Automated Third-Party Library Detection for Android Applications: Are We There Yet?

Xian Zhan*
The Hong Kong Polytechnic University, Hong Kong, China
chichoxian@gmail.com

Lingling Fan*
College of Cyber Science, Nankai Univerisity, China
Nanyang Technological University, Singapore
ecnujanefan@gmail.com

Tianming Liu
Monash University
Australia
tianming.liu@monash.edu

Sen Chen
College of Intelligence and Computing, Tianjin University, China
Nanyang Technological University, Singapore
ecnuchensen@gmail.com

Li Li
Monash University
Australia
li.li@monash.edu

Haoyu Wang
Beijing University of Posts and Telecommunications
China
haoyuwang@bupt.edu.cn

Yifei Xu
Southern University of Science and Technology, China
11611209@mail.sustech.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University, Hong Kong, China
luoxiapu@gmail.com

Yang Liu
Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

ABSTRACT

Third-party libraries (TPLs) have become a significant part of the Android ecosystem. Developers can employ various TPLs with different functionalities to facilitate their app development. Unfortunately, the popularity of TPLs also brings new challenges and even threats. TPLs may carry malicious or vulnerable code, which can infect popular apps to pose threats to mobile users. Besides, the code of third-party libraries could constitute noises in some downstream tasks (e.g., malware and repackaged app detection). Thus, researchers have developed various tools to identify TPLs. However, no existing work has studied these TPL detection tools in detail; different tools focus on different applications with performance differences, but little is known about them.

To better understand existing TPL detection tools and dissect TPL detection techniques, we conduct a comprehensive empirical study to fill the gap by evaluating and comparing all publicly available TPL detection tools based on four criteria: effectiveness, efficiency, code obfuscation-resilience capability, and ease of use. We reveal their advantages and disadvantages based on a systematic and thorough empirical study. Furthermore, we also conduct a user study to evaluate the usability of each tool. The results show that LibScout outperforms others regarding effectiveness, LibRadar takes less time than others and is also regarded as the most easy-to-use one, and LibPecker performs the best in defending against code obfuscation techniques. We further summarize the lessons learned from different perspectives, including users, tool implementation, and researchers. Besides, we enhance these open-sourced tools by fixing their limitations to improve their detection ability. We also build an extensible framework that integrates all existing available TPL detection tools, providing online service for the research community. We make publicly available the evaluation dataset and enhanced tools. We believe our work provides a clear picture of existing TPL detection techniques and also give a road-map for future directions.

*The corresponding authors.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416582

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Libraries and tools** → Program analysis.

KEYWORDS

Third-party library, Android, Library detection, Empirical study

ACM Reference Format:
Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, YifeiXu, Xiapu Luo, and Yang Liu. 2020. Automated Third-Party Library Detection for Android Applications: Are We There Yet?. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416582

1 INTRODUCTION

Nowadays, Android applications (apps) occupy an irreplaceable dominance in the app markets [23] and will continuously hit the new height [10]. Along with the thriving of Android apps is the emerging of countless third-party libraries (TPLs). When app developers implement their own apps, they usually realize some functionalities by integrating various TPLs, such as advertisements, social networking, analytics, etc. Prior research [52] has shown that about 57% of apps contain third-party ad libraries. Wang et al. [58] also revealed that more than 60% of the code in an Android app belongs to TPLs. TPLs can facilitate the development process and provide powerful functionalities for apps. However, every coin has two sides. This situation also brings new security threats. Some TPLs may contain malicious code. When they are integrated into popular apps, they can quickly infect a large number of mobile devices. Besides, TPLs as noises could affect the results of repackaging detection [62], malware detection [47], counterfeit apps detection [58], etc. Thus, research on TPL detection targeting the Android platform continues to emerge.

Generally, there are two ways to identify TPLs. The first one is the whitelist-based approach, and the second one directly extracts features from TPLs to identify them. In the beginning, most repackaging detection [44, 55, 63, 66] and malware detection [45] adopt whitelist to filter out TPLs because whitelist-based approach is simple and easy to implement. However, the whitelist-based method uses the package name to identify TPLs, which is not resilient to package renaming. A recent study showed that more than 50% of the inspected Android TPLs are protected by obfuscation techniques [51], which dramatically decreases the effectiveness of the whitelist-based method. Besides, the whitelists cannot cover all TPLs, especially newly-emerged ones.

In order to improve the detection performance, various research [33, 50, 53, 54, 56, 59, 65] tried to extract different features of TPLs and use different techniques to identify TPLs. However, the advantages and disadvantages, usage scenarios, performance, and capability of obfuscation-resilience of these tools are still not clear. Besides, no unified dataset is available to quantitatively evaluate them without bias. Undoubtedly, identifying these problems can also help us find the limitations and explore new methods in this direction.

Therefore, in this paper, we attempt to fill the gap by conducting a comprehensive and fair comparison of these state-of-the-art TPL detection tools on a unified dataset. We evaluate them by using four metrics: effectiveness, efficiency/scalability, code obfuscation-resilience capability, and ease of use. By investigating the four aspects of these tools, we attempt to achieve three goals in this study: (1) understand the capabilities and usage scenarios of existing TPL detection tools; (2) get a better understanding of the trade-offs in TPL detection and then conclude a better-optimized scheme to guide future work or help developers implement better tools; (3) integrate these publicly available tools as an online service, which provides the detection results of different TPL detection tools to users. In summary, our main contributions are as follows:

- We are the first to conduct a systematic and thorough comparison of existing TPL detection tools by using four metrics: effectiveness, efficiency, code obfuscation-resilience capability, and ease of use.
- We are the first one to construct a comprehensive benchmark including 59 unique TPLs (used by 221 Android apps) with 2,115 versions that can be used to verify the effectiveness of TPL detection tools (Section 5.2). We make this dataset available for community, and future researchers can also use this dataset to evaluate new tools.
- Based on our analysis, we point out the disadvantages of current research and present the potential challenges in this direction. We give suggestions on tool selection under different application scenarios and provide useful insights for future research on TPL detection.
- We build an extensible framework that integrates all existing TPL detection tools to provide an online service to users. We also improve some publicly available tools for better performance. All the related code and dataset and detailed evaluation results can be found on our website.[1]

The rest of this paper is organized as follows. Section 2 introduces the basic concept of the third-party library and detection process. Section 3 shows the related work. Section 4 presents a comprehensive comparison about these state-of-the-art TPL detection tools. Section 5 describes how we design our empirical study. Section 6 reports the evaluation and findings. Section 7 is the discussion. Section 8 concludes our work.

2 PRELIMINARY

2.1 Android Third-party Library

Third-party library (TPL) provides developers with various standalone functional modules, which can be integrated into host apps in order to speed up the development process. Since current TPL detection tools that we compared in this paper only consider Java libraries, we do not discuss the native libraries here. The Java libraries are usually published as ".jar" or ".aar" files. The ".aar" format file can only be used by Android apps, which usually provides UI-related libraries or game engine libraries. ".jar" files consist of class bytecode files, while ".aar" files include both class bytecode files and other Android-specific files such as manifest files and resource files. Most TPL files can be found/downloaded/imported from maven repository [20], Github [16], and Bitbucket [12]. In Android app development, if an app uses TPLs, the app code can be divided into two parts, the logic module from the host app (i.e., primary) and the supplementary module (i.e., non-primary) from TPLs [54, 56]. TPL detection aims to identify TPLs in non-primary modules.

