Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review

Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, *Member, IEEE*, Xiapu Luo, and Yang Liu

Abstract—Third-party libraries (TPLs) have been widely used in mobile apps, which play an essential part in the entire Android ecosystem. However, TPL is a double-edged sword. On the one hand, it can ease the development of mobile apps. On the other hand, it also brings security risks such as privacy leaks or increased attack surfaces (e.g., by introducing over-privileged permissions) to mobile apps. Although there are already many studies for characterizing third-party libraries, including automated detection, security and privacy analysis of TPLs, TPL attributes analysis, etc., what strikes us odd is that there is no systematic study to summarize those studies' endeavors. To this end, we conduct the first systematic literature review on Android TPL-related research. Following a well-defined systematic literature review protocol, we collected 74 primary research papers closely related to Android third-party library from 2012 to 2020. After carefully examining these studies, we designed a taxonomy of TPL-related research studies and conducted a systematic study to summarize current solutions, limitations, challenges and possible implications of new research directions related to third-party library analysis. We hope that these contributions can give readers a clear overview of existing TPL-related studies and inspire them to go beyond the current status quo by advancing the discipline with innovative approaches.

Index Terms—Third-party library, android, literature review, applications

---

1 INTRODUCTION

ANDROID has gained tremendous popularity since it was published in 2007 [1]. Recent years have witnessed the booming market of Android apps. For example, more than 3 millions apps are available on the official Google Play Store [2]. To facilitate the development of Android apps, lots of third-party libraries (TPLs) have been developed from various providers. These TPLs provide a wide variety of development kits, plentiful UIs, and various social media plugins, and have been adopted by many apps. For instance, "Exodus Privacy" provides the up-to-date statistics on third-party libraries in Google Play, and we can find that about 51% of apps in Google Play include the analytic TPL, i.e., Google Firebase [3]; and more than 60% of the code in an Android app belongs to TPLs [4].

Unfortunately, TPL is a double-edged sword, which may bring unwanted security risks to mobile users. For example, malicious TPLs could be introduced into legitimate apps. Adversaries can repackage an app by adding malicious third-party libraries that can send premium SMS services and steal users' private information [5]. They also often modify ad libraries by changing the revenue destination of ad libraries to redirect the profits [6]. Moreover, some TPLs have been reported with behaviors invading users' privacy, such as reading the contact list information or getting users' locations [7], [8], [9], [10], [11]. The security issues caused by TPLs are mainly due to the permission mechanism of Android system, which creates a separate process and private storage for each app [12], [13]. Specific permissions are required for apps to access the system services and resources. However, this permission model works at the app-level, meaning that TPLs and the host apps share the same privileges, which leads to over-privileged problems. To solve this problem, many researchers attempted to limit the privileges of TPLs [14], [15], [16], [17], [18].

Unscrupulous developers may resize the ad libraries to induce users to click the ads, which also may lead to revenue loss or affect the normal operation of apps. We refer to

- *Xian Zhan and Xiapu Luo are with the Hong Kong Polytechnic University, Hong Kong. E-mail: {csxzhan, csxluo}@comp.polyu.edu.hk.*
- *Tianming Liu and Li Li are with the Monash University, Clayton, VIC 3800, Australia. E-mail: {tianming.liu, li.li}@monash.edu.*
- *Lingling Fan is with the College of Cyber Science, Nankai University, Nankai 300071, China. E-mail: linglingfan@nankai.edu.cn.*
- *Sen Chen is with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China. E-mail: ecnuchensen@gmail.com.*
- *Yang Liu is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore. E-mail: yangliu@ntu.edu.sg.*

all violations which affect the profits of developers and users' experience as ad fraud, which attracts many studies recently [19], [20], [21], [22].

Furthermore, vulnerabilities in third-party libraries can pollute the downstream clients, including apps or other TPLs that depend on these vulnerable TPLs. If a popular third-party library is compromised, the threats from this TPL could spread to a large number of apps and affect countless mobile devices [23], [24], [25]. The vulnerability issues in TPLs will usually cause severe consequences because recent studies [23], [24], [26], [27] revealed that app developers seldom follow the fixed scheme of TPLs and tend to delay the replacement of outdated TPLs in apps, even if these TPLs include severe vulnerabilities and could pose serious threats to mobile devices or users. Without a doubt, without patching such vulnerabilities in TPLs, they will pose unexpected threats to the entire mobile ecosystem. Therefore, many studies [7], [8], [9], [28] have been conducted to mitigate the vulnerability issues in TPLs.

TPLs may cause functionality issues in host apps because the direct or transitive dependency of TPLs can also bring dependencies conflicts, which may lead to app crash [29], [30]. An app may depend on multiple versions of the same TPL or class, but only one version can be loaded, which may lead to dependency conflict and bring some unexpected issues, such as runtime exceptions. Recent studies proposed several approaches to address this problem [29], [31], [32].

Moreover, previous research [4], [33], [34] pointed out that TPLs can be noises that could affect the performance of other app analysis, such as the detection of repackaging and malicious apps. The earlier existing studies, such as the repackaged app detection [35], [36], [37], [38], [39] usually used the whitelist to exclude TPLs. However, the whitelist-based method has many limitations. For one thing, it is impossible for whitelist-based method to enumerate all TPLs. For another, whitelist-based method relies on package names to filter TPLs, while many apps can apply obfuscation techniques to encrypt the package name within the app, leading to the overlook of TPLs. As a result, many advanced tools [23], [40], [41], [42], [43] have been proposed to detect third-party libraries.

Based on the above description, we can find that TPLs play a significant role in the entire Android ecosystem. TPLs are essential participants in app development, maintenance, and subsequent detection. Besides, TPLs also can affect the quality, rating, and security of the host apps. However, although there are many studies on TPLs, there is no systematic analysis of them. Due to the importance of TPLs, researchers may be eager to know the current research status quo and the gap of state-of-the-art studies on TPLs. To fill this gap, in this paper, we present to the community the first systematic survey of studies on Android third-party libraries. Our contributions of this paper are as follows:

- *A collection of android TPL related publications.* Following a well-defined *systematic literature review* (SLR) protocol [44], [45] and a thorough examination of the collected primary publications, we collect 74 primary papers closely related to the analysis of Android TPLs.
- *A comprehensive taxonomy.* We design a taxonomy of TPL-related research studies from different perspectives, including research objectives, targeted libraries,

type of TPLs and type of program analysis. Based on the taxonomy, we conduct an in-depth comparative study on the existing research.
- *Useful insights.* We identify the advantages and disadvantages, trends, patterns, gaps via comparative analysis of the collected papers. We also concluded the essential findings of existing studies and proposed possible implications of new research directions.

We hope these contributions can give readers a clear overview of third-party library related research and inspire future researchers to go beyond the current status quo by presenting more useful work. Besides, we also have the following essential findings:

- Most existing tools usually focus on Java TPLs analyses, only a few studies focus on the native library analyses. Future research can pay attention to this direction (see Section 3).
- For TPL identification, we still have a long way to go. Most TPL detection tools have high precision but low recall. Most tools cannot achieve a good resiliency to code obfuscation, especially dead code removal and package flattening. Even though there are many TPL identification, they still cannot achieve a good performance in version identification, partial imported TPL identification and optimization, and so on (see Section 4.1).
- We find existing studies present limited understanding about vulnerable TPLs. Only several types TPL vulnerabilities were studied. We suggest future researchers can pay more attention to how to reveal the entire landscape of TPL vulnerabilities (see Section 4.2).
- Future research can focus on TPL recommendation, GUI-related TPL smell analysis, TPL updating system design, native libraries related research, library compatibility analysis, TPLs' dynamic features analysis, cross-language TPLs analysis (see Sections 4.3, 4.4, and 5.3).

The rest of the paper is organized as follows. Section 6 introduces the related work. Section 2 introduce our literature search methodology. Section 3 provides the taxonomy on related research of TPLs. Section 4 introduces state-of-the-art research work based on our the research objectives. Section 5 gives the implications for readers based on our investigation. Section 7 gives a conclusion of our work.

2 LITERATURE SEARCH METHODOLOGY

Since we investigate the state-of-the-art research of third-party libraries in Android, we first introduce the methodology we use to find relevant literature and then present the basic information about our collected papers.

We follow the well-established guidelines [44], [45] to conduct our lightweight Systematic Literature Review (SLR). The overview of the SLR methodology we applied in this work is as below:

- Define the research scope. This step is used to set up our research scope and clarify our research goal.
- Second, identify the keywords for searching.
- Conduct the search process. The search process consists of two parts: search on commonly-used

TABLE 1
Keywords for Paper Repository Search

Group 1	Group 2	Group 3
android	third-party	lib*
mobile	open-source	component
*phone	ad*	software
APP	reuse	dependenc*
-		

"*" refers to wildcard.

Fig. 1. Word cloud of all major venue names of the examined publications.

publication repositories; and search on major venues, including both conferences and journals.

- Apply exclusion criteria on the search outcomes to enhance the analysis accuracy. In the process of keyword-based searching, it is inevitable to acquire some less related papers. To mitigate this, we evaluate the search results against the exclusive criterion defined in Section 2.2.
- Conduct a backward-snowballing on the remaining papers in case of omission.
- Merge the results.

2.1 Search Strategy

Search Scope. We first define our research scope: The papers should be related to third-party libraries in the Android ecosystem.

Search Keywords. We define the search keywords which are applied to find potentially related papers within the search scope. The search keywords construction is an iterative process. Based on our research scope, Android TPL-related research, we first list some search keywords and find out their synonymous words. During the search process, we continuously refine and extend our search keywords. Finally, we set up three groups of keywords shown in the Table 1. The first group of keywords limits our research scope. The second group of keywords consists of the modifiers of TPLs. The third group of keywords represents our research target; all of them are synonyms for third-party libraries. To ensure the collected papers are as complete as possible, we include "-" in the first group because we find some papers' titles do not include the platform information, but they are related to Android TPL-related research, such as OSSPoLICE [25]. The search terms are composed of two groups of keywords arranged in sequence (i.e., $search\ term = g1 \bigcap g2 \bigcap g3$, where $g1 \in group1$, $g2 \in group2$, $g3 \in group3$). The purpose of this step is to collect as many related papers as possible.

Search Database. 1) Repository. We first look for the potentially related papers in four well-known digital databases: ACM Digital Library [46], IEEE Xplore Digital Library [47], SpringerLink [48], and ScienceDirect [49]. Besides, we employ Google Scholar as our auxiliary tool to complement our collection of papers. Since the search results of the online repositories usually include some irrelevant publications, we set some rules to delete the noises. The specific solutions will be introduced in Section 2.2 (exclusion criteria).

2) Major Venue. Some conferences and journals, such as NDSS [50], have policies (e.g., open proceedings) that cause their publications unavailable in the aforementioned digital repositories [51]. To guarantee the completeness, we further

supplement the well-known venues[1] in our search database to avoid the repository search do not miss any major publications. For this SLR, we select 20 top venues from software systems[2] that are from the field of software engineering and programming languages and other top 20 venues from security and privacy [3] but we do not consider venues in the fundamental cryptography field (e.g., EUROCRYPT, CRYPTO, ASIACRYPT, FC, TCC). Fig. 1 provides a word cloud of the selected venues during the literature search process.

Search Process. For the repository search, we use our search string that is formed as a conjunction of the three groups of the keywords to search the potential related work from the four well-known digital repositories one by one. To consolidate the collected list of relevant papers, we manually checked the searching results by going through the titles and abstracts of these publications to ensure that they are related to Android third-party libraries. For the top venues search, we conduct our search process on DBLP.[4] Because the DBLP only provides the papers' titles, for some papers we cannot determine their content from the titles; we will exploit the Google Scholar to find these papers and check their abstracts.

2.2 Exclusion Criteria

In the process of keyword-based searching, it is inevitable to acquire some less related papers, e.g., repeated work, or even totally unrelated papers. The coarse-granularity search process usually includes some irrelevant papers. For example, the state-of-the-art repository search engines (e.g., the one provided by Springer) may include many irrelevant papers [51], [52] because SpringerLink allows collecting information on the first 1,000 items from its search results. To mitigate it and obtain more reliable results, we define a set of exclusion criteria as follows to filter out the unrelated or less related publications.

1) Non-English papers are filtered out.
2) Papers that are irrelevant to Android platform are excluded. As shown in Table 1, in order to acquire as many related papers as possible, the search string may not include the term "Android" because we include the "-" in the group 1. Consequently, a large

1. https://scholar.google.lu/citations?view_op=top_venues&hl=en&vq=eng
2. https://scholar.google.lu/citations?view_op=top_venues&hl=en&vq=eng_softwaresystems
3. https://scholar.google.lu/citations?view_op=top_venues&hl=en&vq=eng_computersecuritycryptography
4. https://dblp.uni-trier.de/

TABLE 2
Summary of the Selection of Primary Publications

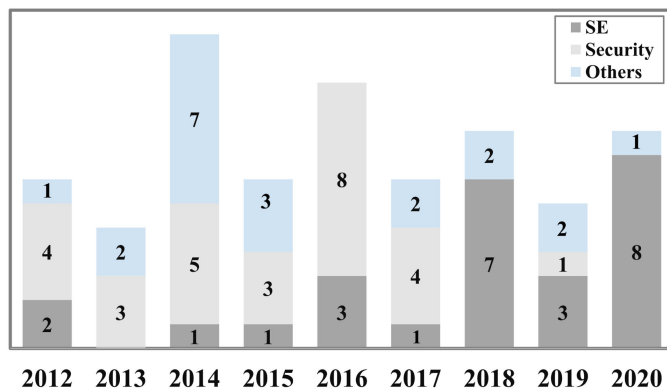| Search Process	#Related paper
Repository and Major Venus Search	5, 953
After reading the titles/abstracts	95
After filtering out irrelevant topics	82
After selecting extensive papers	79
After deleting duplicate papers	76
Final selection	**74**

Fig. 2. Distribution of the repository of literature from 2012 to 2020 based on venue differences (SE: Software engineering; Others: mainly include fields from networking and programming language).

number of publications on other platforms, e.g., iOS and Windows, are included in the paper set, which should be filtered out. Besides, we also delete some papers from other ecosystems (e.g., npm) because the extracted features or identification methods are different from Android libraries.

3) Delete papers are not related to third-party libraries. Based on our search keywords, we may find some studies are related to third-party markets, third-party Android phones. Our research scope just focuses on Android TPLs; therefore, we also exclude these papers.

4) This paper just focuses on extensive works. We have noticed that some papers, such as the short papers, posters, are for work of promising ideas in a preliminary stage. For this paper, if we have found the full formats in the later publications, we will delete the preliminary versions [51], [53]. In accordance with international academic rules [44], [45], publications that are less than five pages in IEEE/ACM-like double-column format or less than seven pages in LNCS-like single-column format should be treated as short papers. According to this item, we delete three papers [4], [40], [54] from our dataset. For the first paper, we found the extensive version is in Chinese [55]. Based on the first item, we also do not consider this paper. For the remaining two papers, MadLens [54] and LibD [40] that first was published in INFOCOM (2018) and ICSE (2017), respectively, then were extend to the journal papers and published in TMC (MAdLens, 2019) [56] and TSE (LibD, 2018) [57]. For these papers, we keep the extended versions. We delete the posters if they just propose some basic ideas.

5) Duplicated papers are removed. Some papers may have a preprint version online, the title or some contents of the published one and the preprint may have some differences. For these papers, if they have the same author list and similar or the same title, we consider they are suspicious pairs. Then we manually check whether their contents share a lot of content or not. If yes, we will keep the recent versions. If a paper published in a conference venue and was extended to a journal venue. We will remain the extensive publications.

2.3 Backward Snowballing

To avoid the omission of related papers, we perform a backward-snowballing on the collected papers. Specifically, we manually check the references of each paper and find potentially related papers that are not in our paper repository or papers that cannot be found by using the defined keywords. We finally add two research papers [25], [58] into our paper repository, whose titles do not contain our predefined keywords.

2.4 Statistics of Selected Publications

Table 2 concludes the statistics of literature during the collection process. We obtained 5,953 papers in total by searching from the aforementioned database and venues. After reading the titles and abstracts, we delete the irrelevant papers, and the number of papers meeting the requirement decreases to 95. The discard rate is very high here primarily due to two reasons: the first one is that SpringerLink may include many false positives by using our search string [51]; the second reasons is due to the predefined search keywords. To ensure the completeness, our search strings contain "lib*", "ad*", "open-source software" that may include some papers from other platforms except Android. Thus, we get more than 5900 irrelevant papers during the search process. The whole selection process is performed by the first author and other co-authors also help to conduct the cross-validation. We then go through the remaining papers by reading their abstract, introduction, and conclusion and finally select 74 research papers, which are the main research subjects for the following analysis.

Fig. 2 shows the distribution of the collected literature through the published year (2012-2020). The period from 2012 to 2016 witnessed a fluctuation in the number of publications about Android third-party libraries. The number of publications reached the peak in 2014 (11). The papers in security and software engineering reached their peak in 2016 and 2018, respectively. According to Fig. 2, we can find the early work most published in networking and reached the peak in 2014. The research on the security-related issues of Android third-party libraries basically remained stable from 2012 to 2015 and increased abruptly in 2016, reached eight papers. And then the following years from 2017 to 2020 witnessed its gradual decline. Since 2018, an increasing number of researchers have focused on software engineering. The recent three years have witnessed a sharp increase. Table 3 enumerates all the 74 papers and the corresponding publication year, venues, and their tool name if any. As for some work without a tool name, we use the first author name to denote them.

