibSift, LibRadar, and whitelist from Li et al. [21] on the same set of 300 real-world Android apps. The Y-axis represents the total number of libraries detected by each approach for the corresponding apps along the X-axis. As shown in the Figure 5, LibSift is capable of detecting more TPLs in most cases. For the 300 apps, the average numbers of TPLs detected by LibSift, LibRadar and whitelist from Li et al. are 17.17, 7.68 and 7.93 respectively.

In general, the TPLs not detected by LibRadar are the less popular ones. In addition, we also observed that in some apps the popular TPLs are detected by LibRadar, but the same TPLs are not detected in some other apps. Upon further investigation, we found out that it is due to different versions of the TPLs using slightly different APIs. As mentioned in their paper, LibRadar [20] enforces strict comparison, such that, two packages can only be cluster together when they share the exact same features (APIs). This results in the detection of multiple versions of the same TPL. However, in the case where the particular version of the TPL is not used frequent enough, possibly due to reason such as short update intervals, despite being a popular TPLs, this version of the TPL will not be detected. On the other hand, the TPLs not detected by using the whitelist from Li et al. are generally the less popular TPLs and those that have their package names obfuscated. Furthermore, Li et al. [21] stated in their paper that they do not list libraries starting with the name *"android.support"*. It is worth noting that when using the whitelist, unlike semantic based approach, all versions of the popular TPLs will always be detected as long as the package name remains the same and non-obfuscated.

With the APKs disassembled, the total time taken for each approach, LibSift, LibRadar and whitelist to detect TPLs from the 300 apps are 1,617, 1,875, and 76 seconds, respectively. As expected, the usage of whitelist is significantly faster as compared to other approaches. However, as mentioned above, the whitelist approach is vulnerable to package renaming obfuscation technique, which is commonly used in the Android apps. Furthermore, despite their attempt to overcome the incompleteness of whitelist by studying a large number of apps, many of the less popular libraries are still not listed. Since the study is based on a large number of apps, it may be expensive to keep the whitelist up to date. Similarly, LibRadar requires a time-consuming pre-processing step of extracting unique features from a large number of apps and clustering them, which can be expensive to update their database. This can be a problem because the Android ecosystem is fast paced and dynamic, new libraries will be emerging at a fast pace. For LibSift, the total time taken to TPLs in all 300 apps is slightly shorter than LibRadar (after the database is established). Currently, LibRadar is able to provide the users with more details, such as meaningful package name for those obfuscated TPLs. However, there are no obvious challenges that prevent LibSift from including these features in the future

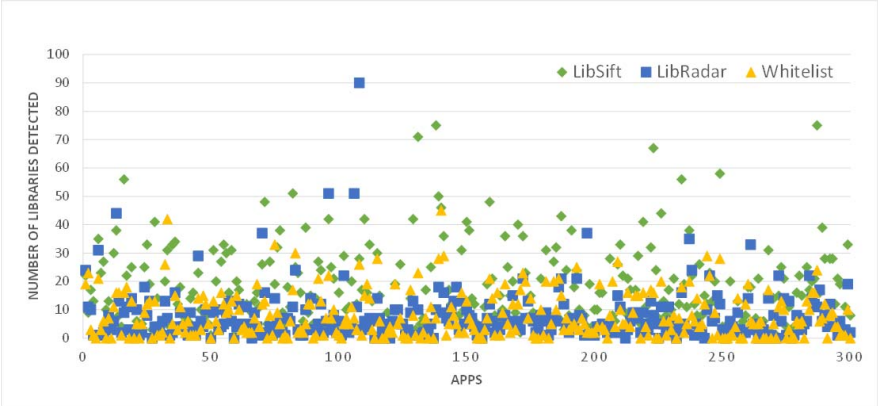

Fig. 5. Number of TPLs Detected by LibSift, LibRadar [20] and Whitelist [21] for 300 apps

and it is part of our plans for future work.

IV. Threats to Validity

As there are currently no conventions to determine whether a part of a software program is actually a TPL, the validity of our assumption that each module is a TPL could be threatened. This can happen when a library contains separated modules that are independent of each other. However, we believe that the negative consequences of missing a library are much greater than splitting up a library. Note that this is also an issue for both the state-of-the-art approaches. Furthermore, it is possible that our reported module contains more than one library that have high dependencies on each other. However, TPLs are designed to work individually and this situation should rarely occur. In addition, our approach checks for package homogeneity before merging the packages into module, thus reducing the possibility this error.

Our analysis is limited to free Android apps and could threaten the validity of the generalization of our findings. For instance, it is very likely that commercial apps have more obfuscated modules to protect their interests. Furthermore, the size of our dataset is a very small amount compared to the millions of apps available across the Android markets. However, to mitigate the threat, we use real-world dataset which covers the top popular apps from different categories.