2.2 TPL Detection Process

As shown in Figure 1, existing TPL detection techniques for Android apps usually unfold in four steps, which are elaborated below.
Step1: **Preprocessing.** Researchers usually first decompile apps by applying reverse-engineering tools such as apktool [9] and Androgurad [5], and then get the appropriate intermediate representation (IR) in this stage to facilitate the following steps.
Step 2: **Library Instance Construction.** The purpose of this step is to find the boundaries of TPL candidates and then separate them from the host apps. This step is optional because some tools can directly extract TPL features without splitting TPLs from host apps. Basically, there are two different strategies identifying the boundaries of TPL candidates: (1) consider all the independent Java packages as library candidates, collect TPLs beforehand as the ground truth and then compare library candidates with the ground truth; (2)

[1]https://sites.google.com/view/libdetect/

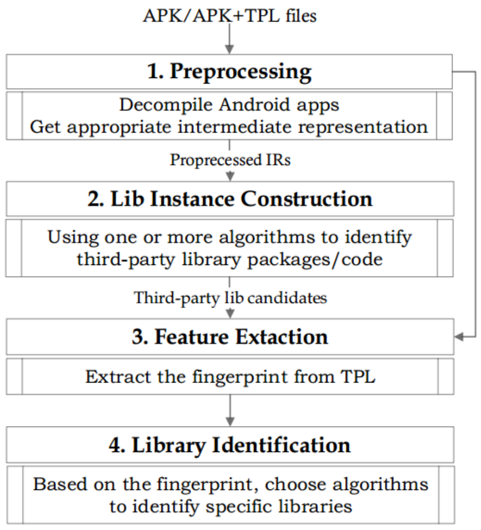
APK/APK+TPL files
1. Preprocessing
Decompile Android apps
Get appropriate intermediate representation
Proprecessed IRs
2. Lib Instance Construction
Using one or more algorithms to identify third-party library packages/code
Third-party lib candidates
3. Feature Extaction
Extract the fingerprint from TPL
4. Library Identification
Based on the fingerprint, choose algorithms to identify specific libraries

Figure 1: Typical process of TPL detection

conduct module decoupling algorithms to get independent modules as candidate library instances.

Step 3: **Feature Extraction.** This step is to extract the features of TPLs which can uniquely represent different TPLs. Existing tools usually extract features such as Android APIs, control flow graph, and variant method signatures to represent TPLs.

Step 4: **Library Instance Identification.** Depending on different library instance construction methods in the second step, existing identification methods can be classified into two types: *clustering-based* method and *similarity comparison* method. The clustering-based method usually depends on sophisticated module decoupling techniques. This method needs to filter out the primary module (i.e., code of the host app) and then cluster non-primary modules with similar features together. The modules in one cluster are considered as a TPL. The similarity comparison method considers all the modules in an app as TPL candidates, thus, it requires collecting TPL files first. By comparing the similarity of the features between the collected TPLs and the in-app TPL candidates, in-app TPLs can be identified.

2.3 Code Obfuscation Strategies

Code obfuscation is often used to protect software against reverse engineering. There are many obfuscators (i.e., obfuscation tools) such as Allatori [8], DashO [13], Proguard [21] helping developers obfuscate their apps and TPLs. Some obfuscation techniques can hide the actual logic of the apps as well as the used libraries. The commonly-used obfuscation strategies are introduced as follows.

Identifier Renaming, which renames identifiers into meaningless characters such as "a" and "b", including the class name, the method name and the file name, etc.

String Encryption, which usually adopts encryption algorithms to protect sensitive information such as telephone or email. After encryption, the sensitive strings defined in the source code are encrypted to meaningless strings.

Package Flattening, which modifies the package hierarchy structure by moving the files from one folder to another. Different obfuscators can flatten the structure to varying degrees. Sometimes the whole package structure can be removed, and all the files are put into the root directory of apps.

Dead Code Removal, which deletes unused code and preserves the functionalities invoked by the host app.

Control Flow Randomization, which modifies the Control Flow Graph (CFG) without changing the actual execution tasks, e.g., inserting redundant control flow or flattening control flows.

Dex Encryption, which allows developers to encrypt the whole DEX file. It can encrypt user-defined functions as well as Android components such as Activities and Services. The protected classes would be removed from the original classes.dex files, thus, cannot be obtained by reverse-engineering tools.

Visualization-based Protection, which translates the code into a stream of pseudo-code bytes that is hard to be recognized by the machine and human. Such apps should be executed in a specific runtime, which will interpret the pseudo-code.

3 RELATED WORK

TPL Detection. Third-party library detection plays an important role in the Android ecosystem, such as malware/repackaging detection, where TPLs are considered as noises, thus should be filtered out. Most malicious/repackaged apps detection employed a whitelist-based method to detect TPL based on the package name. Chen *et al.* [36] collect 73 popular libraries as the whitelist to filter third-party libraries when detecting app clones. Repackaging detection tools [37, 44, 66] and malware detection tool [45] also adopt the whitelist-based method to remove third-party libraries. However, such a method exists hysteresis and lacks robustness, which cannot cover all TPLs and finds emerging libraries, as well as obfuscated libraries. To seek more effective approaches to find in-app TPL, various detection tools appear. We will elaborate on these tools in the following sections.

Android Testing Tool Comparison. Shauvik et al. [40] compared the effectiveness of Android test input generation tools based on four aspects: ease of use, compatibility, code coverage, and fault detection ability. They reveal the strengths and weaknesses of different tools and techniques. Xia et al. [61] also conducted an empirical study of various Android input generation tools and found Monkey could get the best performance. They also developed a new method to improve the code-coverage of Monkey. Kong et al. [46] reviewed 103 papers related to automated testing of Android apps. They summarized the research trends in this direction, highlighted the state-of-the-art methodologies employed, and presented current challenges in Android app testing. They pointed out that new testing approaches should pay attention to app updates, continuous increasing of app size, and the fragmentation problem in the Android ecosystem. Fan et al. [41, 42, 57] evaluated the effectiveness of both dynamic testing tools and static bug detection tools in Android apps, especially for Android-specific bugs. Chen et al. [38, 39] evaluated the effectiveness of static security bug detection tools for Android apps.

Clone App Detection Comparison. Li et al. [48] surveyed 59 state-of-the-art approaches of repackaged app detection, in which they compared different repackaging detection techniques and elaborated current challenges in this research direction. They found that current research on repackaging detection is slowing down. They also presented current open challenges in this direction and compared existing detection solutions. Besides, they also provided a dataset of repackaged apps, which can help researchers reboot this research or replicate current approaches. Zhan et al. [62] conducted a comparative study of Android repackaged app detection. They reproduced all repackaged app detection tools and designed a taxonomy for these detection techniques and then analyzed these techniques and compared their effectiveness. Finally, they listed the advantages and disadvantages of current techniques. Furthermore, Baykara et al. [35] investigated malicious clone Android apps. They revealed potential threats that can affect users' experience. Finally, they provided some potential solutions for these risks.

4 OVERVIEW OF TPL DETECTION

To investigate existing TPL detection techniques, we first follow a well-defined Systematic Literature Review (SLR) methodology [1, 60] to find related research in this area. We search the candidate papers from four digital databases: ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect and top conferences and Journals on both software engineering and security. We do not consider posters [34] or short papers that provide a preliminary idea. Finally, we get nine publications to compare and analyze. *LibD*^{2} [49] is an extension of LibD [50], therefore we discuss them together. Based on the detection process, we introduce and compare the state-of-the-art TPL detection techniques, with details shown in Table 1.

4.1 Preprocessing Comparison

In the preprocessing stage, we can find from Table 1 that Apktool [9] is the most frequently-used tool (5/9). Androguard [5] can be used to generate the class dependency relationship; both Androguard and Soot [22] can be used to construct CFGs. Besides, Androguard and Apktool can restore the package structure, and each independent tree structure indicates a package hierarchy structure, which is used by some systems (e.g., LibRadar [53], LibPecker [64]) as a supplementary feature to construct the library instances.

4.2 Library Instance Construction Comparison

As shown in Table 1 on library instance construction, apart from the package name (PN), another three features are used to identify the boundaries of TPLs: (1) *package hierarchy structure (PHS)*. PHS is a tree, which can be treated as a directed graph where each node indicates a package, a sub-package or a file, and each edge indicates the inclusion relations between two nodes. Tools (i.e., LibID, LibPecker, [43], LibRadar) that use PHS regard each independent directory tree as a library instance candidate. (2) *Homogeny graph*, which indicates the parent or sibling relations between two nodes, including call relations, inheritance relations, and inclusion relation [50]. (3) *package dependency graph (PDG)*. PDG considers the dependency in the intra-packages [56], including member field reference relation, method invocation relation, inheritance relation, and intra-package homogeny relation. Different dependency relations will be set different weights based on intimacy.