TABLE 3
Primary Information of Our Paper Repository

Tool/System Ref.	Year	Venue
Wang et al. [32]	2020	ICSME
LibDetect Analysis [59]	2020	ASE
LibHarmo [31]	2020	ESEC/FSE
LibDX [60]	2020	SANER
LibExtractor [61]	2020	WiSec
LibRoad [62]	2020	Mobile Computing (J)
Ahasanuzzaman et al. [63]	2020	TSE(J)
MadDroid [22]	2020	WWW
Ahasanuzzaman et al. [64]	2020	EMSE
MAdLens [56]	2019	TMC (J)
Yasumatsu et al. [24]	2019	CODASPY
LibID [42]	2019	ISSTA
RIDDLE [30]	2019	ICSE
MadLife [65]	2019	WWW
Salza et al. [66]	2019	Spring Science
APPCOMMUNE [67]	2019	SANER
DECCA [29]	2018	ESEC/FSE
FraudDroid [19]	2018	ESEC/FSE
LibPecker [68]	2018	SANER
ORLIS [43]	2018	MOBILESoft
Salza et al. [69]	2018	ICPC
LibD2 [57]	2018	TSE(J)
Dong et al. [70]	2018	HotMobile
Han et al. [71]	2018	WPC(J)
Ogawa et al. [72]	2018	CANDARW
FLEXDROID [58]	2017	NDSS
Droid-V [73]	2017	MSR
OSSPOLICE [25]	2017	CCS
AppLibRec [74]	2017	Internetware
Derr et al. [75]	2017	CCS
Zhan et al. [76]	2017	ACISP
Gui et al. [77]	2017	CoRR
Son et al. [78]	2016	NDSS
LibCage [79]	2016	ESORICS
LibFinder [80]	2016	S&P
LibRadar [41]	2016	ICSE-C
LibScout [23]	2016	CCS
LibSift [81]	2016	APSEC
Pluto [9]	2016	NDSS
Li et al. [34]	2016	SANER
Ruiz et al. [82]	2016	IEEE Software(J)
Rastogi et al. [83]	2016	NDSS
Wei et al. [84]	2016	NDSS
Madscope [85]	2015	MobiSys
PEDAL [16]	2015	MobiSys
Book et al. [86]	2015	Computer Science(J)
Paturi et al. [87]	2015	NDSS
Gui et al. [88]	2015	ICSE
ClickDroid [89]	2015	ARES
Kühnel et al. [90]	2015	Trustcom
AdDetect [91]	2014	ISSNIP
APKLancet [92]	2014	ASIACCS
COMPAC [93]	2014	CODASPY
DECAF [20]	2014	NSDI
Duet [94]	2014	WiSec
Madfraud [21]	2014	MobiSys
NativeGuard [95]	2014	Wisec
Moonsamy et al. [7]	2014	ISITA
Short et al. [8]	2014	MASS
Ullah et al. [96]	2014	INFOCOM WKSHPS
Ruiz et al. [97]	2014	IEEE Software(J)
Brahmastra [98]	2014	USENIX Security
AFrame [15]	2013	ACSAC
SanAdBox [14]	2013	ICC
Book et al. [99]	2013	SPSM
Book et al. [100]	2013	MoST
Tongaonkar et al. [101]	2013	PAM
AdDroid [18]	2012	ASIACCS
AdRisk [28]	2012	WiSec
AdSplit [17]	2012	USENIX Security
Bauer et al. [102]	2012	ICSM
Leontiadis et al. [103]	2012	HotMobile
Stevens et al. [104]	2012	MoST
Vallina-Rodriguez et al. [105]	2012	IMC

Note that: if a tool is not given the name, we use the first author's name to represent it.

3 TAXONOMY OF EXAMINED PUBLICATIONS

To define a taxonomy for existing Android TPL-related research, we first tried to choose appropriate dimensions and properties in existing surveys [53], [106]. We hope the taxonomy can help characterize existing work and gain insights into the state-of-the-art research as well as assess different techniques. Fig. 3 shows a high-level view of the taxonomy diagram unfolding in four dimensions (i.e., *Research Objectives, Targeted Libraries, Type of TPLs* and *Type of Program Analysis*). We give a detailed introduction about each dimension and sub-dimension in the following subsections.

3.1 Research Objectives

This dimension categorizes existing research with respect to the purposes of their analysis. Different studies have different problems they want to solve, we enumerate five subdimensions in this category: 1) TPL detection, 2) TPL security-related issue analysis, 3) TPL privilege de-escalation, 4) TPL maintenance, and 5) TPL attribute understanding. We now explain each of them as follows.

TPL detection aims to find the TPLs used in Android apps. TPL dependency information is not transparent, not to mention there are many direct and transitive dependencies. On the other hand, TPL detection has many significant application prospects in assisting the downstream tasks, such as malicious app detection, repackaged Android app detection, vulnerable in-app TPL detection, software composition analysis. We also find that TPL detection is becoming a hot topic in recent years; therefore, it is necessary to understand existing TPL detection techniques.

TPL security issue analysis accounts for the largest proportion (31%, 23/74) in collected papers. On the one hand, we can see that many researchers were committed to the research on TPL security; on the other hand, TPLs do exist many security issues. Understanding current research status and their risk to users and devices is utterly necessary. For this dimension, we mainly discuss the following five parts: 1) privacy leakage detection and analysis, 2) vulnerability identification, 3) malicious TPL detection and analysis, 4) ad frauds. Based on existing research, we can find privacy leakage is not uncommon in TPLs [7], [8], [84], [107], [108]. For example, some ad libraries can collect user's demographics information, but these TPLs may leak users' privacy without notice. Aiming at this phenomenon, existing research investigated how many TPLs can cause privacy leakage[84], how was the data leaked [9] so on and so forth. Vulnerability identification involves in-app TPL vulnerability detection. Malicious TPL detection involves finding TPLs with malicious behaviors, such as dynamically loading malicious payloads, leading to revenue loss. Note that malicious TPL detection does not necessarily need to identify the specific TPLs. Most research just needs to find the parts that belong to a TPL and identify the malicious behaviors. That is the difference between TPL identification and malicious TPL detection. Ad frauds mean that unscrupulous developers violate the ad developing rules by placing ads that close to or cover the UIs, which may affect the user experience and induce extra clicks or impressions [19].
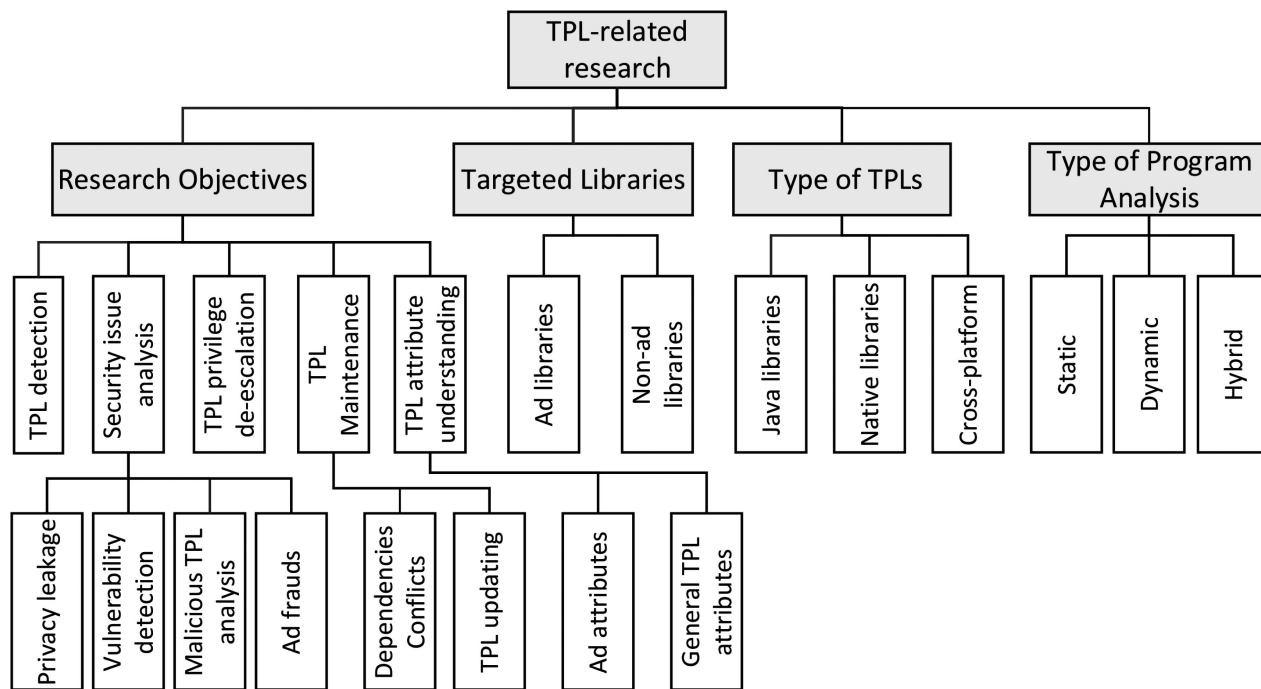
Fig. 3. Taxonomy of TPL-related research.

TPL privilege de-escalation is responsible for separating the privilege of TPLs from the host app. Therefore, we also can call it TPL isolation. Android system allows apps to access the system resources with corresponding privileges, but the permission mechanism is working at the app-level, which means the in-app TPLs share the same permissions with host app [12], [13]. We can find this permission mechanism can bring potential risks because it causes the TPL over-privileged. TPL isolation usually aims to separate TPLs from host apps by allocating them different storage space, permissions, and process IDs to ensure the TPLs cannot use the permissions of the host app to conduct sensitive behaviors.

TPL maintenance plays an essential role in app development, which can help keep the quality and health of apps. As for the TPL maintenance, we mainly introduce the research on dependency conflicts and TPL updating. Dependency conflicts are mainly due to the considerable direct and transitive dependencies within TPLs. Dependency conflict occurs when the loaded version cannot cover the features required by the app, leading to runtime exceptions. TPL updating usually involves vulnerabilities and compatibility issues. For instance, the old version of TPL may be detected within a known vulnerability and the new version has fixed the vulnerability. We can investigate app developers' responses to the updating TPLs. Existing TPL updating research primarily can be divided into two parts, namely the TPL automated updating tool implementation and TPL updating analysis.

TPL attribute understanding includes miscellaneous aspects of TPLs, such as the library recommendation, the impact of TPLs' new features on apps, rating analysis, permission analysis and so on. For this dimension, we mainly describe these studies from two aspects: general TPL-related attributes, and attributes related to the special TPLs, ad libraries. Besides, we find most of research in this dimension usually adopts empirical study, case study and user study as the evaluation methods.

In particular, some articles may be involved in multiple categories, and each category is not entirely independent for this dimension. Therefore, there are intersections among different categories. For example, PEDAL [16] attempts to implement privilege de-escalation for ad libraries (TPL isolation). However, it also needs to identify TPLs first. Thus, PEDAL also implements a tool named *Separator* to identify ad libraries (TPL detection).

3.2 Targeted Libraries

This dimension classifies collected papers based on the ad and non-ad libraries. The non-ad libraries mean not only for ad libraries but also for general TPLs. Table 4 characterize the publications selected from our SLR in terms of the research objectives based on the ad and non-ad libraries. Among our collected papers, we find about a half (35/74) of the collected papers focus on ad libraries. One of the main reasons is that developers can make profits by embedding ad libraries in their apps. The ad library is an essential type of TPLs, which bridges the advertisers, developers, and customers. Besides, ad libraries usually need to get users' information and push customized content to target users. The ad libraries are more likely to collect users' privacy information without users' attention and lead to privacy leakage [104]. Such a category can help us understand existing research on ad and non-ad libraries and figure out the current research gap.

Fig. 4 is a bulb graph that illustrates the number of existing research publications related to ad/non-ad libraries. As can be seen from Fig. 4, ad libraries are the main target for both security-related research and attribute analysis. For security and privacy analysis (third column), we find that ad libraries became the primary targets for many adversaries, which account for 70% (16/23) of the whole research on security. This is because malicious developers can redirect the ad account information to gain illegal revenues, and thus many adversaries conduct various attacks on ad

TABLE 4
The Categorization of Collected Papers Based on the ad and Non-ad Libraries

Tool	Ad	All TPLs	Tool	Ad	All TPLs	Tool	Ad	All TPLs
LibDX [60]		✓	AdRisk [28]	✓		LibHarmo [31]		✓
LibExtractor [61]		✓	MadDroid [22]	✓		DECCA [29]		✓
LibRoad [62]		✓	MadLife [65]	✓		RIDDLE [30]		✓
LibID [42]		✓	Rastogi et al. [83]	✓		Wang et al. [32]		✓
LibPecker [68]		✓	LibFinder [80]		✓	Yasumatsu et al. [24]		✓
ORLIS [43]		✓	Kühnel et al. [90]	✓		APPCOMMUNE [67]		✓
Han et al. [71]		✓	APKLancet [92]		✓	Salza et al. [66]		✓
OSSPoLICE [25]		✓	Duet [94]		✓	Salza et al. [69]		✓
LibD [57]		✓	Madfraud [21]	✓		Ogawa et al. [72]		✓
LibScout [23]		✓	DECAF [20]	✓		Derr et al. [75]		✓
LibRadar [41]		✓	Dong et al. [70]	✓		Ahasanuzzaman et al. [63]	✓	
LibSift [81]		✓	FraudDroid [19]	✓		Ahasanuzzaman et al. [64]	✓	
PEDAL [16]	✓		Zhan et al. [76]		✓	MAdLens [56]	✓	
AdDetect [91]	✓		FLEXDROID [58]		✓	Gui et al. [77]	✓	
Wei et al. [84]	✓		LibCage [79]		✓	Ullah et al. [96]	✓	
Son et al. [78]	✓		ClickDroid [16]	✓		Madscope [85]	✓	
Pluto [9]	✓		NativeGuard [95]			Book et al. [99]	✓	
Paturi et al. [87]		✓	COMPAC [93]		✓	Tongaonkar et al. [101]	✓	
Moonsamy et al. [7]	✓		AFrame [15]		✓	Book et al. [100]	✓	
Short et al. [8]		✓	SanAdBox [14]	✓		Vallina-Rodriguez et al. [105]	✓	
Leontiadis et al. [103]	✓		AdDroid [18]	✓		Gui et al. [88]	✓	
Stevens et al. [104]	✓		AdSplit [17]	✓		Ruiz et al. [97]	✓	
Droid-V [73]		✓	Ruiz et al. [82]	✓		Li et al. [34]		✓
Bauer et al. [102]		✓	Zhan et al. [59]		✓	Book et al. [86]	✓	
Salza et al. [66]		✓	Brahmastra [98]		✓			

libraries. Besides, ad libraries usually can collect targeted information from users and push the corresponding content to users base on the collected information [7], [8], [21], [54]. Therefore, that also attracts many researchers to focus on these problems. As for the TPL attribute analysis, ad libraries have some typical characteristics, such as the GUI and pushed content. Many interesting points still deserve to dig deep. Compared with other types of TPLs, UI design of ad libraries can affect users' experience [56], which can directly affect the rating of an app. Existing research analyzed the traffic consumption [101], display frequency, timing and location [77], the effects on apps, display contents [22]. In contrast, we can find that research w.r.t library detection and maintenance usually tend to not distinguish the types of TPLs; that can make sense by considering their research goals. It is more valuable to analyze all types of TPLs for these studies.

3.3 Type of TPLs

This dimension classifies existing research based on different type of TPLs. Android TPLs can be realized by different languages, such as the most widely used third-party libraries implemented by Java or Kotlin [109], native libraries implemented with C/C++, some libraries with respect to GUI plugins implemented by multiple languages (e.g., Java and JavaScript). A specific languages often have their own unique characteristics. For instance, the reflection mechanism is not available in C language. The extracted CFGs from TPLs developed by C language before and after optimization are quite different. Java has package managers, it can easily import the third-party dependencies but C/C++ has not such a package manager. For different languages, detection techniques and analysis approaches are usually different. For native libraries, researchers usually adopt the binary code to conduct the analysis and usually choose the string constants as the feature. For java library analysis, researchers usually change the Java bytecode into a appropriate intermediate representation(IR) to conduct the subsequent analysis. Thus, some approaches designed for one type of TPLs could not support other type of libraries. Based on this dimension, we can understand current research trend and think about the future research direction.

3.4 Type of Program Analysis

This dimension classifies existing TPL-related studies based on the type of program analysis. The type of program analysis employed in TPL-related research could be static and dynamic. Static analysis can help us understand some TPL structure and code features, which is usually used in TPL identification and security analysis. Dynamic analysis can detect the runtime behaviors, which can capture some features that static analysis cannot, such as the dynamic loaded malicious code, managing some runtime privileges. The
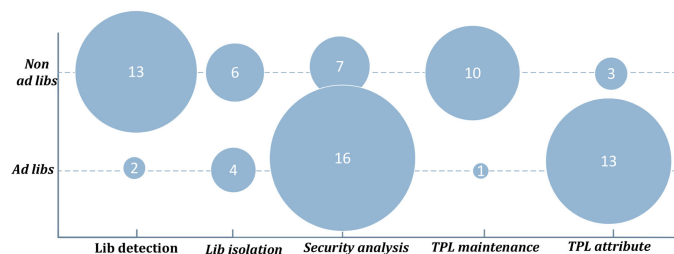
Fig. 4. Distribution of ad/non-ad related-papers from our collected paper repository (note that the total number is 75 because PEDAL belongs to both TPL detection and Lib isolation).