V. Related Work

In this section, we discuss the related work in the existing literature and we divide this section into three parts. We first discuss the related work that involves the detection of TPLs in Android apps to achieve their goals. Secondly, we discuss the work that employs module decoupling technique. Lastly, we discuss the related work that aims to identify TPLs in Android apps.

A number of Android security work requires them to detect TPLs in Android apps. Most of the existing studies do so using a whitelist. For example, Chen et al. [19] use a whitelist to remove apps that use same framework or common libraries from their app clone detection. Aafer et al. [13] use a whitelist to remove any APIs exclusively invoked by TPLs and improve the accuracy of their Android malware detection. DroidMOSS [5] reduces false positives in their app clone detection by removing features from advertisement libraries using a whitelist. Instead of whitelist, Wukong [6] uses the frequency of different Android APIs calls in each sub-package as a feature and performs strict comparison to cluster identical package for identifying and filtering TPLs, before detecting Android app clones. DNADroid [18] identifies TPLs by comparing the SHA-1 hashes of known libraries and excluding them from their Android app clone detection. Andarwin [25] uses program dependency graph and clustering techniques to eliminate TPLs and detect semantically similar Android apps. We believe that the results of these studies can be further improved by using our approach to identify TPLs.

The following studies employ module decoupling technique on Android apps to address different challenges and prove that module decoupling technique works well on Android apps. PiggyApp [22] aims to detect piggybacked apps (legitimate apps with malicious code attached), by first separating the app code into primary and non-primary modules and comparing apps with similar primary modules to identify repacked app with rider code. Addetect [26] decouples Android app into individual modules, they then extract features from these modules and use machine learning technique to identify advertisement libraries. Droidlegacy [27] partitions Android apps into loosely coupled modules and comparing the signature of each module to known malware families to identify piggybacked malicious apps. These studies perform module decoupling at different granularity levels. In the module decoupling performed by PiggyApp, each node in the graph represents a Java package that includes all the Java class files declared within it. However, AdDetect only considers the packages that represent the root of the package subtrees. Whereas in DroidLegacy, each node in the graph presents a Java class. In LibSift we perform module decoupling at the same granularity level as PiggyApp.

The following studies focus on detecting TPLs in Android apps. Li et al. [21] identify TPLs in Android apps from a

large dataset of 1.5 million apps. Their approach is based on the appearance frequency of the package name in the large dataset and subsequent refinements to refine the list. LibRadar [20] extends Wukong's [6] clustering-based technique by performing feature hashing on the features to effectively detect TPLs used in Android apps. Both techniques are based on the assumption that TPLs are used in a large number of apps. Therefore, unlike LibSift they may not be able to identify less popular or new TPLs.

VI. Conclusion and Future Work

In this paper, we have presented our approach to detect third-party libraries in Android apps. Our approach is based on the observation that libraries in software program are highly cohesive but loosely coupled. Therefore, for a given Android app, we perform module decoupling based on its PDG and identify its primary and non-primary modules. We have implemented a prototype of LibSift and evaluated it on a set of real-world Android apps. Our result shows that LibSift is able to effectively identify TPLs in Android apps. We have also compared our approach with two other state-of-the-art approaches, LibRadar [20] and whitelist from Li et al. [21]. The results show that our approach can detect libraries not detected by them.

Despite the good results, LibSift can still be improved in multiple ways. For our future work, we plan to extend our current work by improving on the current algorithm and providing additional useful features. For example, it may be useful to recognize and identify the obfuscated TPLs. Furthermore, additional information such as the type of library and the maliciousness of the library are also important for certain program analysis tasks. All these can be achieved by performing static analysis on the code of each of the modules identified by LibSift.

References

[1] Appbrain: Google play stats. [Online]. Available: http://www.appbrain.com/stats

[2] Sophos: Sophos mobile security threat report. [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[4] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[5] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.

[6] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 71–82.

[7] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 242–251.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[9] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[10] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1036–1046.

[11] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[12] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 426–436.

[13] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 86–103.

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

[15] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu, "Duet: library integrity verification for android applications," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 141–152.

[16] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[17] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following devils footprints: Cross-platform analysis of potentially harmful libraries on android and ios," in *IEEE Symposium on Security and Privacy*. in press, 2016.

[18] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[19] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.

[20] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 653–656.

[21] L. Li, J. Klein, Y. Le Traon *et al.*, "An investigation into the use of common libraries in android apps," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 403–414.

[22] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 185–196.

[23] Apktool: A tool for reverse engineering android apk files. [Online]. Available: http://ibotpeaches.github.io/Apktool/

[24] Smali: An assembler/disassembler for androids dex format. [Online]. Available: https://github.com/JesusFreke/smali

[25] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[26] A. Narayanan, L. Chen, and C. K. 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*. IEEE, 2014, pp. 1–6.

[27] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 2014, p. 3.