Insights. We give a brief discussion about the three features. (1) Tools that only depend on the PHS (e.g., LibID, LibPecker, LibRadar) may miss some TPLs. For instance, TPLs can be inserted into the package of the host app as part of the host app, which may be deleted during pre-preprocessing without further consideration. (2) Tools depending on the homogeny graph (i.e., LibD), if packages of two TPLs have the inclusion or inheritance relations, they may be considered as one TPL. (3) Tools depending on PDG (i.e., LibSift, AdDetect) would be more reliable than other tools since the PDG-based method considers both the PHS and homogeny relations, and it splits an app into different parts based on the package dependency.

4.3 Feature Extraction Comparison

We use two metrics to compare the feature extraction process of existing TPL detection methods: feature generation method and signature representation.

Feature generation method. As shown in Table 1, LibID and LibPecker exploit class dependency relations as features, including class dependency, class inheritance dependency, field dependency, and method prototype dependency, but they adopt different hash algorithms to generate signatures. LibD and Han et al. [43] use opcode from CFG blocks as features and use hash methods to generate the opcode. The only difference is that besides the opcode, Han et al. adopt *Method Type Tag* as well. Both ORLIS and LibScout select the fuzzy method signature as the feature, but with different generation methods. LibScout uses the Merkel tree to generate the hash to represent a TPL based on the package structure. ORLIS first uses one feature hash algorithm (sdhash) [4] to hash the fuzzy method signature to represent the library-level signature and then applies the ssdeep hash algorithm to generate the class-level feature. LibRadar exploits the Android APIs, the total number of Android APIs, and the number of API types to construct the feature vector. LibRadar calculates the hash value of the feature vector as the final fingerprint. AdDetect extracts app component usages information, device identifiers and users' profile, Android permissions, as well as Android APIs to represents ad library features.

Signature Representation. Based on Table 1, we can find five systems that adopt hash value to represent features. LibScout exploits Merkle Tree to generate the TPL feature, and the feature representation is also a hash value at the package level. Note that LibSift does not identify specific TPLs but split independent TPL candidates out. AdDetect employs static analysis to extract the code feature represented as vectors.

4.4 Library Identification Comparison

In this stage, the comparison features have two different granularity: the fine-grained features at the class level and the coarse-grained features at the package level. From another perspective, the identification strategies can be divided into three categories: 1) similarity comparison, 2) clustering-based method, and 3) classification-based method. Table 1 shows that LibD and LibRadar choose the clustering method to identify TPLs, which does not require collecting ground-truth TPLs to build a database. Compared with the clustering method, the similarity comparison methods usually conduct pairwise comparison, which needs to collect the TPL files as ground truth. If the similarity between the in-app TPL and a TPL in the database is large than a pre-defined threshold, tools will consider it as a TPL. Classification-based methods (i.e., AdDetect) employ SVM to classify the ad/non-ad libraries.

Table 1: A comparison of existing TPL detection systems (in descending order by publication year)

Tool (Year)	Tool Available	Preprocessing Tool	Lib Instance Construction		Feature Extraction		Library Identification	
			Feature	Method	Feature	Method	Granularity	Method
LibID (2019)	✓	Androguard dex2jar	PHS,PN	Construct GT BIP	Class dependency	LSH	Class	Similarity comparison
LibPecker (2018)	✓	Apktool Androguard	PHS, PN	Construct GT	Class dependency	Hash	Class	Fuzzy class match (Similarity comparison)
Han et al. [43](2018)	✗	Androguard	PHS, PN	Construct GT	Opcode of CFG Basic Block	Hash	Package	Similarity comparison
LibD (2017)	✓	Apktool Androguard	Homogeny graph, PN	-	Opcode of CFG Basic Block	Hash	Package	Clustering
ORLIS (2018)	✓	Soot	-	Construct GT	Method Signature	Hash	Class	Similarity comparison
LibRadar (2016)	✓	Apktool	PHS, PN	-	API calls Number/types of API	Hash	Package	Clustering
LibSift (2016)	✗	Apktool	PDG	HAC	-	-	Package	-
LibScout (2016)	✓	-	-	Construct GT	Method Signature	Merkle tree	Package	Fuzzy match (Similarity comparison)
AdDetect (2014)	✗	Apktool	PDG	-	API, Permission, etc. (Feature vector)	Static analysis	Package	SVM

PN: package name, PHS: package hierarchy structure; GT: ground truth; PDG: package dependency graph;
LSH: Locality-Sensitive Hashing: HAC: Hierarchy Agglomerative Clustering; BIP: Binary Integer Programming models

5 EMPIRICAL STUDY DESIGN

In this section, we attempt to thoroughly compare the state-of-the-art TPL detection tools using the following four criteria:

C1: Effectiveness. We compare the effectiveness of existing tools on a unified dataset (without bias) by using three metrics: recall, precision, and F1-Score [25].

C2: Efficiency/Scalability. We compare the detection time of each tool and point out the tools that are scalable to large-scale detection and can be extended for industries.

C3: Capability of Obfuscation-resilience. Based on a previous study [51], more than 50% TPL in apps are obfuscated. Obfuscated TPLs can affect the detection accuracy. We thus compare the obfuscation-resilience capability of each tool against different obfuscation strategies. Besides, for the same obfuscation strategy, different obfuscators (obfuscation tools) have various implementation schemes, we also compare the detection ability of existing TPL detection tools against different obfuscators.

C4: Ease of Use. Usability of a tool is usually the primary concern for users. Thus, we attempt to reveal the usability of each detection tool by designing a survey to investigate different users' using experiences and let users rate each tool.

5.1 Tool Selection

Our evaluation only considers the publicly available tools in Table 1, among which LibD is reported containing an error in terms of the hash method by the owner [7], we thus excluded it in this paper but we still conduct the comparison of LibD. The detailed comparison results can be seen in our website [28]. Eventually, we consider presenting the comparison results of five tools (i.e., LibID, LibRadar, LibScout, LibPecker, and ORLIS) here.

5.2 Data Construction

For the evaluation dataset, we collect two datasets for different purposes: (1) Detecting TPLs in closed-source apps (e.g., from Google play store) to evaluate the effectiveness/efficiency of each tool in the real world (C1 & C2). (2) Detecting TPLs in open-source apps with/without obfuscation to evaluate the obfuscation-resilient capability of each tool (C3). The reason we use a separate dataset for assessing C3 is that the first dataset: **1) Lacks controlled trials.** To evaluate the obfuscation-resilient capability, we need to collect apps with/without code obfuscation. However, the real-world apps from Google Play cannot meet this condition; **2) Lacks ground truth for code obfuscation.** The apps from Google Play may be obfuscated by developers, and we cannot know which tool they use to obfuscate apps and which obfuscation techniques are adopted. Therefore, the first dataset cannot be used to evaluate C3.

5.2.1 Dataset for Effectiveness/Efficiency Evaluation. This dataset needs to meet two requirements: 1) providing the mapping information between apks and TPLs; and 2) providing a full version set of each TPL. Note that we need to collect the TPLs with their full versions to ensure fairness when comparing these tools. The reasons are as follows: (1) We can only know the libraries used in an app by referring to some websites, such as AppBrain [2], without knowing the specific library version. Even for the same TPL, the code similarity of different versions also varies, ranging from 0% to 100%. If an app uses TPLs whose versions are not included in the TPL dataset, it could cause false negatives when the code similarity of two versions is below the defined threshold. Thus, to eliminate the side-effects caused by the incomplete versions of TPLs, we should collect the TPLs with their full versions. (2) The ways in which the libraries update are diverse. Some TPLs require developers to manually update them while some TPLs support automatic update. Therefore, it is difficult to ensure the specific mapping relations between TPLs' version and some apps.