TABLE 5
The Categorization of Collected Papers Based on the Type of Program Analysis

Tool	S	D	H	Tool	S	D	H	Tool	S	D	H
LibDX [60]	✓			AdRisk [28]	✓			LibHarmo [31]	✓		
LibExtractor [61]	✓			MadDroid [22]		✓		DECCA [29]	✓		
LibRoad [62]	✓			MadLife [65]		✓		RIDDLE [30]	✓		
LibID [42]	✓			Rastogi et al. [83]			✓	Wang et al. [32]	-	-	-
LibPecker [68]	✓			LibFinder [80]	✓			Yasumatsu et al. [24]	✓		
ORLIS [43]	✓			Kühnel et al. [90]	✓			APPCOMMUNE [67]			✓
Han et al. [71]	✓			APKLancet [92]	✓			Salza et al. [66]	-	-	-
OSSPoLICE [25]	✓			Duet [94]	✓			Salza et al. [69]	✓		
LibD [57]	✓			Madfraud [21]		✓		Ogawa et al. [72]		✓	
LibScout [23]	✓			DECAF [20]		✓		Derr et al. [75]	-	-	-
LibRadar [41]	✓			Dong et al. [70]		✓		Ahasanuzzaman et al. [63]	✓		
LibSift [81]	✓			FraudDroid [19]		✓		Ahasanuzzaman et al. [64]	✓		
PEDAL [16]			✓	Zhan et al. [76]	✓			MAdLens [56]	✓		
AdDetect [91]	✓			FLEXDROID [58]				Gui et al. [77]	-	-	-
Wei et al. [84]	-	-	-	LibCage [79]		✓		Ullah et al. [96]		✓	
Son et al. [78]		✓		ClickDroid [89]		✓		Madscope [85]		✓	
Pluto [9]			✓	NativeGuard [95]			✓	Book et al. [99]	✓		
Paturi et al. [87]			✓	COMPAC [93]	✓			Tongaonkar et al. [101]		✓	
Moonsamy et al. [7]			✓	AFrame [15]	✓			Book et al. [100]	✓		
Short et al. [8]		✓		SanAdBox [14]		✓		Vallina-Rodriguez et al. [105]	✓		
Leontiadis et al. [103]		✓		AdDroid [18]			✓	Gui et al. [88]	-	-	-
Stevens et al. [104]			✓	AdSplit [17]			✓	Ruiz et al. [97]	-	-	-
Droid-V [73]	✓			Ruiz et al. [82]	✓			Li et al. [34]	✓		
Bauer et al. [102]	✓			Zhan et al. [59]	-	-	-	Book et al. [86]	-	-	-
Salza et al. [66]	-	-	-	Brahmastra [98]			✓				

S: static analysis, D: dynamic analysis, H: hybrid analysis (static analysis & dynamic analysis), - "not applicable".

two analysis approaches are complementary to each other. Static analysis is usually more effective, and dynamic can capture the dynamic interaction behaviors and more accurate. For this categorization, we analyze existing research from three sub-dimensions: i.e., static analysis, dynamic analysis, and hybrid analysis.

Table 5 presents the categorization of collected papers based on the type of program analysis. Based on the Table 5, we can find some patterns. For example, 1) all TPL detection tools adopt static analysis methods to identify the in-app TPLs. There are two main reasons: the first one is that it is difficult to capture some dynamic features of some TPLs; the code coverage rate of the dynamic analysis is also limited, which may lead to many false negatives. The second reason is that it is also difficult to find the exact version of TPLs by using dynamic analysis. 2) All ad fraud detections use dynamic analysis. 3) For TPL attributes analysis, researchers usually employ empirical study, case study, and user study. The research approaches also adopt interview and statistical analysis. Thus, the program analysis method is not suitable for most of these studies. 4) When it comes to extracting graphical interfaces, UI states, traffic features, it usually needs to adopt the dynamic analysis, such as MadDroid [22], Ullah et al. [96], and MadLife [65]. 5) Most privacy leakage detections and malicious TPL detections usually use dynamic analysis or hybrid analysis.

3.5 Summary

Based on the taxonomy, we calculate the distribution of collected papers in each dimension. The specific results can be seen from Table 6. For research objectives, most of the research focused on security issues analysis on TPLs, which

accounted for 31% of the total collected papers. The proportion of TPL maintenance is the least. TPL detection plays an essential role for downstream tasks; many research such as the vulnerable TPL identification, TPL isolation may involve in TPL detection. Thus, the total number of papers in each category is greater than the total number of collected papers. For dimension with respect to targeted libraries, we can find that nearly half of the research is focused on the ad libraries. For the dimension regarding the type of TPLs, we can find that existing studies usually concentrate on Java libraries; the native libraries and other libraries that are written by other languages are seldom thoroughly explored.

TABLE 6
The Distribution of Collected Papers in Different Dimensions

Dimension	Sub-dimension	#	%
Research Objectives	TPL detection	15	20%
	Security issue analysis	23	31%
	TPL privilege de-escalation	10	14%
	TPL maintenance	11	15%
	TPL attribute understanding	16	22%
Targeted Libraries	Ad libraries	35	47%
	No-ad libraries	39	53%
Type of TPLs	Java libraries	71	96%
	Native libraries	2	3%
	cross-paltform languages	1	1%
Type of program analysis	Static analysis	36	49%
	Dynamic analysis	17	23%
	Hybrid analysis	10	13%

#: the number of corresponding papers, %: the percentage of the selected papers.

TABLE 7
A Summary of Lib Detection Techniques

Function	Tool/First Author	Year
Lib Detection	LibDX [60]	2020
	LibExtractor [61]	2020
	LibRoad [62]	2020
	LibID [42]	2019
	LibPecker [68]	2018
	ORLIS [43]	2018
	Han *et al.* [71]	2018
	OSSPoLICE [25]	2017
	LibD [40], [57]	2017
	LibScout [23]	2016
	LibRadar [41]	2016
	LibSift [81]	2016
Ad Detection + Isolation	PEDAL [16]	2015
Ad Detection	AdDetect [91]	2014

We only find two papers can handle native libraries, i.e., OSSPOLICE [25] and NativeGuard [95]. OSSPOLICE is a TPL detection tool that can identify Java and native libraries. NativeGuard is the first work that isolates the native libraries from the host app. Only one paper can solve the cross-language TPLs, i.e., LibDX [60]. We encourage future researchers to fill this gap by exploring more of these TPLs instead of only focus on Java libraries. As for the method of program analysis, most of the studies adopt static analysis. Many open challenges such as dynamic loading and reflection still have not well-studied.

4 REVIEW OF TPL RESEARCH

In this part, we discuss different research on Android TPLs in detail based on the research objectives. We use this categorization to organize the following content due to two reasons: 1) this categorization can completely cover our collected papers, 2) we can compare the advantages and disadvantages of existing methods because they target the same objectives.

4.1 TPL Detection

In this section, we introduce the research background of TPL detection and provide a brief description of current research. In particular, we provide a taxonomy of these state-of-the-art techniques from three different perspectives. We also summarize obfuscation-resilient capability and discuss the defects of existing tools.

4.1.1 Research Background

Research such as repackaged app detection and mobile malware detection needs to first identify third-party libraries as these TPLs could be the noises of the host code during the detection process, which would decrease the accuracy of the results. License violation and TPL vulnerability detection require the specific in-app versions. Prior research [16] has shown that about 57% of apps contain ad libraries. Wang *et al.* [4] also pointed out that on average, more than 60% of the code in an Android app belongs to TPLs. CLAN-Droid [110] showed that TPLs could affect the detection

accuracy. Li *et al.* [34] explained why TPLs could affect the detection results and give motivating examples. As we can see, TPL detection has essential functions for downstream tasks. Zhan *et al.* [33] summarized the method of different repackaging systems on how to filter out TPLs. They found that most repackaging detection techniques [35], [36], [111], [112], [113], [114], [115], [116] exploit whitelist-based method to filter out TPLs, while Wukong [117] and PiggyApp [118] use clustering-based method. Besides, other research such as TPL isolation also needs to identify in-app TPLs first [16]. Detecting TPLs in Android apps is evidently an essential task for many downstream research tasks.

In the beginning, most research uses the whitelist-based method to remove TPLs because it is relatively easy to conduct. However, the whitelist-based method is not very reliable, which has many inevitable shortcomings: *1) It is hard to maintain a complete list of libraries*. Existing studies, such as ViewDroid [113], MassVet [35] only choose commonly-used TPLs as their whitelists to identify TPLs. Obviously, it is inevitable for this method to miss some TPLs. *2) Such a method cannot discover new TPLs*. Relying only on the collected whitelist to identify TPLs, this method fails to identify newly-emerged TPLs that are not included in the list. *3) Such a method depends on the package name of TPLs, which is not resilient to code obfuscation such as package renaming and package flattening*. Given the limitations of the whitelist-based method, many researchers start to explore more effective methods to detect TPLs, leading to the emerging of independent TPL detection studies. These approaches pay more attention to the features of TPL themselves and attempt to extract unique features from TPLs, such as the code semantic features, UI features, or string features, to identify TPLs. TPLs can be identified with the help of the machine learning-based methods or similarity comparison algorithms.

4.1.2 Existing Research

We collected 14 research work on TPL detection, as shown in Table 7. In previous research [59], we have conducted a comprehensive comparison on 11 of the 14 studies from the perspective of practical usage and implementation performance. We also published a benchmark that can be used to evaluate TPL detection tools of Android. In this paper, we conduct a systematic survey on different TPL detection tools and also make a supplement to previous work.

As shown in Table 7, the research on TPL detection originated in 2014. Both *PEDAL* and *AdDetect* are ad library identification tools that cannot specify other types of TPLs. Both of them first extract ad library-related features (e.g., APIs, permissions, UIs, traffic features) to construct the feature vectors and then use a binary classifier to distinguish ad libraries and non-ad libraries. *LibSift* is another special tool that can only separate different TPL candidates from host apps but cannot identify specific TPLs either. Hence, the real specific TPL identification research started from LibRadar (2016).

LibRadar is a component of Wukong [117], a repackaged app detection tool. Wukong leverages LibRadar to find in-app TPLs and filter them out because these in-app TPLs could be noises that affect the detection accuracy. The

intuition behind LibRadar is that TPLs are widely-used by many apps. Thus, LibRadar can find these TPLs by clustering them without prior knowledge. LibRadar constructs the TPL candidate instances based on the package hierarchy structures. And then, it extracts the Android APIs of each instance as the code features. These TPLs would be clustered into big groups because the same libraries are used by many different apps [117]. Similar to LibRadar, *LibD* and *LibExtractor* also adopt the clustering-based method to construct the TPL feature database. LibD pointed out that the method of LibRadar in library instance construction could be problematic. Researchers of LibD find that the package structure of different versions of the same TPLs could be different; using the package structure to construct the TPL candidates could lead to mis-identification. Therefore, LibD proposes a new method to adopt the package inclusion and inheritance relationship (named "homogeny graph") as the module decoupling features to improve the construction of in-app TPL instances. LibExtractor uses six class dependency relations to construct the in-app TPL instances and encode the class dependency into the code features. LibExtractors also adopts a clustering algorithm to identify TPL components of large-scale input apps and then identifies malicious libraries. Both *Han et al.* [71] and LibD proposed to adopt the opcode of basic block in control flow graph (CFG) as the code feature of TPLs, but unlike LibD, Han *et al.* [71] is a similarity comparison-based tool that first collects the TPL files in advance and then adopts the similarity comparison method to identify in-app TPLs.

LibScout is the first tool that claims it can pinpoint the precise library versions used in apps. By leveraging LibScout, researchers can detect known vulnerabilities in TPLs that are still used by apps. *ORLIS* and *LibScout* use the same code feature (fuzzy method signature, the specific definition can refer to the Section 4.1.4 "*Extracted features*") but different identification methods to identify in-app TPLs. *OSSPoLICE* can detect the potential software license violations and the known vulnerabilities of in-app TPL versions. OSSPoLICE first uses the string constants and fuzzy method signature as the first stage code feature to identify the potential TPLs. Among these potential TPLs, it uses the function centroid as the fine-grained feature to identify the specific TPL version.

Researchers of *LibPecker* found that tools such as LibScout and ORLIS use a relaxed TPL profile (i.e., the code feature granularity is too coarse), leading to more false negatives. Hence, LibPecker attempts to improve the performance of TPL identification by adopting the internal class dependencies inside a TPL to generate more strict TPL signatures. Meanwhile, it introduces the adaptive class similarity threshold and weight class similarity score when calculating the TPL similarity to ensure better precision and recall. *LibID* is another library version identification tool by formulating the TPL identification problem into a binary integer programming models. LibID uses more semantic information that includes CFG and class dependency in feature extraction to improve the resiliency to code obfuscation.

To improve the detection efficiency, *LibRoad* adopts a combo strategy to identify in-app TPLs. For non-obfuscated parts of TPLs, LibRoad adopts the package name-based matching policy to identify the in-app TPLs; while for
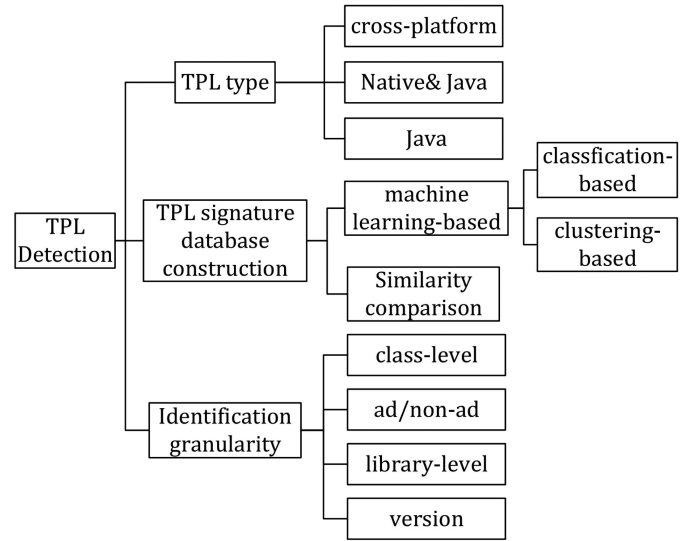

Fig. 5. The taxonomy of TPL detection tools.

obfuscated parts of TPLs, LibRoad adopts the signature-based matching policy by comparing the features of in-app TPLs with the features in the TPL database. *LibDX* is a cross-platform open-source software detection tool. Unlike other tools, LibDx directly extracts code features from the binaries. As can be seen that even if there are many TPL detection tools, they have different concerns and application scenarios and adopt different techniques in TPL detection. More details about each tool will be introduced in the following sections.

4.1.3 Taxonomy

We can classify these TPL detection tools from different perspectives. In this paper, we propose three classification schemes, as shown in Fig. 5. Based on the *detected type of TPLs*, we can further divide existing tools into three types: 1) cross-platform TPL detection tools, 2) native and Java library detection tools, and 3) Java library detection tools. LibDX is the only cross-platform TPL detection tool that can detect various types of TPLs and is not just limited to Android. The remaining tools are Android TPL detection tools and most of them can only identify Java TPLs. OSSPoLICE can detect both native code libraries (C/C++) and Java Libraries. Other tools only can detect Java libraries. Besides, three tools are special among these Java library detection tools, i.e., LibSift, AdDetect, and PEDAL. AdDetect and PEDAL only can distinguish the ad/non-ad libraries. LibSift only can identify the parts belonging to TPL modules, without reporting specific TPLs.

Based on the *method of TPL feature database construction*, we can classify existing tools into two categories: 1) machine learning-based method and 2) similarity comparison-based method. The machine learning-based method can be subdivided into two categories, the classification-based method, and the clustering-based method. AdDetect and PEDAL adopt the classification-based method to distinguish the ad/non-ad libraries. LibExtractor, LibD, and LibRadar are clustering-based tools. The intuition is that TPLs are used by many apps. Taking a considerable number

TABLE 8
The Comparison of Existing State-of-the-Art TPL Detection Tools

		LibExtractor	LibRoad	LibID	LibPecker	ORLIS	Han et al.	OSSPOLICE	LibD	LibScout	LibRadar	LibSift	PEDAL	AdDetect
Pre-processing Tool	Apktool		✓		✓				✓			✓	✓	✓
	Androguard			✓	✓		✓		✓					
	Soot					✓		✓			✓			
	baksmali	✓												
	LIBDEX										✓			
Module Decoupling Feature	Package Structure		✓	✓	✓			✓		✓	✓	✓		
	Homogeny Graph									✓				
	PDG											✓		✓
	Class Dependency	✓				✓								
Extracted Features	Fuzzy Method Sig.		✓	✓		✓		✓		✓				
	CFG			✓			✓		✓					
	APIs											✓	✓	✓
	Class Dependency	✓		✓	✓									
	CFG Centroid							✓						
	Permission,component,UI												✓	✓
	Strings							✓					✓	✓
Comparison Method	Similarity Comparison	✓					✓		✓		✓	-	-	-
	Fuzzy Class Match (Adaptive Match)		✓		✓							-	-	-
	Fuzzy Hash				✓							-	-	-
	Hierarchical Indexing							✓				-	-	-
	LSH		✓									-	-	-
Identification Granularity	Class-level					✓								
	Ad Library												✓	✓
	Library-level	✓	✓		✓				✓		✓			
	Version-level			✓				✓		✓				

PDG: package dependency graph; LSH: Locality-Sensitive Hashing; CFG: Control Flow Graph, '-': not applicable;
✓: represents the use of a tool, a feature or a technique; if a tool is not given the name, we use the first author's name to represent it.

of apps as input, the same TPLs will be clustered together. Therefore, clustering-based tools usually require millions of apps as input to ensure generating enough TPL signatures. Most existing tools are similarity comparison-based tools, which do not require a substantial number of apps as input but require developers to collect TPL files to construct the feature database. Developers use the same algorithm to generate the profile for in-app TPLs and feature database. By comparing the similarity value with the TPLs in the database, these tools can identify the in-app TPLs.

Based on the *identification granularity*, we divide current tools into four types, i.e., the class-level, ad/non-ad level, version-level, and library-level. ORLIS can report the classes that belong to TPLs. PEDAL and AdDetect only can distinguish the ad and non-ad libraries. The library-level identification means that the tool just can identify which TPL is used by apps but cannot determine the specific version. Version-level identification means that the tool not only can identify this TPL and also can specify the version. We give this classification based on whether the literature states it can detect version-level TPL or not. LibID, LibScout, and OSSPoLICE claim that they can identify the TPL at version-level. LibRoad, LibPecker, and [71] mainly report the TPLs are presented in apps. We can find that existing clustering-based tools only can identify TPL at the library-level. In fact, it is challenging for clustering-based tools to identify the TPL at version-level. The code similarity of different versions of the same TPLs is various. It is difficult for clustering algorithms to find perfect parameters to divide different

TPLs into different clusters. Each cluster may include a single version or multiple versions. Besides, the labeling process is also time-consuming, labor-intensive, and most importantly, error-prone.

4.1.4 State-of-the-Art Techniques

Basically, the TPL detection process can be summarized into four steps: 1) pre-processing, 2) library instance construction, 3) feature extraction, 4) library instance identification [59]. The pre-processing mainly decompiles the input app and transforms the bytecode into an appropriate intermediate representation to facilitate subsequent processing. The library instance construction mainly implements the module decoupling algorithm to find the boundaries of different TPLs and constructs the TPL instance candidates. Then TPL detection tools can extract different code features for TPL instance candidates to represent them, such as the graph, hash values, or feature vectors, etc. The last step is library identification by using the similarity comparison techniques to compare the in-app TPL features with the features in the database to identify the specific TPLs/versions.