We find that only ORLIS and LibID released a dataset to evaluate the capability of obfuscation-resilience. However, the number of in-app TPLs from open-source apps is usually small, and most of TPLs are non-obfuscated, which cannot reflect the ability of these tools to handle real-world apps. Besides, they do not provide full versions of TPLs in the dataset, which may lead to bias for some tools. Therefore, we need to collect both the real-world apps and the full versions of used TPLs.

TPL Collection. We use `library-scraper` [19] to crawl TPL files from Maven Central [20], Jcenter [18], Google's maven repository [17], etc. We refer to AppBrain [2] to get the app-library mapping information and manually check their relationship to ensure the correctness. Besides, we also crawl the apps that use the TPLs in our dataset from Google Play. We filter out TPLs whose full versions are not included in our dataset. Finally, we select 59 unique TPLs and 2,115 corresponding library versions.

App Collection. According to the collected TPLs, we can acquire the apps using these TPLs from AppBrain. We download 221 Android apps (the newest versions) that use at least one TPL in our collected TPL database from Google play. This dataset containing TPLs and the corresponding apps is used as the ground-truth to evaluate the accuracy and performance of each tool. We clarify a confusing concept here. LibRadar and LibD adopt clustering-based method to identify in-app TPLs, which require considerable number of apps (million-level) as input to generate enough TPL signatures. Whereas, we collect apps to verify their performance here, thus, we do not need so many apps here. Moreover, The size of our dataset is closed to existing similarity detection tools, such as LibScout [33].

5.2.2 Dataset for Obfuscation Evaluation. In order to investigate the obfuscation-resilient ability of existing available tools in terms of Android apps protected by code obfuscation techniques, we employ the benchmark [11] containing 162 open-source apps downloaded from F-Droid [15], mapping to 164 TPLs. The dataset includes two parts: apps with non-obfuscated TPLs and the corresponding obfuscated ones. We use the dataset to evaluate two aspects: *1) capabilities towards different obfuscation tools; 2) capabilities towards different obfuscation techniques.*

To evaluate the abilities regarding different obfuscators, TPLs in each app from our benchmark are obfuscated by using three obfuscators (i.e., *Proguard* [21], *DashO* [13], and *Allatori* [8]), respectively. Finally, our dataset includes four sets: 162 non-obfuscated apps with three sets of apps (162 × 3) whose TPLs are obfuscated by three obfuscators, respectively. To further investigate the capability regarding different obfuscation strategies, we randomly choose 88 open-source Android apps in the previous experiment and choose DashO to obfuscate the 88 apps (non-obfuscation) with different obfuscation techniques (i.e., control flow randomization, package flattening, and dead code removal). Finally, we get three groups (88 × 3) of obfuscated apps.

6 EVALUATION

Our experiments were conducted on 3 servers running Ubuntu 16.04 with 18-core Intel(R) Xeon(R) CPU @ 2.30GHz and 192GB memory.

6.1 C1: Effectiveness

We aim to compare existing library detection tools regarding the dataset collected in Section 5.2. Note that LibRadar (a clustering-based method) have published their TPL signature database, we directly employ this database to evaluate its effectiveness.

Overall Results. As shown in Figure 2, we can observe that most existing tools can achieve high precision but all tools have low

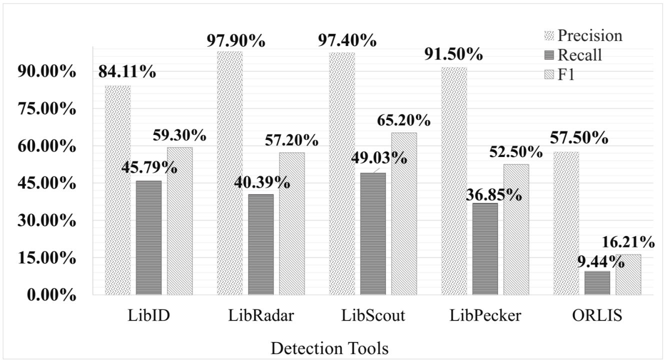

Figure 2: Detection result of different TPL detection tools

recall (i.e., less than 50%), indicating that existing tools can only detect less than half of the TPLs used by the apps. LibScout (49.03%) achieves the highest recall, followed by LibID (45.79%). As for the precision, LibRadar achieves the best performance, which reaches 97.9%, the precision of LibRadar (97.90%) and LibScout (97.40%) and LibPecker (91.50%) are very close; all of them are above 90%. The precision and recall of ORLIS are the lowest among these tools, which are 57.50% and 9.44%, respectively. To evaluate the comprehensive performance of these tools, we use the F1 value as an indicator. We can see that LibScout outperforms other tools, achieving 65.20%, followed by LibID (59.30%). ORLIS has the lowest performance, reaching only 16.21%.

FP Analysis. The false positives are mainly caused by two reasons. The first one is due to the TPL dependency. If a TPL LIB3 is built on LIB1 and the test app include the LIB3, LIB2 and LIB3 are the same TPL but different versions. LIB2 does not depend on other TPLs, the core code of LIB2 and LIB3 are the same, and the signatures of LIB1, LIB2 as well as LIB3 are stored in our database. When a tool search the database, the in-app modules may match LIB1 and LIB2 at the same time, leading to false positives.

Other false positives come from the code of closed versions with high similarity and some TPL detection. tools (e.g., LibRadar, LibID, and LibScout) choosing package hierarchy as a supplementary feature to identify TPLs. Different versions of the same TPL may have different package hierarchy structure. Taking the library "OkHttp" as the example, for the versions before 3.0.0, the root package was "com/squareup/okhttp", while it changed to "okhttp3" for versions after 3.0.0. They are considered as two different TPLs since they have different root package structures. These tools find the TPLs but report it twice; one of them is regarded as a false positive. This example also illustrates that these tools cannot identify the root package mutation of the same TPL.

FN Analysis. We take an in-depth analysis of the reasons for the low detection rate of these tools. There are two reasons affecting the recall: 1) code obfuscation, 2) TPL identification methods.

Apps in our dataset are from Google Play Store, which may be obfuscated by developers. If the obfuscator removes the whole package structure of a TPL and put all files in the root directory of apps, this code obfuscation can dramatically decrease the detection rate of all tools. Dead code removal can also affect all tools. Moreover, the package structure can affect the fingerprint generated by LibScout and LibRadar, and the package name mutation can affect the

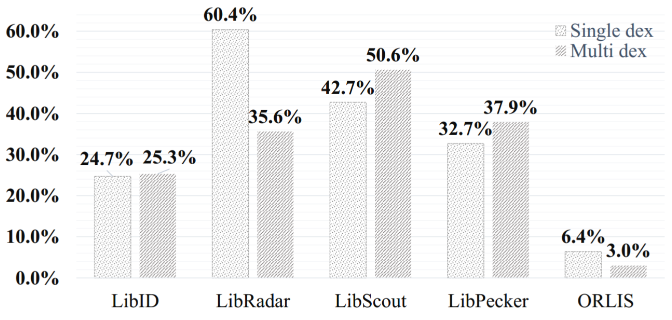

Figure 3: Detection rate of different tools towards handling multiple Dex and single Dex problems

TPL identification of LibID, LibRadar. Therefore, code obfuscation causes false negatives of current TPL detection tools.

In addition, the selected identification algorithms may also affect the detection rate. The detection rate of clustering-based methods primarily relies on the number of collected apps and the reuse rate of TPLs used by these apps. It may cause false negatives when an insufficient number of apps are collected for clustering. Besides, clustering-based tools assume the modules that are used by a large number of apps are TPLs. This assumption causes it can only find some widely-used TPLs. If some TPLs are seldom used by apps or very new, they will fail to identify them. Another minor reason for the false negatives of LibRadar is that its pre-defined database may not contain the TPLs we use in our experiment. In contrast, the similarity comparison algorithm can alleviate the FN caused by rarely-used TPLs by adding them to the database. From our experimental result, we can find the recall of the similarity comparison methods (i.e., LibID, LibScout) is higher than that of clustering-based methods (i.e., LibRadar, LibD).

Besides, we find LibID, LibPecker and ORLIS cannot handle TPLs in ".aar" files. In fact, a TPL could include both the ".aar" format files and ".jar" format files. In our dataset, about 52% TPLs are represented in ".aar" format. Thus, this is another reason that results in the false negatives of these three tools. We also find that LibID would report some errors when it profiles the TPL files by using dex2jar [14], above reasons lead to 1,106 TPL signatures missing in the database of LibID and directly increase the FN of LibID.

Multidex Problem. Another reason which can affect the recall is the 64K limit problem. In Android, the executable Java code is compiled into Dalvik Executable (Dex) file and is stored in the APK file. The Dalvik Executable specification limits the total number of method references to 64K in a single Dex file. Since Android 5.0 (API level 21), it supports to compile an app into multiple Dex files if it exceeds the limited size. We thus refer to the source code of these publicly-available tools and find that LibRadar and ORLIS only consider the analysis on a single Dex file, which causes the missing of a considerable amount of TPLs in detection. The specific detection result of these tools to handle the single dex and Multidex is shown in Figure 3. The result also indicates that when LibRadar and ORLIS deal with the single dex, the detection rate of LibRadar and ORLIS are 60.4% and 38.6%, respectively. However, when they handle the apps with multi-dex, the detection rate decreases by almost half than the original one.

Table 2: Detection time of different tools

Tool	LibID	LibRadar	LibScout	LibPecker	ORLIS
Q1	2.07h	5s	47s	3.38h	876.75s
Mean	23.12h	5.53s	82.42s	5.11h	1438.50s
Median	6.56h	5s	66s	4.65h	1199.5s
Q3	20.04h	6s	95.25s	6.46h	1571.5s

We further investigate the percentage of such multiple Dex apps in our evaluation dataset. We find that only 41 apps contain a single Dex file, and the remaining 180 apps contain multiple Dex files, more than 80% of them contain three Dex files. Surprisingly, the app "com.playgendary.tom" even contains 98 Dex files. For example, "bubbleshooter.orig" uses 19 TPLs, but the classes.dex file only contains 4 TPLs of them; the remaining libraries are included in other `classes_N.dex` files. Without a doubt, if the tool just considers the single Dex situation, its effectiveness (especially the recall) will be significantly decreased. Therefore, we suggest that TPL detection tools should take into account multiple Dex situations to ensure reliable results.

Answer to C1: As for effectiveness, LibScout performs the best. Most TPL detection tools achieve high precision but low recall since they more or less depend on package name/structure (which is fragile and unreliable) as auxiliary features to generate TPL signatures, leading to lots of false negatives.

6.2 C2: Efficiency

Considering efficiency, we compare the time cost of each tool. To ensure a fair comparison, we first employ each tool to generate TPL features for these tools which do not offer database. The detection time does not include the TPL feature generation time, which is the practice in all related work. The time cost consists of the TPL detection process for an app, i.e., pre-process an app, profile the app, extract the code feature, identify TPLs inside. For each tool, the detection time is closely related to the number of TPLs in the collected database and the number of TPLs in each app.

Table 2 shows the detection time of the five selected tools. The average detection time of LibRadar is 5.53s, which is the fastest one among these tools, followed by LibScout (82.24s). LibRadar can directly process the `classes.dex` files by using their tool LIBDEX [6] that dramatically improves the performance while other tools adopt the reverse-engineering tools (e.g., Soot and Androguard) that are much more time-consuming. Besides, as we mentioned in Section 6.1, LibRadar only handles the single Dex file and ignores the multiple Dex problem, which is also one of the reasons LibRadar is faster than other tools.

The detection time of LibID and LibPecker is much longer; the median detection time of these two tools is more than **4** hours for each app. Surprisingly, the average detection time of LibID is even nearly one day per app. We find that if the size of a dex file of a TPL is larger than 5MB, the detection time of LibID dramatically increases. Based on our observation, we find that LibID cannot handle too many TPLs at one time and is computation heavy in terms of CPU and memory. For each detection process, LibID needs to load all the features of TPLs in the memory, and it costs about 21 minutes to load all data. Furthermore, the average detection time of ORLIS is 1438.50s per app. These three tools are not suitable for large-scale TPL detection. We summarize two factors affecting the efficiency of different tools as follows.

Comparison Strategy. We found that comparison strategies dramatically affect efficiency. LibRadar and LibD first compare the top package level hash and the order of comparison is top-down, which is more efficient. While LibPecker, LibID, LibScout, and ORLIS adopt the pair-wise comparison strategy to identify a specific library and the comparison strategy of LibID, LibPecker, and ORLIS are bottom-up. We find this comparison strategy is more time-consuming.

Feature Granularity. We also found that the granularity of TPL code feature can affect detection efficiency. Currently, there are two levels of granularity used by existing TPL detection tools: package-level (i.e., LibRadar and LibScout) and class-level (i.e., LibID, ORLIS, and LibPecker). The number of package-level items is far less than that of the class level. According to our experimental result, we can also observe that the overhead of systems that use class-level features is higher than systems using package-level features. Therefore, the efficiency of LibID, LibPecker and ORLIS is obviously worse than other tools. If the functionality of a TPL is complicated, the number of classes and methods could also increase, thus the comparison time increases accordingly. The growth rate of comparison time between the lib-class and app-class is exponential. Thus, we can find that the average detection time of LibID, LibPecker and ORLIS is much longer than the median time. This is because it takes more time to extract code features of some complicated TPLs. Take LibPecker as an example, even if an app contains just one TPL, one comparison could cost time ranging from about 5s to 10s, and there are 2,115 TPL features to be compared with in our dataset, the average detection time could reach to 5.11 hours.

Answer to C2: LibRadar outperforms existing TPL detection tools in terms of the detection time, taking only 5.53s per app on average. LibID takes more time than others, almost one day per app. Feature granularity can affect the performance of detection tools, and the features at class-level cost more system resources and detection time.

6.3 C3: Obfuscation-resilient Capability

In this section, we attempt to investigate the obfuscation-resilient capability of existing tools from two aspects: (1) towards different obfuscation tools; and (2) towards different obfuscation strategies, by comparing their detection rate. The detection rate is the ratio of the number of the correct identified TPLs to the total number of TPLs in the ground truth.

6.3.1 *Evaluation towards Different Obfuscators.*

Users can configure the obfuscation strategies of obfuscators by themselves. The obfuscation strategies of the three obfuscators are shown in Table 3. We can see that *Proguard* only enables two strategies, including identifier renaming and package flattening while *DashO* enables all of the listed strategies. We compare the detection rate of the five tools on apps with/without being obfuscated by different obfuscation tools, in an attempt to investigate their effectiveness.

Table 3: Enabled obfuscation strategies of each obfuscator

Obfuscation strategy	Proguard	Allatori	DashO
Dead Code Removal	✗	✗	✓
String Encryption	✗	✓	✓
Control Flow Randomization	✗	✓	✓
Identifier Renaming	✓	✓	✓
Package Flattening	✓	✓	✓

✓ : enabled ✗: disabled

Table 4 shows the detection results. As for the apps without code obfuscation, LibPecker outperforms others, reaching 98.72%, followed by LibScout with 88.73% of detection rate. The performance of LibID is the worst, only reaching 11.70%. We can note that the performance of LibID in C1 and C3 has orders of magnitude of differences. The recall of LibID dramatically dropped in C3. That is because the detection capability of LibID is greatly limited by dex2jar [14]. Most problems are caused by the compatibility of TPLs. Java only supported the Android to the version eight, if a TPL is developed by Java 9+, dex2jar cannot identify some new features in this TPL. We find that many TPLs cannot be decompiled successfully in this dataset. Thus, LibID cannot generate code signatures for these TPLs, leading to false negatives when the TPL to be identified is matched with these unsuccessful-decompiled TPLs. Besides, the experiment in C1 has TPLs with full versions while in this experiment, we can ensure the specific TPL version in each app in C3. Therefore, we just give one version in this dataset, which also leads to the recall decrease of LibID in C3 because our detection granularity is at the library level instead of versions. Above mentioned reasons lead to the inconsistent results of LibID in C1 and C3. More importantly, for apps with obfuscation, we can find that the detection rate of all tools remain unchanged for apps obfuscated by Proguard, indicating that all tools can effectively detect TPLs in apps obfuscated by Proguard. However, they all fail to effectively detect TPLs obfuscated by Dasho, leading to a sharp decline in detection rate. LibScout is the worst one, i.e., 22.02% (dropping by 66.71%), followed by LibPecker, LibRadar and ORLIS, and they reduce to 34.95% (dropping by 64.13%), 8.96% (dropping by 56.47%) and 30.41% (dropping by 33.10%), respectively. The code obfuscation effect of Allatori lies between Proguard and DashO.