In Section 4.1.2, we give a brief introduction about each TPL detection tool. To allow readers to better understand the existing TPL detection tools, we give a more detailed introduction about each tool by comparing their commonalities and differences based on the TPL detection process in this section, as shown in Table 8. LibDX is the only cross-platform TPL detection tool. LibDX extracts the read-only

DATA segment (composed of string constants) of binaries and fuzzy filename as the code features. Then LibDX adopts a gene map method to implement the binary-to-binary match comparison. Because its approach is far from other TPL detection tools that focus on Java TPLs, we do not compare LibDX with other tools here.

• *Pre-Processing*

In Table 8, we also can see the connection of current TPL detection tools. We can find that most tools choose Apktool [119] as the reverse-engineering tool that can keep the complete package structure of the decompiled code. Baksmali [120] is another reverse-engineering tool, but it just work on the dex file directly. Researchers of LibRadar develop a decompilation tool (i.e., LIBDEX) for quickly handling the dex file, while its functions are similar to baksmali. Compared with Apktool, baksmali is more efficient because it does not need to handle resource files. Androguard [121] is used to extract the class dependency, and Soot [122] is used to extract the CFG (control flow graph).

• *Module Decoupling*

Considering the module decoupling, the features used by existing tools can be divided into four types: i.e., the package structure and package name, homogeny graph, package dependency graph (PDG), and class dependency. In fact, the homogeny graph and PDG also involve the package hierarchy structures. Based on the Table 8, we can find most TPL detection tools adopt the package structure as the module decoupling features, since Java uses packages to organize class files, and almost all existing detection tools focus on Java TPLs. In general, an independent TPL usually corresponds to an independent package structure. However, when these TPLs are imported into apps, things will be more complicated. Some different TPLs may share the same root package when these TPLs are imported into an app. For instance, Google Android GMS [123] and Google Android Library [124] share the same root package "com. google.android". Besides, TPL files can also depend on other TPLs, which also are called nested TPLs [125]. This type of TPLs usually has several parallel root packages, while these interdependent parts together constitute one TPL [27]. If a tool only uses the package hierarchy structure as the module decoupling feature, it may generate incorrect TPL instances.

LibD employs the *Homogeny Graph* as a basic unit in TPL partition to construct TPL instance candidates. A homogeny graph is a directed graph, where each node indicates a package or a class file, and each edge denotes the nodes with inclusion or inheritance relations.

PDG is a weighted directed graph, which includes the package homogeny relationship (parent-child or sibling relationships among packages) and class dependency (e.g., field references, call relations and class inheritance). Existing tools (LibSift and AdDetect) adopt the hierarchical agglomerative clustering (HAC) by cutting a PDG into different modules. These separated modules will be treated as TPL candidates. Considering the accuracy of module decoupling, PDG can achieve better performance than that of homogeny graph because PDG considers the degree of correlation between different parts. This PDG-based method can effectively split modules that only exist the package inclusion dependency without code dependency.

However, homogeny graph and PDG sometimes cannot generate accurate TPL instances because different TPLs can share the same root package or even the sub-packages; different TPLs also can be nested in one package hierarchy tree within different level. If the nested package levels exceed a certain number, PDG also cannot effectively separate these modules.

Another commonly-used module decoupling feature is class dependency. Although ORLIS and LibExtractor adopt class dependency as the code feature, the class dependency relations they adopted are somewhat different. ORLIS just uses the call graph to construct the TPL candidates while LibExtractor adopts class inheritance and interface relations, function invocation relations, and field references as the decoupling features. The advantage of class dependency relations is that it does not depend on the package structures. Therefore this method is resilient to package flattening.

• *Extracted Features*

As can be seen from Table 8, although there are various TPL detection tools, we find that extracted features of existing techniques can be divided into five categories (i.e., fuzzy method signature, CFG, API call, class dependency, and function centroid). Note that some tools may use several code features to represent a TPL at the same time.

The fuzzy method signature [23] means using a placeholder X to replace the developer-defined variables and types in a method signature. For example, `int methodA (classA, int, classB)` is a normal method signature, its fuzzy method signature is `int X(X, int, X)`. We need to use X to replace the developer-defined variable (i.e., methodA) and type (e.g., classA). The purpose is to defend against renaming obfuscation. LibRoad and LibScout use fuzzy method signature as the only code features. Apart from the fuzzy method signatures, LibID and OSSPoLICE also employ other features in signature generation, as mentioned in the following paragraphs.

Both the tool [71] that was developed by Han *et al.* and LibD choose the opcode of the basic block of CFG (control flow graph) as the TPL code feature. They first build the CFG of each method and then extract the opcode of each basic block on the CFG. This feature is hashed as the method feature, and all of the method feature values are concatenated in a certain order as the class feature and each class feature is hashed again as the class-level feature. All class features are sorted by using a certain order according to the hash value and then these features are hashed again as the TPL features. LibID also extracts the CFG from the TPL instance candidate and uses all instructions of basic blocks as a part of a method signature. Besides, the complete method signature also includes the class access flag, superclass name, class interface, and fuzzy method signature. LibID adopts the Locality-Sensitive Hashing to calculate the compared pairs. It then uses the binary integer identification to determine whether a potential matched pairs has a corresponding class dependency.

LibRadar extracts the Android API calls, the total number of APIs and total kinds of APIs to construct the feature vector and calculate a hash value for each feature vector.

LibID uses the class dependency to judge whether the compared TPL instance has the corresponding class

dependency with the potential matched TPL in the database. Each class signature of LibID does not contain the other dependency class features. In contrast, LibPecker and LibExtractor encode the direct class dependency relations into each class signature. The signature of each class contains the dependent classes with class dependencies, method invocation, or field reference relations. The difference between LibExtractor and LibPecker is that LibPecker does not include the interface relationship in the class dependency relations because some obfuscation tools may delete the interface class.

A function centroid [114] can be constructed via a deterministic traverse of the CFG. A function centroid is a three-dimensional vector composed of basic block index, outgoing degree, and loop depth. OSSPoLICE first uses string constants and fuzzy method signatures to identify the potential in-app TPLs, then determines the specific versions of TPLs by exploiting the function centroid.

Among these features, API calls and fuzzy method signatures only contain the syntactic information. The remaining code features contain both syntactic and semantic information at the same time, which can achieve better performance of resiliency to code obfuscation and adversarial attacks. However, such semantic features usually consume more computing resources.

• *Comparison Method*

Most existing tools just adopt a simple similarity comparison method to identify the in-app TPLs by comparing the features with the signatures in database. LSH and hierarchical indexing search are used to improve searching efficiency. Fuzzy hash can effectively handle the code obfuscation since it can tolerate some changes in the in-app TPLs caused by code obfuscation. Similar to the fuzzy hash, LibPecker introduces an adaptive class similarity threshold and weighted class similarity score to identify potential in-app TPLs. This method will give a higher weight value to more important classes, which can effectively distinguish different TPLs and improve the accuracy.

4.1.5 Obfuscation-Resilient Capability Comparison

Code obfuscation is often used to protect Android apps by hiding the actual logic of the apps as well as the used libraries. The commonly-used obfuscation strategies such as API hiding, control flow randomization, dead code removal can modify the code of in-app TPLs, which leads to the code features of in-app TPLs to be different from the original TPL files. The capability of code obfuscation-resilience is one of the most important indexes to evaluate the performance of TPL detection tools. Therefore, we aim to investigate the impact of code obfuscation on these state-of-the-art TPL detection tools. To achieve it, we summarize a comparative result towards common obfuscation techniques in Table 9. Note that LibSift only implements the TPL separation without TPL identification, so we do not discuss the resiliency to code obfuscation here.

• *Dead code removal* (a.k.a. code elimination), is able to delete the TPL code that is not invoked by the host app. When judging whether a tool is resilient to dead code removal, we mainly depend on two points: 1) the module decoupling features and 2) the extracted features. To sum

TABLE 9
Comparison of Obfuscation-Resilient Capability of Different TPL Detection Systems

	Dead code removal	Control-flow randomization	Identifier renaming	String encryption	Package flattening
LibDX	◐	◐	✓	✗	✗
LibExtractor	◐	◐	✓	✓	◐
LibRoad	✗	◐	✓	✓	◐
LibID	✗	✗	✓	✓	✗
LibPecker	◐	◐	✓	✓	✗
ORLIS	✓	◐	✓	✓	✓
Han et al. [71]	◐	✗	✓	✓	✗
OSSPoLICE	✗	✗	✓	✗	✗
LibD	◐	✗	✓	✓	✗
LibScout	✗	◐	✓	✓	✗
LibRadar	✗	◐	✓	✓	✗
PEDAL	✗	◐	◐	✗	✗
AdDetect	✗	◐	◐	✗	✗

✓ : can; ✗ : cannot; ◐ : partially effective.

up, if a tool is resilient to dead code removal, the constructed TPL instances should include at least the method invocation relations and the extracted features should include the invoked methods.

LibRoad, LibID, LibScout, and OSSPoLICE more or less adopt the fuzzy method signatures as the TPL signatures. However, fuzzy method signatures do not include the method invocation information. Some methods can be deleted in dead code removal so these tools are not resilient to dead code removal. LibDX uses the read-only DATA in binary as the code feature and failed to consider the method call relationship. Therefore, LibDX also can be affected by dead code removal. LibD and this tool [71] use the opcode of CFG as the code feature. CFGs include the semantic information of TPLs but some opcode can be deleted in dead code removal so LibD and [71] are partially resilient to this obfuscation.

LibExtractor, LibID, LibPecker, and ORLIS all include the class dependencies in their code features but the role of class dependencies is slightly different. LibExtractor uses class dependencies to construct the TPL instances and encode the class dependencies into the TPL signature. LibID uses class dependencies in the match stage. LibPecker encodes the class dependencies in TPL signatures. ORLIS just considers the call graph to construct the in-app TPL candidate so the code features do not contain the dead code. Besides, ORLIS just reports the class files that belong to the TPLs instead of a complete TPL. Therefore, ORLIS is resilient to dead code removal. However, in practice, the detection rate of ORLIS may decrease to some extent when it handles the obfuscated apps because the detection rate is also affected by the extracted feature granularity, comparison strategy and algorithm.

LibExtractor considers the interface class in feature extraction. However, the interface classes can be deleted by dead code removal. Therefore, dead code removal can affect code features of in-app TPLs, which may decrease the detection rate of LibExtractor. Similar to LibExtractor, LibPecker also uses class dependencies as the code features but it does

not consider the interface classes that can be deleted by some obfuscators. However, both LibPecker and LibID adopt the package structures in module decoupling stages, which leads to the TPL instances including irrelevant code (may including other TPLs or uncalled code), hence reducing the resiliency to dead code removal. Therefore, the resiliency to dead code removal of ORLIS is better than that of LibExtractor, and LibExtractor is better than LibID and LibPecker.

- *Control flow randomization* usually involves modifying the original control flows. Therefore, such obfuscation method can directly affect TPL detection tools that rely on CFG as the code features, including LibID, OSSPoLICE, [71] and LibD. Both LibD and [71] use the opcode of the basic block of CFG as the signature. OSSPoLICE extracts the CFG centroid as the fine-grained feature to identify specific versions. LibID employs the basic block signature as one of the TPL signatures. Besides, the modification of the CFG also could change method dependencies, therefore, it also affects tools that extract the method invocation relations in feature generation (e.g., LibPecker, LibExtractor).

- *Identifier renaming* is also called renaming obfuscation, which can modify the identifiers such as the class name, method name, field name, variables into a meaningless string or hash value. PEDAL and AdDetect leverage the class name as one of the features to identify the classes that belong to ad libraries. As can be seen from Table 9, apart from PEDAL and AdDetect, the remaining tools are resilient to renaming obfuscation.

- *String encryption* is to use the encryption algorithms to encrypt constant strings to protect sensitive information such as the URL, username, and email address. LibDX, OSS-PoLICE, PEDAL, and AdDetect all adopt the constant strings as one of the features so the string encryption can discount their detection performance.

- *Package flattening* can change the package hierarchy structures and package names. Even worse, it can delete the whole package hierarchy structure. We can find that most tools are not resilient to package flattening. ORLIS is the only tool that does not depend on package hierarchy structures and package names in TPL identification so it is completely resilient to code obfuscation. LibRoad has two strategies to identify in-app TPLs. For TPLs with package flattening, LibRoad just adopts a simple method (whether the package is a single character or not) to judge whether this TPL is obfuscated. However, determining whether a TPL is obfuscated by the package flattening technique is non-trivial. The judgment method of LibRoad can lead to false negatives. LibExtractor extracts the relative path of the dependency class as the code feature and uses a placeholder A to replace each path segment. LibExtractor attempts to use this method to enhance the resiliency to code obfuscation. However, different developers may adopt different strategies to change the hierarchy structures, and even the same TPL can be obfuscated into different package patterns. Therefore, this method of LibExtractor can generate different code features for the same in-app TPLs.

Our analysis finds tools that include more semantic code features and use class dependency to build TPL instances can achieve better resiliency to code obfuscation.

4.1.6 Disadvantages Analysis

In this section, we summarize the disadvantages of existing tools, aiming to inspire future researchers to develop better tools.

■ *Disadvantages of Clustering-Based Methods*

The intuition of clustering-based methods to detect TPLs is that TPLs are usually widely-used by many apps [117]. One of the advantages of this approach is that we do not need to collect TPL files in advance; we can directly get the TPL features with clustering and identify in-app TPLs without prior knowledge. However, this method also has the following disadvantages: 1) *The decision of clustering parameters.* Clustering algorithms often require developers to decide the number of clusters or set appropriate parameters to get the clustering results, while in fact deciding the parameters is very challenging. The code similarity of different versions of the same TPLs could be various. For example, we empirically find the code similarity of different versions of the TPL *okio* ranges from about 80% to nearly 100%. While the code similarity of *okhttp* 2.7.x and *okhttp* 3.x only reaches less than 30%. The version diversity makes the parameter decision of clustering-based algorithms difficult, and it is impossible to find the perfect parameters to get completely correct clustering results. The millions of input apps also add difficulties. Some clustering may contain a single TPL version, while others may contain several versions of the same TPL, or even contain different TPLs. 2) *The clustering result may contain impurities.* The quality of clustering-based results generally depend on the number of input apps and the usage rate of TPLs. Apart from the aforementioned cases, some situations also can lead to the results containing noises. For instance, if the input dataset includes several apps that are cloned many times, the clustering results may include the host apps instead of the TPLs. 3) *Labor-intensive verification.* When we get the clustering-based results, we cannot directly use them before labeling the clusters. Developers need to conduct verification to guarantee the clustering results correct. However, this process is labor-intensive and time-consuming. And the verification is also error-prone. Based on the above-mentioned analysis, we can find that it is impossible for clustering-based methods to identify a specific in-app TPL version. 4) *Identify new and niche TPLs.* In fact, clustering-based methods only can identity commonly-used TPLs and may miss some niche and new TPLs, whose recall depends on the number of input apps and the reuse rate of TPLs. Additionally, the input apps could be out-of-date, and updating the new results is really labor-insensitive, which requires clustering and labeling again.

Apart from these common disadvantages of clustering-based methods, existing clustering tools also have their own shortcomings.

LibExtractor generates class signatures by hashing the relative paths of their direct dependency classes. For each class, LibExtractor gets the relative path from this class to its dependency classes, and uses a placeholder A to replace the package name in this relative path link, to gain resiliency to the obfuscation of package renaming. However, the package flattening obfuscation technique can change the hierarchy structures via customizing. Users can choose different packages to obfuscate and the hierarchy level also

can be changed into different forms. Therefore, the same TPL can be transformed into different obfuscated hierarchy structures, leading to the mutation of relative paths for the same TPL. In short, the method of LibExtractor is not resilient to package flattening.

LibRadar uses package hierarchy as the module decoupling feature. However, for in-app TPLs, an independent package may not correspond to an independent TPLs; Several different TPLs may have the same root package and nested sub-packages. An independent TPLs also may have several parallel independent root packages due to the TPL dependency. Using the package structure as module decoupling features is error-prone, which could cut a complete TPL into different parts or cluster several TPLs as the one.

LibD also has the aforementioned issues, although it adopts homogeny graphs as the module decoupling features. Besides, LibD adopts the package-level hash values to detect the in-app TPLs. A small change could lead to the final signatures different. Many obfuscation techniques can easily change the code features, leading to false negatives.

■ *Disadvantages of Similarity Comparison Methods*

Similarity comparison methods require developers to collect the TPLs files to build a predefined feature database. Thus, the size of feature database can directly affect the recall of similarity comparison-based tools. Apart from that, we find existing similarity comparison-based tools have other disadvantages.

One of the obvious disadvantages for most existing similarity comparison tools is they more or less depend on the package structure to construct the TPL candidates. However, package structures are not stable. The package structure of different versions could mutate, and package flattening technique can easily change the package structures. Sometimes an independent package tree could include several different TPLs. Such as the Google ads and Android TPL, they are two different TPLs but they have the same root package "com.google.android".

LibDX is a cross-platform version detection tool that extracts the DATA read-only segment of binary files to identify different TPLs. However, these features may not be very effective against Java libraries, especially after obfuscation, such as string encryption.

Both LibID and LibPecker set too strong assumption on package hierarchy structures. LibPecker assumes the package hierarchy information of a library is retained during the obfuscation [68]. LibID assumes that the inter package hierarchy structure will not be changed during obfuscation [42]. They only can identify the in-app TPLs without inter hierarchy structure modification but package flattening can easily change the structures, which directly discounts their recalls.

LibRoad adopts a combined strategy to identify in-app TPLs. For the non-obfuscated parts of TPLs, LibRoad uses the package name to match the potential TPLs. It assumes that each root package corresponds to a TPL. However, this assumption may not be valid in reality because some different TPLs can share the same root package and one TPL also can have different parallel root packages. Besides, LibRoad uses a simple strategy to judge whether the package names are obfuscated or not, which also can affect detection performance.

The code feature granularity of LibScout and LibRoad are too coarse. Both of them use the fuzzy hash method as

the signature, however, the code features only include the syntactic information and cannot find some tiny changes of the inside method, such as the statement insertion and deletion. OSSPoLICE adopts a two-stage method to identify the in-app TPL versions. It first uses string constants and fuzzy method signature as the coarse-grained features to find the potential in-app TPLs. LibScout also uses the fuzzy method signatures as the signatures. Therefore, OSSPoLICE inherits the shortcomings of LibScout. Additionally, OSSPoLICE adopts the CFG centroid as the fine-grained feature to ensure the specific version of potential TPLs. However, getting the circle loop of a CFG is time-consuming, especially for the method with rich functionalities, which is not good for large-scale analysis.

ORLIS just adopts the call relationship to construct the in-app TPL candidates, which could lose classes that exist in the inheritance relationship, field reference dependencies, etc., resulting in false negatives in identification. ORLIS just reports the matched classes, which is not practical for most users either.

■ *Summary*

Although different tools have different disadvantages, the most primary issue of existing tools is using package structures in TPL candidate construction, which directly leads to many false negatives and false positives. Apart from ORLIS, the remaining tools more or less depend on the package structure to conduct the module decoupling. First, these tools are not resilient to package flattening obfuscation. If the TPLs are without the package structure, these tools cannot handle TPLs without packages that could cause false negatives. Second, using package structures to build the TPL candidates cannot handle the TPL dependency and different TPLs with nested packages.

4.1.7 Essential Findings

Based on above analyses, we summarize the essential findings in TPL identification.

• We find that most existing TPL detection tools more or less depend on the package structure to conduct the module decoupling. However, package structures are not stable, which can be easily obfuscated by code obfuscation. Besides, an independent TPL also can include several parallel root packages. One package tree also can contain multiple TPLs. Therefore, package structure cannot accurately split the in-app TPL instances, leading to the low recall. TPL detection tools use the class dependencies as the module decoupling features can resiliency to package flattening because this feature does not depend on the package structure.

• For TPL detection tools, the extracted features within more semantic features can achieve better resiliency to code obfuscation than tools only include the syntactic information. In the comparison stage, class-level features can achieve better resiliency to code obfuscation than package-level features.

• Most TPL detection tools have high precision but low recall. As for the version identification, we have a long way to go. We need to overcome many challenges, especially the diverse code differences among different versions of the same TPL. We find the code differences among some

versions are huge and the code differences among some versions are very tiny. How to choose the feature granularity can effectively reflect the differences and does not affect the detection efficiency is what future researchers need to consider.

● Future researchers can consider handling the challenges, such as the TPL shrink and optimization.

4.1.8 Implications and Future Work

Even though current tools have various disadvantages, they also have many advantages in different detection steps. We think existing techniques are highly complementary to each other. By combining their advantages, we can design better tools. For instance, we can learn from the method of ORLIS and LibExtractor in module decoupling. We suggest using the class dependencies but does not include the interfaces to build TPL candidates, which can achieve better resiliency to package flattening. For feature extraction, we can refer to the method of LibPecker; each class signature includes their direct dependency classes. This method includes rich semantic information, which can achieve better resiliency to code obfuscation. For the comparison stage, we can adopt the strategy of LibRoad; regarding the non-obfuscate TPLs, we can first use the package name to narrow the search scope and use signature-based methods to locate the specific versions.

Based on our analysis, we find that all current tools adopt the static analysis method to identify in-app TPLs. Therefore, all of them cannot identify the dynamically-loaded TPLs and classes. Besides, they also are not resilient to the sophisticated code obfuscation, such as API hiding that is a kind of obfuscation by leverage the Java reflection and dynamic class load to hide a part of codes, visualization-based protection that translates the code into a stream of pseudo-code bytes that is hard to be recognized by the machine and human and only can be interpreted during the runtime. None of these problems have a good solution. Hence, we suggest future researchers try to solve these problems.

On top of that, most research just focuses on the detection of Android Java TPLs while only two tools can detect native libraries but they are not resilient to string encryption, and just use simple syntactic features, leading to not resilient to other sophisticated code obfuscation. We highlight that future researchers pay more attention to the native library detection.

4.2 TPL Security-Related Issue Analysis

This section introduces research on existing TPL security-related issues.

4.2.1 Research Background

TPL is a double-edged sword. On the one hand, it can facilitate app development and decrease the release time; on the other hand, TPLs could also bring various security risks. As previously mentioned, the permission-control mechanism of Android only works at the app-level; therefore, the TPLs and the main app share the same permissions. This permission model obviously violates the principle of least privilege [12], [13]. Malicious TPLs could abuse the permissions of host apps and easily access privacy data and resources,

TABLE 10
A Summary of TPL Security-Related Issue Analysis

Function	Tool/First Author	Year
Privacy Leakage	Son *et al.* [78]	2016
	Wei *et al.* [84]	2016
	Pluto [9]	2016
	Paturi *et al.* [87]	2015
	Moonsamy *et al.* [7]	2014
	Short *et al.* [8]	2014
	Leontiadis *et al.* [103]	2012
	Stevens *et al.* [104]	2012
Vulnerability Identification	Droid-V [73]	2017
	AdRisk [28]	2012
Malicious TPL Analysis	MadDroid [22]	2020
	MadLife [65]	2019
	Rastogi *et al.* [83]	2016
	LibFinder [80]	2016
	Kühnel *et al.* [90]	2015
	APKLancet [92]	2014
	Duet [94]	2014
	Brahmastra [98]	2014
Ad Frauds	Madfraud [21]	2014
	DECAF [20]	2014
	ClickDroid [89]	2015
	Dong *et al.* [70]	2018
	FraudDroid [19]	2018

resulting in data and privacy leakage. Besides, some serious vulnerabilities in TPLs are disclosed as Common Vulnerabilities and Exposures (CVEs). Researchers [23] also found that developers usually do not replace the vulnerable TPLs in time, which aggravates the spread of vulnerabilities. Besides, some attackers attempt to masquerade malicious libraries by modifying the package names as existing legal libraries. For example, DroidKungFu [126] is a well-known malware, which uses names like "com.google.update" and "com.google.ssearch" to confuse users. These malicious libraries usually can leak users' private data, hijack the SNS account, read/send text messages, or lead to money loss. Furthermore, some unscrupulous developers try to violate the ad library guidelines and develop ads that trick users into clicking or watching the in-app ads [21].

Another security risk comes from the direct and transitive use of TPLs. Android apps are widely built on the top of TPLs; however, if the imported TPL includes a vulnerability and can be easily exploited, without a doubt, it can bring inestimable risks to downstream apps and app users. Therefore, it is necessary to take a closer look at existing research on TPL-related security issues to understand the current research status and the existing gap.

4.2.2 Existing Research

Considering existing threats, many researchers conduct different studies in these directions. Based on our analysis, we find current research primarily focuses on the following fields: 1) privacy leakage analysis, 2) vulnerability identification, 3) malicious TPL detection and analysis, 4) ad frauds. The research related to each field can be seen in Table 10.

Privacy Leakage Detection. Son *et al.* [78] first introduced how Android ad libraries isolated the in-app ads to prevent them from sharing the privileges of the host apps. They find the ad libraries confined the ads to a separate WebView instance, which can prevent the ads from reading the local storage. They further proposed an attack model under this protective measure, where malicious advertisers could infer whether local files with a certain name exist within the storage. They demonstrated how sensitive information of users, including browsing history and social graph, could be revealed by this existing information alone. They also proposed defensive measures towards such exploits.

Wei *et al.* [84] conducted a survey on 231 real users and also verify ad revenue heavily depends on both users' privacy data such as user interests and demographic information. They point out that a user's demographics information plays an essential part in determining the targeted ads. Pluto [9] is an automatic modular framework for privacy risk assessment of in-app ad libraries. This work systematically explored the data collection of ad libraries from four channels: using unprotected APIs to learn other apps' information on the mobile; using protected APIs through permissions inherited from the apps to access sensitive data, gaining access to the storage of host app, and getting user inputs from the host apps. Pluto combines the static and dynamic analysis and NLP techniques to assess 2,553 real-world apps from Google Play and finds a trend in ad libraries to become more aggressive towards reachable user information. As we all know, the permission lists only display information accessible to the host app but ad libraries can gain access to sensitive information from users' devices due to the shared permission mechanism. Paturi *et al.* [87] proposed a novel icon-based privacy threat interface that displays privacy risks from app providers and TPLs separately to substitute the traditional permissions list before installation. They considered three privacy granules: location, identity, and query (refers to a user's search queries). Besides, they performed two online usability studies to obtain user feedback on their new permission module. They detect related data access through both static and dynamic analysis. Their user studies show that users are positive to the new interface, which helps them better understand privacy threats within apps. Moonsamy *et al.* [7] analyzed the device ID leakage problem from the in-app ads. They studied 123 apps by combining static analysis and dynamic analysis. They exploit DroidBox [127] to track the information leakage and find that 13 apps leak device-related information. To detect data leakage from TPLs, Short *et al.* [8] extended DroidBox by modifying TaintDroid [128], a taint tracking tool. They use dynamic analysis to catch the data leakage. Leontiadis *et al.* [103] analyzed more than 250,000 Android apps and proved that the demographic information has a close link with the ad-supported revenue delivered to developers. It is conducive to more revenues if a precise targeting ad is delivered to the customers in need. At the same time, they also point out that there are no conflicts between privacy protection and developers' revenue. Therefore, they proposed a feasible privacy control framework, which can achieve an equilibrium between the private information flow and the generated advertisement revenue. This framework established a feedback control loop mechanism that can adjust the level of privacy protection on smartphones based on the generated ad revenues. Stevens *et al.* [104] first compared the commonalities and differences between in-browser ads and in-app ads and found that in-app ads are more likely to leak users' privacy. They examined 13 Android ad networking and attempted to find the risk of privacy leakage from the perspective of permission. Besides, they also discover the vulnerabilities in the use of the JavaScript extension mechanism in several ad libraries. Finally, they proposed potential solutions to address the above issues.

Vulnerability Identification. Droid-V [73] mainly studies four types of vulnerabilities of mobile libraries in both free apps and paid apps, including information disclosure, SSL/TLS and cryptography, inter-component communication, and WebView. AdRisk [28] is a vulnerability detection framework for in-app advertisements on Android platform. This research reveals a set of privacy and security problems in 100 representative ad libraries. AdRisk first collected sensitive APIs and the required permissions; it then exploits the control flow graph to find a dataflow path from the dangerous API calls to an external sink, which could lead to personal data leakage. Apart from reporting potentially-feasible paths, AdRisk also analyzes five suspicious code patterns: use of reflection, dynamic code loading, permission probing, JavaScript extension functions, and reading installed list of packages. AdRisk found a number of security risks in 100 representative ad libraries such as uploading sensitive information to ad servers, executing suspicious code in the context of the host app environment, fetching malicious payloads.

Malicious TPL Analysis. MadDroid [22] is a dynamic framework for automatic detection of malicious ad contents in android apps. It adopts a novel method to identify ad traffic by building a mapping between ad libraries and ad hosts through HTTP hooking. They classified devious ad contents into two categories: ad loading content (pre-click) and ad clicking content (post-click). They adopted Mad-Droid to 40K adware apps and found 6% of apps with devious ad contents.

Chen *et al.* [65] studied the security threats introduced by in-app advertising, including click fraud, malvertising, and inappropriate ad contents. They developed Madlife, a dynamic ad collection tool that automatically records all ad-related data for android apps. They collected 83K ads from 5.7K apps on Google Play and discovered 37 apps with click fraud and 1.49% ads related to malvertising. They also found the click fraud and malvertising are strongly correlated.

Rastogi *et al.* [83] held the opinion that some apps may be benign, but such in-app ads can redirect users to a certain website, which could play an essential role in propagating attacks. Thus, they analyzed more than 600,000 Android apps from Google play and four other app markets from China in two months. They attempted to understand the web-app interface attacks. They identified several malware and scam campaigns propagating through in-app ads and web links.

LibFinder [80] is a cross-platform system that can detect potentially harmful libraries over both Android and iOS. Based on the observation that many iOS libraries have counterparts in Android apps. The authors found that a
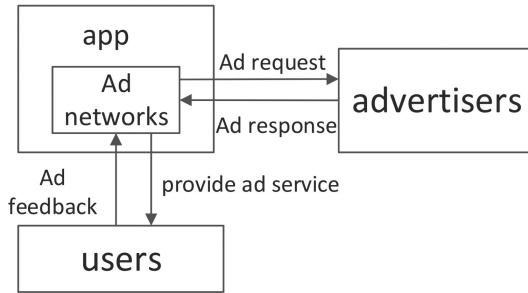
Fig. 6. Overview of the in-app advertising ecosystem.

TABLE 11
Summary of Ad Fraud Detection Techniques

Tool Name	Dynamic Detection	Network Traffic Data	UI Features
MadFraud	✓	HTTP request pages	-
DECAF	✓	-	DOM tree
FraudDroid	✓	HTTP request data	DOM Tree
Dong et al.	✓	-	Ad views
ClickDroid	✓	-	-

"-" means not use.

considerable portion of third-party services to Apple devices are also provided to Android. LibFinder first identifies suspicious libraries on the Android platform and then tries to find the corresponding iOS versions based on the common features of the same service with the Android and iOS libraries.

Kühnel *et al.* [90] performed a fast detection of ad libraries within android malware apps by statically checking the smali code for the invocations of advertising APIs which are publicly available. They further discovered a decrease in the usage of ad libraries for android malware. Through a manual analysis of the samples from malware families that adopt fewer ad libraries, they discovered some of the malware would send premium SMS on the first run.

Brahmastra [98] is an app automation tool for dynamically testing the security of third-party components within apps. Brahmastra first constructs a call graph for the app using static analysis, through which they build activity transition paths for exercising the TPL code. Brahmastra then attempts to jump start the activities in the transition paths to trigger TPL code through Android Debug Bridge [129]. They further rewrite apps for self-execution by inserting the callback functions that trigger the desired transitions. Their experiment shows Brahmastra outperforms start-of-the-art GUI testing tools by 170% in triggering targeted TPL methods.

APKLancet [92] is an automated app diagnosis system that can identify the tumor payload in Android apps. Tumor payloads include the malicious code fragment and some unwelcome advertising/analytical libraries. Duet [94] is a library integrity verification tool for Android apps, which aims to detect 1) library modification threats, 2) masquerading threats, and 3) aggressive library threats. Duet extracts the library files from the DEX files and generates library digests for both the original library and the libraries from tested apps.

Ad Fraud Detection. Before introducing the concept of ad frauds, we first illustrate the mobile advertising ecosystem in Fig. 6. The advertising ecosystem involves three essential roles: the advertisers, developers, and users. The ad libraries (ad networks) act as the bridge connecting the users and developers, the developers and advertisers. Advertisers usually attempt to propagate their products or service through the in-app ads. The mobile developers who can embed the ad networks in their apps help the advertisers to disseminate their commodities. When ads are observed (called impression) or clicked by users, the developers can get revenue from the advertisers. The ad network serves as a proxy that connects the app developers and advertisers by exchanging the ad impression and revenue across the ecosystem. Ad networks usually require that app developers strictly follow the guidelines or documents [70] that are used to instruct developers to set the in-app ads. However, some unscrupulous developers attempt to cheat either the advertisers or users by violating the guidelines. For example, unscrupulous developers can modify the code to fetch the ads but make them run in the background instead of displaying the ad impression for users. Some ads are resized by developers, making them too small to read. Some unscrupulous developers may place the ads that close to or cover the UI controllers of the main apps. These settings may affect the user experience and induce users to click the ads. We call all violations of the guidelines as *ad frauds*, regardless of whether they cheat the users or the advertisers. Dong *et al.* [70] summarized a taxonomy of behavior policies based on Admob [130], a popular ad library, and then investigated 3,661 popular apps that all apply the Admob library to detect the advertising policy-violation apps. They first designed an automated event-driven testing tool to send random events and captured the state of the user interface (UI). After that, they removed other UI states and kept the ad-related UI states based on the UI type features, UI location features, and strings. Finally, they performed a violation detection. This research shows that behavior policy violation is a real problem in Android ecosystem. Security experts and developers need to pay more attention to these issues.

Cho *et al.* [89] performed an empirical study of click fraud on eight popular ad networks to examine their ability to defend against such ad fraud. They developed ClickDroid, which automatically simulates click events on mobile ads repeatedly. ChickDroid updates the device identifiers after each click as an attempt to bypass the security policies of ad networks and gain more profit. Their experiment showed 6 out of 8 ad networks failed to detect their fraudulent clicks. They also discuss countermeasures against such click fraud.

According to our analysis, we compare existing ad fraud detection techniques in Table 11. All of them employ the dynamic method to detect ad frauds. MadFraud [21] attempts to find two kinds of ad frauds in mobile apps: 1) requesting ads while the app is in the background. 2) clicking ads without user interaction. MadFraud builds the HTTP request trees and extracts the features from the HTTP request pages. By using the machine learning method, it can identify the ad-impression and ad-clicking frauds. DECAF [20] focuses on the placement ad frauds. It uses monkey [131] to get the UI state transition graph. By extracting the information from the DOM tree, DECAF can identify the placement ad frauds. FraudDroid [19] defines two kinds

TABLE 12
Comparison of Detection Capabilities of Different ad Fraud Detection Tools

Fraud Types	MadFraud	DECAF	FraudDroid	ClickDroid
Automatic Click	✓	✕	✕	✓
Ad Hidden	✕	✓	✓	✕
Ad Size	✕	✓	✓	✕
Ad numbers	✕	✓	✓	✕
Ad Overlap	✕	✓	✓	✕
Interaction	✕	✕	✓	✕
Driven-by download	✕	✕	✓	✕
Outside Ad fraud	✓	✕	✓	✕
Frequent	✕	✕	✓	✕
Non-content	✕	✕	✓	✕

✕ *means cannot; ✓ means can.*

of ad frauds, i.e., the static placement fraud and dynamic interaction fraud, including nine types. It can detect both of them by extracting the UI features and traffic features. The work of FraudDroid is based on the work of Dong *et al.* [70].