There are two main reasons for such differences in detection rate decline: (1) different enabled obfuscation strategies in different obfuscation tools; the more obfuscation strategies are enabled, the lower the detection rate is. (2) Even if apps use the same code obfuscation technique, different tools have different implementations and final effects may also be different. In our dataset, we find that the package flattening strategy implemented by Proguard only changes the package name, while both Allatori and DashO change the package name/hierarchy structure but DashO also includes the class encryption in this process, which directly affects the detection rate. It is worth noting that the recall (without obfuscation) of some tools is higher than that in C1 because the apps in C1 are closed-source apps from Google Play, and some of them have been obfuscated by developers.

Based on the result, we can find that all TPL detection tools are obfuscation-resilient to identifier/ package renaming. DashO has the best obfuscation performance and LibPecker has the best performance to defend against the three popular obfuscators.

Table 4: Results of code obfuscation-resilient capability for different obfuscators

Tool	Without obfuscation	With Obfuscation		
		Proguard	Allatori	Dasho
LibID	11.70%	11.70%	8.49%	5.80%
LibPecker	**98.72%**	**98.72%**	**95.13%**	**34.95%**
ORLIS	63.51%	63.51%	60.31%	30.41%
LibRadar	65.43%	65.43%	63.38%	8.96%
LibScout	88.73%	88.73%	27.40%	22.02%

6.3.2 *Evaluation towards Different Obfuscation Techniques.*

We evaluate the capabilities of the five selected TPL detection tools towards defending against three obfuscation techniques: 1) control flow obfuscation, 2) package flattening, 3) dead code removal. The change of CFG structure can affect some tools that employ CFG as signature. Package flattening technique can affect tools depending on the package hierarchy to generate code features, for example, LibRadar computes a hash value for each package level; LibScout uses the package tree to generate code features. Besides, dead code removal can remove some fingerprints of TPLs and decrease the detection rate. We aim to investigate how the code obfuscation techniques affect the detection rate of each tool, and which technique has the most prominent effects on these detection tools.

According to Table 5, for apps without obfuscation, LibPecker has the highest detection rate (98.91%), followed by LibScout (87.75%). LibID achieves 12.19%, which is still the worst. For the obfuscated apps, we can see the three techniques reduce the detection rate of these tools. LibPecker outperforms other tools in defending against these obfuscation techniques, achieving 81.61% (CFO), 73.52% (PKG FLT), and 73.74% (code RMV) of detection rate. LibID still has the lowest detection rates of three obfuscation techniques are 0.00%, 0.09% and 1.45%, respectively. The reasons were elaborated in Section 6.3.1. Besides, we also find that feature granularity can affect the obfuscation-resilient capabilities. LibPecker and ORLIS achieve better performance than other tools in this experiment. LibPecker and ORLIS use fine-grained code features (i.e., class level) that are not sensitive to small code changes. In contrast, LibScout and LibRadar use coarse-grained features (i.e., package level), whose hash values may easily change due to slight code modification.

Moreover, we can see that different code obfuscation techniques have various effects on different tools. We can see that package flattening has the most prominent effect on the detection rate of LibPecker and LibRadar while it has little effect on ORLIS; control flow randomization has the least effect on LibPecker and LibRadar while has the biggest effect on ORLIS. All of the three obfuscation techniques all have great impact on the detection rate of LibScout, dropping by more than 70%. LibRadar uses APIs as the code feature. Therefore, the dead code removal and control flow randomization can affect the final library code feature. It generates the feature vector based on the package structure. The change of package structure can modify the feature vector. LibScout is sensitive to control flow obfuscation, package flattening, and dead code removal. The reason is that LibScout exploits the Merkle tree to generate the TPL profile, and it partially depends on the package hierarchy structure. Besides, LibScout uses MD5 to generate the library fingerprint, a small change can lead to the fingerprint change and finally affect the detection accuracy. Thus, the three obfuscations can affect the detection rate. Moreover, three code obfuscation techniques have similar effects on existing tools.

Table 5: Evaluation of the capabilities of existing tools for different code obfuscation techniques

Tool	Without obfuscation	With Obfuscation		
		CFO	PKG FLT	Code RMV
LibID	12.19%	***0.00%***	0.09%	1.45%
LibPecker	**98.91%**	81.61%	***73.52%***	73.74%
ORLIS	63.46%	***58.86%***	63.46%	60.61%
LibRadar	64.77%	***49.67%***	48.36%	49.02%
LibScout	87.75%	17.72%	***16.63%***	***16.63%***

CFO: Control Flow Obfuscation; PKG FLT: Package Flattening; Code RMV: Dead Code Removal

Answer to C3: LibPecker outperforms other tools in defending against different obfuscators and different obfuscation techniques. Tools using class-level features have better performance than those using package-level features regarding defending against code obfuscation.

6.4 C4: Ease of Use

Whether a tool is user-friendly is an essential factor in evaluating the usability of the tool. We attempt to compare the usability of the available tools (i.e., LibID, LibPecker, ORLIS, LibRadar, LibD[2], and LibScout) from three aspects: 1) the installation and setup process, 2) the usage steps, and 3) the result presentation. To assess them objectively, we design a questionnaire [32] and recruit participants to rate for these tools from the three aspects.

Participant Recruitment. We recruit 20 people from different industrial companies and universities via word-of-mouth, who are developers in IT companies, post-doc, Ph.D. students, etc. To minimize the interference factors due to unprofessional factors of participants, all the participants we recruited have over 3-year experience in Android app development, and they are from different countries such as Singapore, Germany, China and India. Besides, they did not install or use these tools before. The participants received a $50 coupon as a compensation of their time.

Experiment Procedure. We provided the links of the source code together with the instruction files that guide participants to install and use these tools. Note that since some tools require users to take apks (and TPLs) as input, we also provide another repository containing some sample apps and sample TPLs in case that participants have no idea about where to download the input data, which may hinder the process of using them. In fact, when conducting it in the real world, it is even more difficult since users have to find where to download and collect these input datasets, especially for the tools that require a ground-truth database beforehand. We ask the participants to install and use these six tools one by one, and rate each tool from the aforementioned three aspects. The specific rating criteria can be seen in Table 6. All participants carried out experiments independently without any discussions with each other and they were encouraged to write some comments about each

[2]We consider the usability of LibD though it was reported containing a calculation error, which would not affect the installation and using process.

Table 6: Rating options for each item in the questionnaire

Installation	Easy ★★★★	Acceptable ★★★	Complicated ★★	Very complicated ★
Usage	Easy ★★★★	Acceptable ★★★	Complicated ★★	Very complicated ★
Output	Clear & direct ★★★	Understand/ not concise ★★	Confusing ★	

tool. After finishing the tasks, we also interviewed them about the user experience with detailed records.

Results. Figure 4 shows the results of the questionnaire. For each rating item, we take the average of the rating stars from all participants. According to Figure 4, we can find that LibRadar gets the most stars while ORLIS receives the lowest score.