As aforementioned, advertisers pay the publishers based on the number of ad impressions and ad clicks. Based on the previous research [19], [20], [21], we summarized ten different ways of ad frauds as follows. 1) *Automatic Click Fraud:* Clicking on ads without user interaction. 2) *Ad Hidden Fraud:* Ads are placed under other controls or are hidden so that users cannot find them, giving users a wrong impression that it is an "ad-free app". 3) *Ad Size Fraud:* Developers resize the ads to make them too small to read, or so large that users are forced to click or view the ads. 4) *Ad Number Fraud:* The number of ads exceeds a normal quantity in one UI page. Developers want to attract users' attention and increase the probability of interacting with the ads, which obviously affects user experience. 5) *Ad Overlap Fraud:* The ads are placed close to the actionable components or cover the normal functional UI components of the host app. Developers want to trigger accidental clicks in this way to earn illegal revenue. 6) *Interaction Fraud:* When users interact with host apps, the advertisement pops up unexpectedly, causing the user to click. 7) *Driven-by download:* Triggering the unintentional download of other apps when clicking on the ads. 8) *Outside Ad Fraud:* Displaying the ad even when the app is running in the background without any interaction with users. 9) *Frequency Fraud:* Ads being popped up too often upon different user operations. 10) *Non-content Ad Fraud:* Placing ads at non-content-based pages such as the login or exit screen, which may cause users to mistake the ads for the real app content. Following the taxonomy of FraudDroid, we divide them into two categories: the static fraud involving the ad placement issues and the dynamic fraud involving the user interaction. Among these ad frauds, Ad hidden fraud, Ad size fraud, Ad number fraud and Ad overlap fraud are static frauds; the remaining frauds are dynamic frauds.

Table 12 compares the detection capabilities of Mad-Fraud, DECAF, FraudDroid, and ClickDroid. As can be seen from the Table 12, MadFraud can detect two types of ad frauds, including the `automatic click` and `outside ad fraud`. DECAF can detect all the static ad frauds. Fraud-Droid has the most robust detection capability, which can detect all the ad frauds except the `automatic click`. In

contrast, ClickDroid can only detect automatic click and cannot detect other types of ad frauds.

4.2.3 Future Work

According to our survey, we find current research mainly focus on the security problem of Ad libraries, such as the privacy leakage of ad libraries, violation of ad fraud. Besides, some violation behaviors of TPLs are also investigated. However, we find only two papers focus on vulnerable TPL analysis. In addition, the research scope of existing work on vulnerability of TPL analysis is also very limited. For example, Droid-V just studied four types of vulnerabilities of TPLs. AdRisk detected the potential bugs in ad libraries only. In fact, existing vulnerabilities of Android TPLs are far more than those that have been revealed by current research. Many vulnerabilities of Android TPLs can be found in National Vulnerability Database (NVD) [132] or other related website [133]. Currently, there lacks a systematic study revealing the threats of these vulnerabilities. There still exist many blind spots in our understanding of these vulnerabilities, such as their number and types, their severity and threats to apps and users, whether these vulnerabilities are easy to exploit, and how to exploit them. Apart from these known vulnerabilities, there may be a large number of 0-day vulnerabilities in TPLs that have not been found. How to detect these vulnerabilities and reveal their threats is also another tough work. We believe that revealing the vulnerability issues of TPLs would be a significant contribution to our community, which also can help improve the quality of apps. We suggest future researchers conduct more related work in this direction.

4.3 TPL Privilege De-Escalation

4.3.1 Research Background

There are two security mechanisms in Android system: the sandbox mechanism and the permission framework. Regarding the sandbox mechanism for isolating apps, Android system assigns a unique user ID (UID) to each app and lets it run in a separate process. Regarding the permission framework for controlling the privileges of each app, Android system allows apps to access the system resources (e.g., telephone ID/status, location, camera) with the corresponding permissions. This permission-based security model can restrict apps from accessing resources and private data. *However, these two security mechanisms have a flaw that they only work at the app level.* Apps and their incorporated third-party libraries share the same permissions and UID with the host app, which means TPLs could also access the same sensitive data and resources if the host apps own these permissions.

We provide an example to explain this security mechanism in Fig. 7, which illustrates the relationship between the host app, TPLs, as well as the Android permission system. When users try to install an app on a mobile device, Android system will parse the APK file and get the component information and the requested permissions during the installation process. In Android, Package Manager Service (PMS) is responsible for creating a new user and private storage for this new app. When the app is launched, Activity Manager Service (AMS) initiates a process to run this
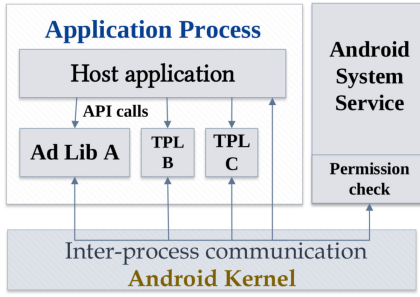
Fig. 7. An example of security mechanism of Android apps.

Tool/First Author	Year	Venue
Zhan *et al.* [76]	2017	ACISP
FLEXDROID [58]	2016	NDSS
LibCage [79]	2016	ESORICS
PEDAL [16]	2015	MobiSys
NativeGuard [95]	2015	Wisec
COMPAC [93]	2014	CODASPY
AFrame [15]	2013	ACSAC
SanAdBox [14]	2013	ICC
AdDroid [18]	2012	ASIACCS
AdSplit [17]	2012	USNIX Security

app. AMS first retrieves the process information (i.e., UID, GID, and storage information) from the PMS, and sends a process creation request to the Zygote process that will fork a new process (sandbox) for this new app. This security mechanism works at the app level. Therefore, the bundled TPLs and the host app share the same permissions, UID, GID, and storage space. We can see an example of this mechanism from Fig. 7. This app has one ad library A and other two third-party libraries, B and C, running in the same process. Both the host app and the TPLs can access the Android system service via inter-process communication (IPC) if they have the corresponding permissions to control these resources.

4.3.2 Existing Problem

Based on previous introduction, we can find that the permission mechanism and the sandbox mechanism can bring potential risks because both of them lead to over-privileged problems [12], [13], which could pose threats to the privacy and security of users. To handle this situation, many researchers attempted to solve this problem through *TPL isolation*. They tried to find methods to implement privilege de-escalation.

Besides, we also need to note that Android provides two file storage methods for apps, i.e., external storage and internal storage. The external storage is global storage that can be accessed by any application while the internal storage is a private space that can only be visited by the app itself. Since the TPLs and the host app share the same UID and process storage, the TPLs are able to get sensitive data from the host apps. Therefore, it is necessary to give separated storage to the host app and TPLs.

4.3.3 Existing Solutions

Existing isolation techniques can be concluded into two types, as illustrated in Fig. 8 which are based on the example in Fig. 7. The first scheme in Fig. 8a tries to split these TPLs into different processes and lets Android system assign a separate storage space and permissions. Under this situation, these TPLs can only access the system service with the corresponding permissions that the TPLs required, and they cannot share the permissions with the host app. This scheme usually requires the modification of the Android framework. TPLs can access the system resources via two ways: (1) directly invoke the system calls to get the system services or resources; (2) indirectly access the resource through the interfaces from the host app logic. The second scheme is shown in the Fig. 8b attempts to block the direct system invocations so that TPLs can only get the resources through the host app by API calls. In this situation, developers can configure some fake information to the TPLs rather than share the real information with them. This scheme does not require the modification of the operating system and APIs; instead, it usually requires the developers to rewrite the resource access functions to achieve the TPLs isolation and permission separation.

4.3.4 Existing Research

According to our collected papers, we summarized ten publications regarding TPL isolation techniques in Table 13, most of which were published in security-related conferences. Since some tools are improved based on previous tools, we will introduce these tools in chronological order.
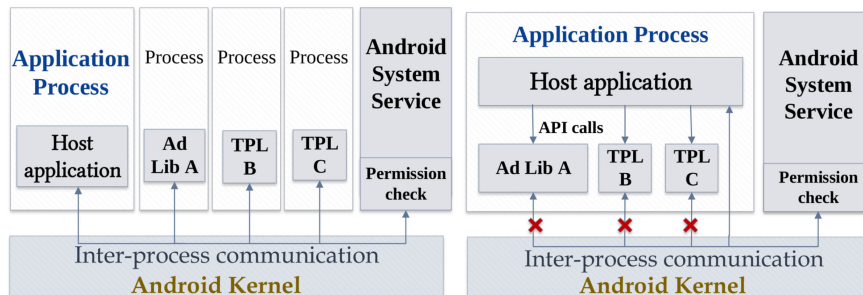
(a) Split the TPLs into independent processes (b) Cut off the direct communication between the system service and the libraries

Fig. 8. Two TPLs isolation schemes based on the example in the Fig. 7.

• *Scheme of splitting TPLs into independent processes.*

AdSplit [17] is built on QUIRE [134] which can separate the advertising TPL and the host app into two activities within different processes but sharing the same screen. AdSplit achieves the permission isolation since the ad libraries and the host app run in separate processes. The screen sharing is achieved with a transparency technique where the advertisement activity is put under the app activity with a see-through region for the in-app ad slot. However, there are some disadvantages of this method. First, the transparency technique will cause considerable overhead in drawing, especially for a mobile phone, since it requires several layers of drawing windows to be combined. Second, the transparency technique has been wildly used by click-jacking, which can make attack detection of apps more difficult. Third, this method also may affect the normal operation of the ads and directly block the interaction with the users and ad network.

AdDroid [18] achieves the ad libraries and host app isolation. To accomplish this goal, AdDroid introduces extended advertising APIs into the Android SDK. Meanwhile, it also adds an AdDroid Service like the Android system service, which provides the all advertisement required permissions. Host apps offer the APIs with configuration data to identify which advertising networks and to fetch advertising contents, contextual information about the advertisement. The extended APIs handle the user interface events.

SanAdBox [14] is a privilege separation framework that can separate the advertising libraries from the host app and run the advertising libraries as different independent applications. In addition, SanAdBox is an updating sandbox where the host app and ad libraries run into and only can invoke their own permissions. The authors conduct an installer responsible for managing the communication between the host app and dependent advertising libraries. The advantage of SanAdBox does not require system modification.

AFrame [15] achieves not only the process and permissions isolation but also the display and input isolation. A-Frame is similar to a typical View component; inside this area runs the TPL process, which is called AFrame process. The main process runs the host app. From the user perspective, the host activity and the TPL activity look like the one. From the system perspective, there are two processes. Compared with AdSplit, the advantage of AFrame is that the users can interact with the advertisement. Besides, AFrame can support all TPL isolation. AFrame needs to modify the Android framework and bytecode of apps. AFrame adds a new parsing module in Package Manager Service (PMS). The modified PMS will create an extra process for AFrame. In this way, AFrame can create an independent process for TPLs. TPLs in different processes cannot share the same permissions with the host apps. Therefore, AFrame achieves the permission isolation, process isolation, input/output isolation, and display isolation.

COMPAC [93] also achieves a fine-grained access control at third-party components level by modifying the Android Kernel and Framework. It keeps the original permission checking strategy to avoid compatibility issues. COMPAC extends the Android access control architecture, the Framework Reference Monitor (FRM), and Kernel Reference Monitor (KRM) by adding two Policy Managers (PMs):

Framework Policy Manager (FPM) and Kernel Policy Manager (KPM). The two reference monitors are responsible for permission checking of the app while the new PMs are responsible for the permission checking of TPLs.

NativeGuard [95] is the first work that proposes to separate the native libraries from Android apps to achieve limiting the over privileges of native libraries. It isolates native libraries by splitting one Android app into two apps. Specifically, by leveraging the reverse-engineering techniques, NativeGuard first finds the code of all the native libraries and then moves them to an entirely-new service app, which can be started by the host app. The two apps can communicate through the interfaces defined by the Android Interface Definition Language (AIDL) [135]. They create a library that acts as a proxy to call the AIDL interface. NativeGuard does not need to modify the Android framework or access to the source code of an app.

FLEXDROID [58] modifies different access control policies such as the Kernel of Android OS, the Android Framework, Dalvik VM, Bionic, the Java core library, the Binder library, and SELinux setting to achieve a fine-grained access control for TPLs. FLEXDROID allows developers to totally control the grant permissions and sensitive behaviors to access the system resources and data. Moreover, FLEX-DROID implements a new permission mechanism to conduct the inter-process stack inspection and isolate TPLs, which is not affected by JNI [136], Java reflection [137] and dynamic code execution [138].

Zhan et al. [76] extend the package manage service (PMS) to enable the system to identify the permission at the library-level. Besides, it allows developers to configure the permission policy at the library-level instead of the app-level. They add a module named the system libraries to get the specific Android API that is invoked by the host app and the corresponding invoking method in the host app. When the system finds the invoking method that belongs to TPL, it will search the corresponding privileges and then decide whether to grant it or not. The advantage of this approach does not require considerable modification on host apps, and the modified system can easily adapt to the DVM and ART virtual machine.

• *Scheme of cutting off the communication between TPLs and the host apps.*

PEDAL [16] resets the resources access rules for the ad libraries by rewriting the bytecode on user-specified privacy policy. It implements three different levels of accessing privileges (i.e., allow, obscure, block). Users can configure different access privileges to the ad libraries of the registered apps through the controller app.

LibCage [79] allows developers to grant different permissions to each TPL and put each TPL in a separate file, and it does not need to modify the Android Framework or the bytecode of libraries. LibCage creates a new sandbox on the system-level and lets the host app and TPLs work in this process. Besides, LibCage contains a permission checker that can restrict the access of TPLs to sensitive resources.

4.3.5 State-of-the-Art Techniques

Generally speaking, the main purpose of existing tools is to achieve TPL isolation and privilege de-escalation. Table 14

TABLE 14
A Summary of TPL Isolation Techniques

Technique	Zhan et al. [76]	FLEXDROID	LibCage	PEDAL	NativeGuard	COMPAC	AFrame	SanAdBox	AdDroid	Adsplit
Dynamic	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓
Bytecode modification	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗
System modification	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓
Java reflection	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
Dynamic code execution	✓	✓	✓	✓	✗	-	✓	✓	✗	✗
DVM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ART	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗
Ad libs	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
TPLs	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗
Permission separation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Storage separation	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓

✓ means can; ✗ means cannot; - means not mention.

TABLE 15
A Summary of Related Work on TPL Maintenance

Function	Tool/First Author	Year
Dependency Conflicts	LibHarmo [31]	2020
	DECCA [29]	2018
	RIDDLE [30]	2019
TPL Updating	Wang *et al.* [32]	2020
	Yasumatsu *et al.* [24]	2019
	APPCOMMUNE [67]	2019
	Salza *et al.* [66]	2019
	Salza *et al.* [69]	2018
	Ogawa *et al.* [72]	2018
	Derr *et al.* [75]	2017
	Ruiz *et al.* [82]	2016

provides the comparison details of ten different TPL isolation tools. We summarized existing techniques from the following aspects.

Modification Mode. The modification mode usually involves three ways: 1) modify the bytecode of Android apps 2) modify the Android system. 3) both 1) and 2). Both PEDAL and SanAdBox involve modifying the bytecode of the app, while the remaining five tools, i.e., FLEXDROID, COMPAC, AdDroid, Adsplit, and the tool of Zhan *et al.* [76] extend the Android framework to achieve the TPL isolation and privilege separation. AFrame and NativeGuard modify both of them.

Dynamic Feature Support. Simply modifying the permission check mechanism cannot check certain dynamic sensitive behaviors such as the dynamic class load, JNI method, and Java reflection. These dynamic methods can easily bypass the DVM permission checking module in a virtual machine while the dynamic class loading or dynamic code execution and reflection technologies generally cannot affect bytecode rewrite methods. But FLEXDROID, PEDAL, SanAdBox, LibCage, and [76] can also handle the java reflection and dynamic code execution.

Virtual Machine Support. As shown in Table 14, PEDAL, LibCage, NativeGuard, and [76] support the Dalvik Virtual Machine (DVM) and Android Runtime (ART) virtual machines. The remaining systems AFrame, SanAdBox, AdDroid, COMPAC, FLEXDROID, and Adsplit only can adapt to the DVM. Generally speaking, if the modification is on Android apps, it can support both ART and DVM. If the changes just happen on DVM-specific features, it cannot support the ART mechanism.

Library Type Support. PEDAL, SanAdBox, AdDroid, and AdSpilt only achieve the ad TPL isolation; NativeGuard isolates the native libraries; the remaining tools all implement the Java TPL isolation.

Separation Items. TPL isolation usually includes two parts, i.e., *permission separation* and *storage separation*. Both of them involve limiting the privileges to TPLs and improving the security performance of apps. All of these systems have already achieved permission separation, while FLEXDROID, NativeGuard, and COMPAC do not support storage separation. Besides, AFrame achieves not only the process and permission isolation but the display and input isolation.

4.3.6 Summary

According to our observation, TPL isolation techniques are directly related to the Android security mechanism, and the isolation process usually leverages dynamic analysis techniques. Based on our analysis, we know that the PMS involves process allocation and permission management. Hence, existing tools usually involve PMS modification or sandbox mechanism updating. Besides, we also find that most existing tools only can handle Java TPLs; only NativeGuard can handle Native libraries. We expect future work can conduct more research on Native libraries, even though the task is challenging. Also, the state-of-the-art isolation methods are not very practical because they need to modify either the Android apps or the Android system. Providing the new process for each TPLs will add resource consumption, especially for mobile devices. Unlike the communication with other apps, TPLs have more interaction with the host app. We hope future researchers can propose more practical and valuable approaches. Furthermore, none of them are publicly available. Therefore, it is impossible to compare their advantages and disadvantages from practical perspectives.

4.4 TPL Maintenance

For TPL maintenance, we mainly introduce research on TPL dependency conflicts and in-app TPL updating. The related work can be seen from Table 15.

4.4.1 Dependency Conflicts

● *Research Background.*

Dependency conflict (DC) is another essential issue in Android TPLs. Nowadays, many apps and TPLs directly or transitively depend on other TPLs. As a result of the intensive dependencies on TPLs, an app may depend on multiple versions of the same TPL or class, among which only one version will be loaded. Dependency conflict occurs when the loaded version cannot cover the features required by the app, leading to runtime exceptions or system crashes [29].

● *Existing Research.*

In our research scope, we find three related work. Even though all of them focus on Java projects, it still provides insights to the research on the DC issues of Android platform, which involves extensive use of Java libraries. Therefore, we include the three research in our paper repository.