• **Installation:** As for the installation process, LibPecker gets the highest score (4 stars) since it only needs one command, and LibID is regarded to be the most complicated one because it requires participants to install the Gurobi Optimizer [26] and register the license by themselves. Some users commented that:

"*When I first installed LibID, I spent about 2 hours. Installing the necessary dependency (e.g., Gurobi Optimizer) takes most of my time since the instruction is confusing.*"

"*The design of the website of Gurobi Optimizer is terrible. It is difficult to follow the instructions because some of them are scattered.* "

Both LibRadar and LibD get 3 scores because they just need to install basic Java and python running environment and then users can run them. The installation of ORLIS and LibScout is almost acceptable (2.5 stars). Both of them require users to download the Android SDK. Besides, ORLIS requires users to download some dependencies and TPLs to make it run.

• **Usage:** As for the usage, LibRadar and LibD get the highest scores, while LibID is regarded as the most unsatisfactory tool. Participants said they consider more about the execution efficiency of the tools when using them:

"*For LibID, LibPecker and ORLIS, I have to wait for several minutes for some apps, sometimes LibID even needs more than 30 minutes to get the results. That's too long for me. While the execution time of LibRadar and LibScout is more acceptable.*"

"*LibID usually gets crash when detecting some TPLs. For me, it does work when processing the TPL named AppLovin, while crashes when processing the TPL named Dropbox.*"

In fact, the reason for such crashes of LibID is that the size of these TPLs is too large. LibID needs to load all the TPLs and apps into the memory first; its detection strategy usually consumes many computing resources, especially the memories and CPU. When users input many large size TPLs, it will crash when it exceeds the memory. Another reason that may affect the rating is that LibID, ORLIS and LibPecker would report an error when processing some ".aar" files and users need to modify them manually to proceed with the detection process.

• **Output:** According to the results, participants thought the detection results of LibID, LibRadar, and LibScout are much easier to understand; and ORLIS gets the lowest score since it only provides the matching relations of the class name, without telling the apps or TPLs that the class belongs to, making participants confused.

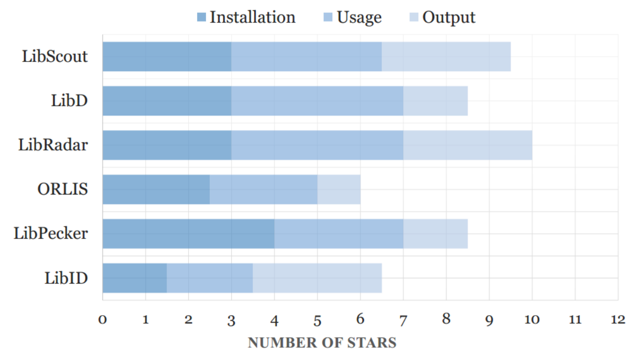

Figure 4: Rating result of each tool from the questionnaire

"*The results of LibID, LibRadar and LibScout are easy to understand. All of the results are represented in ".json" format, I can quickly find the in-app TPLs, the similarity value and other meta information.*"

The result provided by LibD is the MD5 of TPLs, which requires users to find the used TPLs by mapping with the database file provided by LibD:

"*I can understand the result of LibD, but not very direct and clear. Some information that I really interested in is missing, e.g., the similarity value and library name.*"

Answer to C4: LibRadar gets the highest score from participants mainly due to its simple usage and user-friendly output format, which is regarded as the most easy-to-use tool. ORLIS should be improved most, especially the result representation.

7 DISCUSSION

Based on our evaluation, we highlight lessons learned from different perspectives (i.e., tool users, tool implementation), and provide useful insights for future research.

7.1 Lessons Learned

7.1.1 ***From the perspective of users.*** We give an in-depth discussion on current Android TPL detection tools and propose tool selection suggestions for different stakeholders with different purposes. (1) For malware/repackaged app detection, we suggest choosing the strategy of LibSift to help filter TPLs out, because it uses the package dependency graph (PDG) as the feature and hierarchy clustering to split different TPL candidate modules from the host app. Different from the clustering-based method depending on the input apps reuse rate of in-app TPLs, the module decoupling method focuses more on the characteristics of the app itself. Using the PDG to split TPLs is more reliable than other methods that depend on package trees. Therefore, it can achieve a better recall than others and is a good choice to filter TPLs when conduct malware/repackaged app detection. (2) For vulnerable in-app TPL detection, we recommend LibScout that has better performance in identifying the specific library versions in TPL detection. It is easy to confirm the vulnerabilities via the TPL version information in vulnerability database like NVD [31]. (3) For component analysis of apps, we recommend LibPecker which is proven to have the best code obfuscation-resilient capability against common obfuscators and common obfuscation techniques. (4) For large-scale TPL detection, we suggest using LibRadar that has high-efficiency (i.e., 5.53s per app on average) and is scalable to large-scale detection. (5) For advertising investors or developers who want to choose popular ad networking (i.e., ad libraries) to show their ads, we recommend them to use LibRadar that can efficiently find commonly-used ad libraries in Android apps at market scale. With these identified TPLs, developers can choose some competitive ones to embed in their products to ensure their competitive edge in the market.

7.1.2 *From the perspective of tool implementation.*

We elaborate on the key points that are usually ignored by previous research in this paper to raise the attention to future tool implementation.

Android system updates frequently and usually introduces new features. However, current researchers seem to pay less attention to these new features, which could directly discount the detection performance. For example, (1) *Android runtime (ART) compilation mechanism* [3], proposed since 2013, is ignored by existing tools. Current tools usually can handle apps with traditional DVM compilation mechanism, which only generate one single "classes.dex" file. Apps compiled by ART will generate the ".oat" file that can be decompiled into a set of ".dex" files. Besides, we find most apps in Google Play are compiled by ART since 2015, and more than 81% of the apps in our dataset are compiled by ART, therefore, researchers should pay more attention to the new features of Android to improve their tools empirically. (2) *New app formats.* The apps published with Android App Bundle [24] format will finally be released with the file suffix ".apks" or ".xapk", which cannot be directly handled by all existing tools. We observed that some versions of the apps in our dataset are published with such formats, which should draw more attention to tool developers/researchers when implementing their tools.

7.2 Tool Enhancement

Defect Repair. Based on our study, we fixed the defects of existing TPL detection tools with: (1) ability to handle all formats of TPLs. The recall of LibID, LibPecker, and ORLIS has increased by 6.30%, 19.32%, and 12.59%, respectively. (2) Ability to handle APKs ($API >$ 21) that are compiled by ART mechanism (i.e., LibRadar and ORLIS). The recall of LibRadar and ORLIS has increased by 16.15% and 14.69%. (3) Apart from the aforementioned common problems, we find that LibID can be enhanced from two aspects: 1) Bypass reverse-engineering protection (for dex2jar tool), and 2) fix run-time errors. The detection rate of LibID increased by 14.84%. We have published the related code on GitHub [30] anonymously. Users can directly use our enhanced tools to achieve better performance. For more details, please refer to our website [29].

Online service for TPL detection. To make it more convenient for users to access and compare these tools, we build a framework which integrates the five publicly available tools and make it as an online service [29] to detect the in-app TPLs. Our framework can easily be extended if new tools are available. Users can upload an app to the online platform, and our framework will show the detection result of each tool. Users can compare each tool intuitively and clearly observe the commonalities and differences of the results.

7.3 Future Research Directions

We highlight some useful insights to inspire and motivate future research on TPL detection techniques. (1) **Select stable features.** Existing tools mostly depend on package name and package hierarchy structure to identify TPLs. However, these approaches are not reliable enough. Firstly, many different TPLs may have the same package name if they belong to the same group. For example, there are 16 different TPLs in the same group "com.google.dagger", which means these different TPLs have the same package name. It is difficult to use the package name/structure to correctly split different TPLs if one app includes some TPLs from the same group, and using package name and package structure would generate incorrect code features and lead to misidentification. Secondly, package flattening obfuscation can remove the entire package tree or modify the package structure, which also can change the signatures of the in-app TPLs. Using the package structure as the supplementary feature to split the TPL could lead to some false negatives, we suggest researchers can directly use more stable features such as the class dependency relation to split the TPLs. (2) **Consider TPLs developed in other languages.** According to our study, we find over 15% in-app TPLs are developed in Kotlin [27]. However, some grammar rules of Kotlin are different from Java, which can directly affect performance of existing TPL detection tools. Specifically, the source files of Kotlin can be placed in any directory which can cause similar effects like the package flattening obfuscation technique. Therefore, existing tools depending on package structure to generate code features may become ineffective if app developers customize Kotlin TPLs, especially modify the package name/structure that can easily change the signatures of TPLs. Besides, future researchers also can consider the native library (binary code) detection and related security problems understanding. (3) **Detect vulnerable TPLs.** Although LibScout claims that it can detect vulnerable TPLs, the complete dataset of vulnerable TPLs is missing now. We know nothing about the risks of vulnerable TPLs and infected apps. Future research on vulnerable TPL detection and understandings is necessary and meaningful. (4) **Catch emerging TPLs.** Existing tools rely on a reference database to identify TPLs, which limits their ability in detecting TPLs in the database, thus cannot identify newly-published TPLs that are not in the database. (5) **Identify TPLs by using dynamic techniques.** Current methods for TPL detection are static analysis, which cannot identify TPLs that are dynamically loaded at run-time or with dynamic behaviors.

8 CONCLUSION

In this paper, we investigated existing TPL detection techniques from both literature-based perspectives and implemented tool perspectives. We conducted a thorough comparison on existing tools from 4 aspects, including effectiveness, efficiency, code obfuscation-resilience capability, and ease of use, and summarized their advantages and disadvantages. We also discuss lessons learned from different perspectives, enhance existing tools and further provide an online service for TPL detection. Besides, our dataset and evaluation details are publicly available. We believe our research can provide the community with a clear viewpoint on this direction and inspire future researchers to find more creative ideas in this area.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. This work is partly supported by the National Natural Science Foundation of China (No. 61702045), the Australian Research Council (ARC) under projects DE200100016 and DP200100020, the Hong Kong RGC Projects (No. 152223/17E, 152239/18E), the Hong Kong PhD Fellowship Scheme and the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001.

REFERENCES

[1] 2007. survey. Guidelines for performing systematic literature reviews in software engineering.
[2] 2010-2019. AppBrain. https://www.appbrain.com/stats/libraries/.
[3] 2013. ART. https://source.android.com/devices/tech/dalvik.
[4] 2013. sdhash. http://roussev.net/sdhash/sdhash.html.
[5] 2016. Androguard. https://github.com/androguard/androguard.
[6] 2016. *LibRadar.* https://github.com/pkumza/LibRadar
[7] 2017. *LibD.* https://github.com/IIE-LibD/libd
[8] 2019. Allatori. http://www.allatori.com/
[9] 2019. Apktool. https://ibotpeaches.github.io/Apktool/.
[10] 2019. *App Future.* https://www.smashingmagazine.com/2017/02/current-trends-future-prospects-mobile-app-market/
[11] 2019. *Benchmark data.* https://github.com/presto-osu/orlis-orcis/tree/master/orlis/open_source_benchmarks
[12] 2019. BitBucket. https://bitbucket.org/
[13] 2019. *DashO.* https://www.preemptive.com/products/dasho/overview
[14] 2019. dex2jar. https://github.com/pxb1988/dex2jar
[15] 2019. *F-Droid.* https://f-droid.org/en/packages/
[16] 2019. Github. https://github.com/
[17] 2019. *Google Mvn.* https://dl.google.com/dl/android/maven2/index.html
[18] 2019. *Jcenter.* https://jcenter.bintray.com/
[19] 2019. *Library Scraper.* https://github.com/reddr/LibScout/blob/master/scripts/library-scraper.py
[20] 2019. *Maven.* https://mvnrepository.com/
[21] 2019. *Proguard.* https://www.guardsquare.com/en/products/proguard
[22] 2019. *Soot.* https://github.com/Sable/soot
[23] 2019. statista. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
[24] 2020. Android App Bundle. https://developer.android.com/platform/technology/app-bundle.
[25] 2020. F1 score. https://en.wikipedia.org/wiki/F1_score.
[26] 2020. gurobi. https://www.gurobi.com/.
[27] 2020. kotlin. https://kotlinlang.org/.
[28] 2020. *LibDetect.* https://sites.google.com/view/libdetect
[29] 2020. LibDetect. https://sites.google.com/view/libdetect/.
[30] 2020. *LibID updated code.* https://github.com/MIchicho/LibID
[31] 2020. *National Vulnerability Database.* https://nvd.nist.gov/
[32] 2020. Questionnaire of User Study. https://forms.gle/ueJAkuone9ZnCXn68.
[33] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *CCS*.
[34] Salman A. Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. 2017. Identifying Android Library Dependencies in the Presence of Code Obfuscation and Minimization. In *Proceedings of the 39th International Conference on Software Engineering Companion*.
[35] M. Baykara and E. Colak. 2018. A review of cloned mobile malware applications for Android devices. In *Proc. ISDFS*. 1–5. https://doi.org/10.1109/ISDFS.2018.8355388
[36] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.
[37] Kai Chen, Peng Liu, and Y. Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proc. ICSE*.
[38] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proceedings of the 42st International Conference on Software Engineering*. IEEE Press, 596–607.
[39] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? What can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 797–802.
[40] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proc. ASE*.
[41] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 486–497.
[42] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 408–419.
[43] Hongmu Han, Ruixuan Li, and Junwei Tang. 2018. Identify and Inspect Libraries in Android Applications. *Wireless Personal Communications vol 103, pp491-503* (2018).
[44] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. 2012. Juxtapp: a scalable system for detecting code reuse among Android applications. In *Proc. DIMVA*.
[45] C. Kai, W. Peng, L. Yeonjoon, Wang XiaoFeng, Zhang Nan, Huang Heqing, Zou Wei, and Liu Peng. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proc. USENIX Security*.
[46] P. Kong, L. Li, J. Gao, K. Liu, T. F. BissyandÃ©, and J. Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* 68, 1 (March 2019), 45–66.
[47] Li Li, Taegawende Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *SANER*.
[48] L. Li, T. F. Bissyande, and J. Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2901679
[49] M. Li, P. Wang, W. Wang, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo, and W. Zou. 2018. Large-scale Third-party Library Detection in Android Markets. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2872958
[50] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-party Library Detection in Android Markets. In *Proc. ICSE*.
[51] J. Lin, B. Liu, N. Sadeh, and J.I. Hong. 2014. Modeling users mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Proc. SOUPS*.
[52] B. Liu, B. Liu, H. Jin, and R. Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys*.
[53] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proc. ICSE-C*.
[54] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *Proc. ISSNIP*.
[55] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Proc. ACSAC*.
[56] C. Soh, H. B. K. Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang. 2016. LibSift: Automated Detection of Third-Party Libraries in Android Applications. In *APSEC*.
[57] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why My App Crashes Understanding and Benchmarking Framework-specific Exceptions of Android apps. *IEEE Transactions on Software Engineering* (2020).
[58] Haoyu Wang and Yao Guo. 2017. Understanding Third-party Libraries in Mobile App Analysis. In *Proc. ICSE-C*.
[59] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: Obfuscation-resilient Library Detection for Android. In *Proc. MOBILESoft*.
[60] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proc. 18thInt. Conf. Eval. Assessment Softw. Eng.*
[61] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There yet in an Industrial Case?. In *Proc. FSE*.
[62] Xian Zhan, Tao Zhang, and Yutian Tang. 2019. A Comparative Study of Android Repackaged Apps Detection Techniques. In *Proc. SANER*.
[63] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *Proc. ACM WiSec*.
[64] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *SANER*.
[65] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. 2013. Fast, scalable detection of Piggybacked mobile applications. In *Proc. CODASPY*.
[66] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. 2012. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proc. CODASPY*.