DECCA [29] is a DC detection tool based on static analysis, which supports the assessment of DC issues' severity. Through an empirical study on real-world DC issues, they categorized DC issues in three patterns: conflicts in library versions, conflicts in classes among libraries, and conflicts in classes between projects and libraries. DECCA first constructs the TPL dependency tree from the dependency management script (e.g,pom.xml) and identifies duplicate classes with the same fully-qualified name. DECCA further deduces whether the duplicate classes are loaded or shadowed based on the class loading mechanisms of the build tools. DECCA then extract methods referenced by the host project from the duplicate classes and deduce a DC severity level according to the subset relationship among the referenced, loaded, and shadowed method set.

RIDDLE [30] is an automated test generation approach for JAVA projects with DC issues. It further collects crashing stack traces for reproducing and debugging the relevant issues. To assist developers in facilitating diagnosing and fixing DC issues, RIDDLE can report a) the root cause of dependency conflicts in the project, b) risky method set, and c) test cases with the fewest unrestored branch conditions and corresponding program variant. RIDDLE is a follow-up work of DECCA. DECCA base on the static analysis to identify dependency conflicts and can provide the severity levels of dependency conflicts. RIDDLE uses DECCA to identify the risky methods causing the dependency conflict issues, and can also offer stack trace information and failure-introducing conditions with the help of dynamic analysis. RIDDLE provides more practical value to developers, which can help them reproduce and debug the dependency conflict issues.

LibHarmo [31] tackles the TPL version inconsistency problem in JAVA projects, which is one of the major cause of the DC issues according to DECCA. LibHarmo is an interactive and effort-aware library version harmonization technique. It first detects TPL version inconsistencies and false consistencies (a separate declaration of the same versions of TPL) through statically analyzing the inheritance relationships and the declared TPL versions among POM files. LibHarmo then recommends harmonized TPL versions for developers to choose, with detailed harmonization efforts manifested by the number of called libraries APIs deleted or changed in the harmonized version and the number of calls to those libraries APIs in the project. LibHarmo significantly outperforms Maven's `enforcer` Plugin with 4X inconsistencies detected in their evaluation.

Compared to DECCA and RIDDLE which mainly focus on the examination of DC issues, LibHarmo focuses more on the maintenance and harmonization of TPLs. Specifically, LibHarmo differs from DECCA and RIDDLE in that, 1) When detecting version inconsistencies, DECCA and RIDDLE work on the class level, a finer granularity, while LibHarmo works only on the library level. 2) DECCA and RIDDLE only focus on inconsistencies that cause DC issues, while LibHarmo also detects library version inconsistencies that do not cause DC issues for maintenance purposes. 3) LibHarmo does not support transitive library dependencies, since the inconsistencies in transitive dependencies are often out of developers' control hence cannot be harmonized.

4.4.2   TPL Updating

● *Research Background*

As mentioned before, software reuse has become a very common practice in mobile app development. Many developers use third-party libraries to facilitate the development progress. However, in-app third-party libraries could amplify their host apps' attack surface at the same time, if mobile apps contain vulnerable TPLs. Derr *et al.* [23] reported that about 70% apps with TPLs have the library outdated problem, which means even though the vulnerabilities of TPLs have been fixed, developers may still use the vulnerable versions in apps. Fast response to these vulnerable TPLs can decrease these threats in the mobile app ecosystem.

From developers' perspectives, they need to consider many factors when updating apps. For example, whether the new TPLs will affect users' experience or introduce bugs, or whether it is worth the extra effort to update a TPL, the cost/benefit ratio. Current research help reveal many unknown parts in TPL updating. We introduce their research one by one in the following subsections.

● *Existing Research*

Table 15 provides a summary of eight papers focus on the library updating. Except two papers [67], [72] propose prototypes to help developers update TPL automatically, the remaining papers mainly investigate the factors about library updating. We can see that compared to other studies, the research on library updating appeared later, where the earliest began in 2016.

Wang et al. [32] performed an empirical study of TPL updates on 806 well-maintained JAVA projects from Github. They find that only 3.5% of these projects were using the latest versions of TPLs. 14.1% projects have never updated any declared libraries and 40.8% updated at most 50% of the declared libraries. As for the update delays, only 23.1% of projects update their dependencies within a month, while over half TPLs updated at a lag of over 60 days. They also reveal that 56% of projects adopt buggy library versions. They further propose a bug-driven prototype to alert users of risky library API calls in their projects based on whether buggy library methods are invoked. They define buggy library methods as the changed methods in the bug-fixing security patch. The prototype system also attempts to evaluate the effort of updating a TPL version based on the changes (deleted or modified) of called library APIs in the project. Though they mainly focused on the TPL updates of JAVA projects, this study still provides insights into Android TPL updates.

Yasumatsu et al. [24] attempt to find how long it takes for developers to update a TPL and what are the essential factors that determine developers into update a TPL. They find that 50% of apps update the new version libraries more than 3 months after the release of new libraries. About 50% of apps still use outdated libraries for more than 10 months. Besides, they also found that popular apps get faster library updating responses from developers. Also, developers tend to update the advertising libraries faster. Furthermore, this paper not only discusses the updating of TPLs from time dimension, apps and TPLs related attributes, but also studies the reasons that can promote developers to fix vulnerable TPLs. They found that if a vulnerability has been

targeted by Google's App Security Improvement Program (ASI), developers tend to fix the vulnerability.

APPCOMMUNE [67] proposes a novel TPL sharing system in Android platform which separates TPLs from app codes and centrally managed all TPLs in a new app. Apps can still access the separated TPLs (Java & Native libraries) through a dynamic loading mechanism. The new manager app updates the TPLs with a conservative strategy to ensure stability and each TPL will be updated to the latest version. In addition to providing in-time TPL updates, APPCOMMUNE also saves storage and bandwidth by sharing TPLs.

Salza et al. [66] conducted an empirical study on 2,752 mobile apps by interviewing 73 mobile developers to reveal the updating problems of TPLs in mobile apps. They find that developers seldom update TPLs and they usually prioritize the updating of GUI-related TPLs. The main reason why they update the TPLs is to try to avoid the propagation of vulnerable libraries.

Salza et al. [69] analyzed TPL updating on the evolution history of 291 open-source Android projects on F-Droid. They find that developers merely update the old version of TPLs in their apps: only 15% are updated constantly, and about 63% are never updated. Besides, the TPL updating includes not only the upgrades but also the downgrades of TPLs when issues are induced in the previous upgrade. Also, TPLs related to UI and support tools are more likely to be updated; developers prefer to update the TPLs in high-rating apps. These findings are consistent with that of Yasumatsu *et al.* [24].

Ogawa et al. [72] proposed a prototype to automatically update TPLs with the help of an external server. When users send an app to the remote server, this server first identifies the TPLs and their versions and updates TPLs if necessary, and then generates a new apk file with the updated TPLs and sends it back to the users.

Derr et al. [75] investigated 203 app developers and attempted to find the reason why app developers do not update TPLs in Android apps. Based on their survey, we can find that developers do not choose to update TPLs because of incompatible problems, difficulties at debugging, cost/benefit ratio, and unawareness of library updating. Besides, the main reason that developers choose to update TPLs is bug fixing.

Ruiz et al. [82] conducted an empirical study regarding the ad library updates in Android apps. They conclude three reasons why developers try to update the ad libraries: 1) fix bugs, 2) add new functionalities, 3) improve personal information management.

● *Summary of TPL Updating*

Regarding current empirical study on TPL updating, these studies have analyzed TPL updating from different perspectives (e.g., the app developers, security problems). Overall, existing empirical studies have revealed most mysteries on TPL updating, while a few studies focus on how to solve in-app TPL updating automatically. We believe that if future researchers can invent more effective methods to help developers automatically update vulnerable TPLs in time, it will be a significant contribution to the community.

Based on the aforementioned studies, we can find many of the research insights share intersections. We provide the significant conclusions of existing library updating research in the following:

- Most library upgrades (85.6%) do not require modification of the host app code [75], [82].
- Most commonly-used library versions (97.8%) with a known vulnerability could be easily fixed by replacing a fixed library version [75].
- Most Apps have problems with delaying the update of TPLs [24], [69], [75].
- Developers are more willing to update GUI-related TPLs and advertisement libraries [24], [69].
- Developers seldom update the TPLs if the update parts of the TPL are not invoked by the host apps [69].
- TPL update frequency of apps with high rating score is higher than that of apps with low rating [24], [69].

The common delay response of library updating is due to the lack of timely information and incentives. We suggest that TPL vendors and app markets should help decrease the spread of vulnerable and outdated TPLs with an online platform for developers to tell if their apps may include vulnerable or outdated TPLs. At the same time, the app markets should set up a penalty system: TPL vulnerabilities will be reported to app developers should update the fixed version within a certain period; otherwise, they will be fined or their app will be deleted from the market. We believe this strategy can effectively decrease the risks of TPL vulnerabilities and outdatedness to users.

4.5 TPL Attribute Understanding

4.5.1 Research Background

TPL attributes understanding is also a critical part of TPL analysis. These papers help to understand the in-app TPLs from various perspectives. We discuss this direction by splitting current research into two parts: ad library related topics and general library related topics. Because in-app ads have a significant position in TPLs and mobile app ecosystem. For one thing, mobile advertising is essential for monetizing and ensuring that developers can earn revenue. For another, ad libraries have abundant UIs, which can directly affect users' experiences. Therefore, there are many interesting points we can explore and study.

The scope of TPL attribute-related studies covers a wide range, such as the relationship between TPLs and app maintenance [102], analyzing the relationship between apps quality/rating and TPLs [77], [97], and the impact of TPLs on repackaging detection or malware detection [34]. For the ad-related studies, previous studies revealed the UIs, permission characteristics of ad libraries [85], the symbiotic relationship understanding between different types of advertising and apps, and the collected targeting information understanding. Gui *et al.* [88] conducted an empirical study to analyze the extra cost for host apps due to the inserting ad libraries [88]. Some researchers figured out what specific information that ad collects from users and what targeting information they published to users [86], [96]. Some research focused on APIs and permissions of in-app ads [99], [100].

TABLE 16
A Summary of TPLs Attribution Understanding

Function	Tool/First Author	Year
Ad Analysis	Ahasanuzzaman et al. [63]	2020
	Ahasanuzzaman et al. [64]	2020
	MAdLens [56]	2019
	Gui et al. [77]	2017
	Book et al. [86]	2015
	Madscope [85]	2015
	Ullah et al. [96]	2014
	Book et al. [99]	2013
	Tongaonkar et al. [101]	2013
	Book et al. [100]	2013
	Vallina-Rodriguez et al. [105]	2012
Rating Analysis	Gui et al. [88]	2015
	Ruiz et al. [97]	2014
Lib Recommendation	AppLibRec [74]	2017
Miscellaneous Analysis	Li et al. [34]	2016
	Bauer et al. [102]	2012

4.5.2 Existing Research

According to our paper repository, we classify existing literature into four categories as shown in Table 16. These studies mainly discuss the impacts of TPLs on app rating [56], [77], [88], [97], the relations between the apps' quality and TPLs [56], [77], [88], [97], what information the ad libraries will collect from the target users and how this information is exploited, library recommendation [74], the impact of TPL usage on the app maintenance [102] and how in-app TPLs impact the downstream detection [34]. More detailed analysis is as follows.

● *In-app ad attributes.*

Ahasanuzzaman et al. [63] studied the integration strategies of ad libraries in 1,837 top free apps on Google Play. They classified the apps into ad-displaying apps and non-ad-displaying apps through statically analysing whether ad-displaying methods were invoked in-app activities. They discovered 22.5% of the non-ad-displaying apps integrated Google AdMob ad library for analytical purposes instead of ad displaying. They also find 57.9% of the ad-displaying apps integrate more than one ad library, which is a common practice for more popular apps. They manually analyzed 10% of the apps with multiple ad libraries, and identified four integration strategies: external-mediation, self-mediation, scattered, and mixed.

Ahasanuzzaman et al. [64] studied the evolution of 8 most popular ad libraries from Apr. 2016 to Dec. 2018. They find ad libraries are evolving continuously with a median release interval of 34 days. While the size of all but two studied ad libraries are increasing over time, three approaches were used by ad library developers trying to reduce the library size. They investigated the motivations for a new release of the ad library through manual analysis of the release notes, which include supporting new Android platform and the video ad functionality. They also proposed a reference architecture of the ad libraries which could be useful for ad library developers.

Jin et al. [56] proposed a taxonomy of mobile ads that classified them into five types: embedded, popup, notification, offerwall, and floating. They further developed MAdLens, a static analysis system for Android apps to identify ad networks and relevant APIs. Then they performed a large-scale study using MAdLens and discovered that developers tend to be conservative when embedding advertising TPLs, with 71% apps containing at most one ad network. They also find that using too many advertising TPLs in one app will annoy users and lead to a low rating.

Gui et al. [77] investigated diverse topics of ad-related complaints from users. They found most complaints about ads are UI-related topics, including the display frequency, the timing of when ads are displayed and the ads display location. They found that app developers usually attempt to give more exposure of ads to users help add chances of ad impressions and clicks; in this way, they hope earn more ad revenues. Based on this work, they pointed out that improper exposure may lead to the bad user experience and even has negative impact on app's rating.

Suman [85] tried to understand what targeting information the mobile apps sent to the ad networks (ad libraries) and how ad networks employ this information for targeting users. Thus, the author proposed a tool named MAdScope which can probe ad networks to characterize their targeting mechanism and learn the targeting behaviors of users. The author found the ad libraries usually tend to collect the users' location, device information, demographics and long-term behaviors. Besides, they found that the targeting information has a statistically significant impact on how an ad library selects ads.

Ullah et al. [96] conducted a comprehensive analysis of the ads serving mechanism of AdMob [130]. They pointed out that the level of targeting service in the Mobile Ads market is still quite low. We still have a long way to go in personalization and targeting advertising services. They suggested that future researchers and developers can make more efforts on efficient usage of collected users' date, which can help make better service to users based on the targeting service.

Besides, Book et al. [86] also investigated the ad targeting of Google AdMob library. They showed that AdMob was targeted on the application, user location, time, and real profile of users. They found that the targeting of mobile ads has some relations with the users' profile. The in-app ads were associated with the device IDs, which may bring the hidden dangers of privacy leakage.

Tongaonkar et al. [101] investigated the mobile apps' behavior patterns from the in-app ad flow. They mainly analyzed the different traffic patterns in different TPLs. They believed that is a new direction for analyzing the usage behavior of mobile apps based on ad flows.

Based on our previous analysis and Fig. 7, we can know that the ad libraries can obtain sensitive information through the system calls or the library API calls. Book et al. [99] collected 103 ad-related APIs and surveyed the relationship between these APIs and the privacy leakage of mobile apps. They found that these APIs can have access to users' personal information and device profile information. Besides, these privacy-related APIs are widely used by the top popular libraries. They also found that the system calls

and library API calls to get sensitive information are two independent processes. Book *et al.* [100] researched the ad permission ecosystem and showed the use of the permission of ad libraries. Book *et al.* adopted the static analysis by extract the APIs within the ad libraries and corresponding permissions to investigate the particular risks to user privacy and security. They investigated the use of the eight permissions of the ad libraries. They reported that ad libraries were increasingly making use of permissions which were requested by the host apps. They witnessed a growth in the usage of various dangerous permissions that could pose potential privacy risks.

Vallina-Rodriguez *et al.* [105] performed a large-scale measurement study of mobile ad traffic on an anonymized data set from a major European mobile network containing 1.7 billion traffic connections. They proposed a rule-based approach to identify and classify HTTP-based ad traffic. They found that ad traffic is a significant component of all mobile traffic, and objects in ad traffic are constantly re-downloaded. To alleviate the redundant energy overhead introduced by mobile ad traffic, they further proposed AdCache, a cached-based ad delivery system.

● *Rating Analysis.* Gui *et al.* [88] studied the hidden cost of in-app ads. They selected 21 top popular apps from Google play and analyzed the extra costs of ad libraries from five aspects: app performance, energy consumption, network usage, maintenance effort for ad-related code, and app reviews. The results show that the apps with ads consume: 48% more CPU time, 16% more energy and 79% more network data. Besides, developers need to spend extra energy to maintain the apps due to the updating of ad libraries. They also found that the complaints on ads can affect the app rating. Ruiz *et al.* [97] conducted an empirical study on the relationship between the ad libraries and the rating of apps. They found that specific ad libraries (i.e., Wooboo, Leadbolt, Airpush) have negative impacts on apps' ratings.

Considering the library rating analysis, most of the research on library rating [56], [77], [88], [97] usually involves the review analysis, as negative comments could affect other users' choices, and scores and complaints can affect the rating directly. By analyzing users' comments, we can summarize the corresponding issues and give some implications to developers.

● *Library Recommendation.* Given a new app, AppLibRec [74] could recommend third-party libraries based on the app's similar apps. AppLibRec combines the topic modeling techniques and collaborative filter component [139] to perform libraries' recommendation. It implements two steps analysis: README file (textual description) based analysis (RM-based) and Libraries based analysis (Lib-based). In the RM-based analysis, AppLibRec employs the topic model algorithm Latent Dirichlet Allocation [140] to extract topics from the README files (textual description). The libraries will be recommended based on the similar topic distribution. In Lib-based module, collaborative filtering is used to a recommendation based on the apps' similarity.

● *Lib Exterior Analysis.* Li *et al.* [34] extract 1,113 common libraries from Android apps on the Google store scale. At the same time, they clarified the impact of common libraries on the results of malware analysis, repackaged apps detection and app analysis. They also conduct an empirical investigation and evaluation of the use of common libraries in apps.

Bauer *et al.* [102] proposed a systematic approach to assess the impact of TPL usage on a project in terms of maintainability. They further provide a guidance for preselecting significant TPL candidates based on their entangledness with the project (manifested mainly by the number and scatteredness of method calls to the TPL) to reduce the assessment effort. An industrial case study indicates the effectiveness of the approach.

4.5.3 Summary

Based on our study, we think we still can do some research on targeting advertising service. On the one hand, to offer better services to customers, ad networking need to collect some profile user information. On the other hand, whether the collected user information can be used reasonably and effectively is a problem we need to think about. Previous study have pointed out that ad networks has the problem of over collecting the user information. Besides, a lot of targeting information has not been fully utilized to provided better customized services for users. Even now some fin-grained monitoring technique has been proposed to limit the over privileged issues, we still believe more empirical studies on this area should be done.

5 DISCUSSION

5.1 Threats to Validity

Paper Collection. We do not consider the following types of papers, including the books, Master or Ph.D. dissertations on TPL-related research. Instead, we search these authors' articles related to the topics we are interested in and finally include those not in our paper repository as a supplement. Even though we have tried our best to collect TPL-related papers as many as possible by following the state-of-the-art SLR methodology, our search results may still miss some relevant papers. One possible reason may be that some existing repository search engines are not very accurate; they may provide some irrelevant papers or omit some papers. The other reason may be that our keywords cannot find all the relevant papers. Considering the first reason, we reviewed all the references of our collected papers and tried to find the papers that were not in our paper repository. To mitigate the second reason, we iteratively optimized our searching keywords and tried our best to find the synonyms in order to extend the search scope and cover as many relevant papers as possible. However, it is still possible to miss some synonyms in this work. We consider using the Natural Language Processing (NLP) technique as our future work to enhance the keywords for SLR.

Besides, we implement the "major venue search" by choosing the top-tier venues from 3 fields (i.e., Software engineering, Security, and Program language). Some related research may belong to a certain conference which is not included in our search scope. Besides, some papers

may not be from top venues. To mitigate this threat, we also include some venues that are not from the top venues but they are pretty representatives, such as CODASPY and WiSec.

5.2 Threat Reasons and Arm Race

Based on the aforementioned research, we provide some useful insights and implications.

Reasons for Data Exposure in Third-Party Libraries. If we want to solve the data and privacy leakage problems, we should first understand the potential channels of data exposure. Therefore, we summarize the reasons of data exposure based on the existing literature we collected. The main reasons are as bellows: (1) The sandbox and permission mechanisms are working at the app level. Therefore, TPLs and host apps share the same storage space, permissions. TPLs can use protected APIs via permissions inherited from the host app to access sensitive information, visit the storage of the host app and get users' input from the host app. (2) To increase developers' revenue and improve the targeted advertising, some developers and advertisers will deliberately collect more user-related information (e.g., users' interests, profiles, and demographic information). The adversaries may make use of this and steal users' information by adding some malicious code into the TPLs. (3) Some third-party libs could dynamically update. It is impossible to find any security risks of dynamically update code based on the current version detection [9]. (4) Due to the `ClassLoader` functionality and reflection mechanism. Some ad libs can download the suspicious payload at the runtime from the remote servers and execute it in the context of the host apps. (5) Android allows developers to use some public/unprotected APIs without requiring any permission. These unprotected APIs can get access to platform-wide information. For example, collect the list of all apps installed on mobile phones [28]. (6) Android provides a cross-platform compatibility mechanism that allows `JavaScript` code to run in a Web-View object to invoke a set of callback functions through an interface. Through this interface, JavaScript ads can dynamically invoke other functions at the runtime, similar to Java reflection. For instance, Mobclix ad library can get access to location information by registering the `gpsStart(...)` function.

Existing Solutions for Security Risks. (1) Considering the over privilege problem of the permission mechanism, there are two main solutions to solve this problem. One is rewriting the bytecode of TPLs. By rewriting the resource access strategies and sharing functions, it can achieve de-escalation. The other solution is rewriting the Android framework. A set of new permissions are developed for TPLs alone and let each TPL run in an independent space. (2) To achieve the balance between the targeted information sending to the users by advertiser and privacy information fetching from users, researchers have proposed a framework that builds a dynamic feedback control loop which can adjust the level of privacy protection on mobile phones based on the advertising revenue. The prototype first decouples the host apps and ad libraries.

It implements a real-time monitor of the control flow of private information and can control the exposed data to advertisers. The prototype can generate real information and fuzzy information and send it to the ad servers based on the value of privacy information. (3) For the Java reflection and dynamic class loading, we usually adopt dynamic analysis to catch the potential risks.

5.3 Open Challenges and Research Directions

Based on our investigation of state-of-the-art research work on Android TPLs, we summarize some limitations in the existing work and point out some topics that are worth further investigation.

- *TPL Detection.* (1) Even though there are many TPL detection tools, most of them can reach a low recall [59]. A previous study [59] conducts an empirical study on these publicly available tools by using a well-designed dataset, which finds that these publicly available tools can only find out about half of in-app TPLs. Besides, only a few of them claim to be able to find the specific library version, but the results usually include many false positives. Based on our study, we find that the code differences of different versions are various. Some code differences among some versions are very tiny while some code differences may be very large. Library version identification has many practical usages, which can be used to find the license violation, vulnerable TPL versions, and some outdated libraries. Via identifying these vulnerable TPLs, we can inform the developers in time to replace these problematic TPLs. However, we still have a long way to go in this direction at present, some challenges such as code optimization, large-scale analysis, high precision version identification, partial import, and customize imported TPL identification. We believe that if a well-designed and accurate version identification tool is implemented, it will be meaningful and essential to industry and academia. (2) Current detection tools still cannot be effectively resilient to some sophisticated obfuscation techniques, such as class encryption and virtualization-based protection, even if a few apps use these technologies. Besides, many detection systems cannot handle the API hiding, control-flow randomization, and package flattening technique very well, and the future tool should try to find a more effective way. We suggest researchers can include richer semantic information in TPL identification, which can achieve better resiliency to code obfuscation [27], [59]. Besides, we also find the feature granularity can affect the resiliency, existing features usually include two granularities: package-level features and the class-level features. We find the class-level features can achieve better resiliency to code obfuscation. (3) Existing TPL detection tools are not good at finding emerging TPLs; we suggest that future researchers pay attention to this limitation. On the one hand, the speed of the TPL update is rapid; on the other hand, current TPL detection methods all have hysteresis. There will be more third-party libraries in the future, and it is meaningful to detect these emerging TPLs in real-time. (4) Based on our analysis, most previous tools only focus on Java library identification; researchers can try to focus on native library identification and multiple-language TPL identification.

• *Security and Privacy Issues.* (1) Current vulnerable TPL study is very limited, existing research only focuses on several typical TPLs. We think that future research directions can be studied from two branches: 1) the known vulnerabilities 2) the unknown vulnerabilities. For the known vulnerabilities, we find many work can be done here. We lack a comprehensive understanding of these vulnerabilities, their impact scope and threats and so on. It is necessary to collect these third-part libraries with vulnerabilities and conduct in-depth research on them. For the unknown vulnerabilities of TPLs, we can try to how to find these vulnerabilities and detect these vulnerabilities in both TPLs and apps. (3) Future researchers also can analyse whether the display contents of ad libraries is appropriate for specific groups. For example, the app is designed for children, the ad contents including the violence, sex, gambling and so on should be considered inappropriate.

• *TPL Isolation.* (1) More and more Android apps use some TPLs which are written in C/C++. We hope future work should pay more attention to the native code of TPLs isolation. The sophisticated tool should work on both ART or DVM mode and can effectively detect some dynamic behaves (such as Java reflection, dynamic class-load, etc.) and limit the privileges and sensitive storage access. (2) Based on our observation, the existing solutions are of little practical value because the performance overheads are usually very large. It is impossible to locate a independent process for each TPL or several TPLs on the mobile phone, because TPLs usually need to interact with host app.

• *TPL Attribution Analysis.* Based on the Section 4.5, we can find that about 70% (11/16) of existing research focused on ad analysis. (1) Without a doubt, the ad library is an essential part of TPLs. Nevertheless, other TPLs still have some unique features; we should understand them in-depth. We only collected papers from 2012 to 2020, but we have found a few papers have begun to focus on other type of TPLs, such as the analytic libraries [108], [141]. Harty *et al.* [141] conducted an empirical study on Google Firebase that is an analytic TPL. They found its logs are less pervasive and less maintained than traditional logging code. Tang *et al.* [108] analyzed 25 special TPLs, named Application Performance Management (APM) that is also a kind of analytic library. They explored the usage patterns of APMs and discovered the potential misuses of APMs. The other TPLs' attributes have not been widely analyzed, we think this may be a chance for future researchers. A large-scale TPL analysis on their features and the connection between the apps used these TPLs can be done. These can help researchers understand more about the relations between TPLs and Android apps. (2) Understanding new TPLs. We know that many TPLs are developed by Java. However, more and more TPLs are also developed by Kotlin [109] nowadays. It is also necessary to investigate these new types of TPLs. (3) The compatibility of TPLs analysis also deserves to analyze thoroughly and deeply. We find that all existing studies on TPL updating are mainly focus on the reason understanding and the effects of delay updating. In addition to the propagation of vulnerabilities, the delayed

updating of TPLs may also cause some compatibility issues (cf. Section 4.4).

6 RELATED WORK

To the best of our knowledge, this is the first literature review in the research area of Android third-party library analysis. The most previous surveys usually focus on other aspects of Android apps such as Android repackaging app detection, Android malware detection, program analysis techniques used in app analysis, and Android app testing. Based on our search, most literature review only focus on Android apps instead of Android TPLs. Till now, there is no taxonomy and comprehensive survey of third-party libraries in Android apps.

Sadeghi *et al.* [106] conducted a literature review on the assessment of Android security. They proposed a comprehensive taxonomy to classify and characterize research on 336 papers on Android security published from 2008 to the beginning of 2016. Moreover, they also highlighted the key challenges and future research direction. Based on the research, they found the gap in existing research regarding special vulnerable features of Android, such as the native and dynamically loaded code. They encouraged future researchers to pay more attention to hybrid analysis techniques instead of pure static or dynamic analysis. The survey showed that future research could consider combinations of multiple apps and Android framework.

Li *et al.* [142] conducted a survey on 59 state-of-the-art approaches of repackaged app detection. They compared different repackaging detection techniques and elaborated on current challenges in this research direction. They found that current research on repackaging detection was slowing down. Besides, they also provided a dataset of repackaged apps, which can help researchers reboot this research or replicate current approaches. Zhan *et al.* [33] also compared existing repackaged app detection tools from the implementation perspective. They evaluated the state-of-the-art tools on a uniform dataset [143] and pointed out the advantages and disadvantages of each tool. Baykara *et al.* [144] investigated malicious clone Android apps. They revealed potential threats that can affect users' experience. Finally, they provided some potential solutions for these risks.

Qiu *et al.* [145] systematically investigated the challenges of the latest deep learning-based Android malware detection and taxonomy. Rashidi *et al.* [146] discussed the existing Android security problems and existing security detection solutions from 2010 to 2015. They also gave a taxonomy of these systems and investigated their functionalities. At last, they also provided a review on the advantages and disadvantages of existing systems. Sufatrio *et al.* [147] also provided a survey and tried to classify existing security detection tools. They inspected their similarities and showed their differences. This paper also sheds light on the limitations and existing challenges. Faruki *et al.* [148] first discussed Android security mechanisms and existing problems for these security mechanisms, malware penetration, and stealth techniques. Then they analyzed the static and dynamic analysis for malware detection techniques. They compared the advantages and disadvantages of these two

analysis methods. Finally, they summarized existing systems based on their research purpose, methodology, and deployment.

Li *et al.* [51] concluded the state-of-the-art static analysis techniques of Android apps. They followed a well-defined systematic literature review methodology and collected 124 research papers. They mainly investigated the fundamental methods leveraged in related papers, the implementation methods, and relevant evaluation comparisons.

Kong *et al.* [53] 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 app size, and the fragmentation problem in the Android ecosystem. Choudhary *et al.* [149] conducted a comprehensive comparison of the primary existing test input generation tools for Android. They evaluated their advantages and disadvantages, effectiveness, and the corresponding methodology based on four criteria: ease of use, Android framework compatibility, code coverage, and fault detection capability. Their study gives a landscape of the state-of-the-art Android input test tools and provides implications for future research direction.

As we can see, there are various surveys on Android, but still lack a systematic review for third-party libraries on the Android platform. Actually, third-party libraries have become an essential part of the Android ecosystem. Furthermore, we also find many studies on Android third-party library and they focus on different perspectives. Therefore, it is necessary to conduct systematic research in this field. Thus, in this paper, we concluded the significant research achievements in third-party library analysis and conducted a detailed investigation from the following aspects: research purpose, background, application, methodologies, etc. Upon filling the gap in this research direction, we believe our work can provide a clear overview of Android TPL-related studies and inspire fellow researchers to take a step further in this direction.

7 CONCLUSION

In this paper, we conducted a systematic literature review regarding the third-party library analysis on the Android platform. We employed a well-defined SLR method to get a comprehensive paper repository that includes 74 publications for Android TPL-related analysis. We first summarized a taxonomy of existing Android TPL-related studies from four dimensions. For each category, we provided a thorough review of the existing work, compared the state-of-the-art research from different perspectives, and summarized the key insights which could shed light on the follow-up research of the corresponding research line. Finally, We discussed the open challenges and proposed new research ideas of Android TPL-related research. We believe our work can give researchers a clear overview of this direction, and inspire them to come up with more creative ideas in this area, and develop more effective approaches to solve current challenges.

REFERENCES

[1] S. A. Baset, S.-W. Li, P. Suter, and O. Tripp, "Identifying android library dependencies in the presence of code obfuscation and minimization," in *Proc. EEE/ACM 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 250–252.

[2] Statista, 2021. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] Exodus privacy, 2021. [Online]. Available: https://reports.exodus-privacy.eu.org/en/trackers/stats/

[4] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 515–516.

[5] L. Li, T. F. Bissyandé, and J. Klein, "SimiDroid: Identifying and explaining similarities in android apps," in *Proc. 16th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun.*, 2017, pp. 136–143.

[6] L. Li *et al.* "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.

[7] V. Moonsamy and L. Batten, "Android applications: Data leaks via advertising libraries," in *Proc. Int. Symp. Inf. Theory Appl.*, 2014, pp. 314–317.

[8] A. Short and F. Li, "Android smartphone third party advertising library data leak analysis," in *Proc. IEEE 11th Int. Conf. Mobile Ad Hoc Sensor Syst.*, 2014, pp. 749–754.

[9] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! Assessing user data exposure to advertising libraries on android," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[10] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 538–549.

[11] L. Yu *et al.* "PPChecker: Towards accessing the trustworthiness of android apps' privacy policies," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 221–242, Feb. 2019.

[12] M. Hammad, H. Bagheri, and S. Malek, "DelDroid: An automated approach for determination and enforcement of least-privilege architecture in android," *J. Syst. Softw.*, vol. 149, pp. 83–100, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302589

[13] M. Hammad, H. Bagheri, and S. Malek, "Determination and enforcement of least-privilege architecture in android," in *Proc. IEEE Int. Conf. Softw. Archit.*, 2017, pp. 59–68.

[14] H. Kawabata *et al.* "SanAdBox: Sandboxing third party advertising libraries in a mobile application," in *Proc. IEEE Int. Conf. Commun.*, 2013, pp. 2150–2154.

[15] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating advertisements from mobile applications in android," in *Proc. 29th Annu. Comput. Secur. Appl. Conf.*, 2013, pp. 9–18.

[16] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2015, pp. 89–103.

[17] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, Art. no 28.

[18] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in android," in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, 2012, pp. 71–72.

[19] F. Dong *et al.*, "FraudDroid: Automated ad fraud detection for android apps," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 257–268.

[20] B. Liu, S. Nath, R. Govindan, and J. Liu, "DECAF: Detecting and characterizing ad fraud in mobile app," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 57–70.

[21] J. Crussell, R. Stevens, and H. Chen, "MAdFraud: Investigating ad fraud in android applications," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2014, pp. 123–134.

[22] T. Liu *et al.*, "MadDroid: Characterizing and detecting devious ad contents for android apps," in *Proc. Web Conf.*, 2020, pp. 1715–1726.

[23] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 356–367.

[24] T. Yasumatsu, T. Watanabe, F. Kanei, E. Shioji, M. Akiyama, and T. Mori, "Understanding the responsiveness of mobile app developers to software library updates," in *Proc. 9th ACM Conf. Data Appl. Secur. Privacy*, 2019, pp. 13–24.

[25] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2169–2185.

[26] Z. Zhang, W. Diao, C. Hu, S. Guo, C. Zuo, and L. Li, "An empirical study of potentially malicious third-party libraries in android apps," in *Proc. 13th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2020, pp. 144–154.

[27] X. Zhan et al., "ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1695–1707.

[28] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2012, pp. 101–112.

[29] Y. Wang et al., "Do the dependency conflicts in my project matter?," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 319–330.

[30] Y. Wang et al., "Could I have a stack trace to examine the dependency conflict issue?," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 572–583.

[31] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 518–529.

[32] Y. Wang et al., "An empirical study of usages, updates and risks of third-party libraries in java projects," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 35–45.

[33] X. Zhan, T. Zhang, and Y. Tang, "A comparative study of android repackaged apps detection techniques," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reeng.*, 2019, pp. 321–331.

[34] L. Li, T. Bissyandé, J. Klein, and Y. L. Traon, "An investigation into the use of common libraries in android apps," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 403–414.

[35] C. Kai et al., "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale," in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 659–674.

[36] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. 2nd ACM Conf. Data Appl. Secur. Privacy*, 2012, pp. 317–326.

[37] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 56–65.

[38] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in android applications," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 415–426.

[39] L. Yu, X. Luo, C. Qian, S. Wang, and H. K. N. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 834–854, Sep. 2018.

[40] M. Li et al., "LibD: Scalable and precise third-party library detection in android markets," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 335–346.

[41] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and accurate detection of third-party libraries in android apps," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 653–656.

[42] J. Zhang, A. R. Beresford, and S. A. Kollmann, "LibID: Reliable identification of obfuscated third-party android libraries," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 55–65.

[43] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "ORLIS: Obfuscation-resilient library detection for android," in *Proc. IEEE/ACM 5th Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 13–23.

[44] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–10.

[45] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[46] ACM Digital Library, 2021. [Online]. Available: https://dl.acm.org/

[47] IEEE Xplore Digital Library, 2021. [Online]. Available: https://ieeexplore.ieee.org/Xplore/

[48] SpringerLink, 2021. [Online]. Available: https://link.springer.com/

[49] ScienceDirect, 2021. [Online]. Available: https://www.sciencedirect.com/

[50] "The network and distributed system security symposium," 1992–2021. [Online]. Available: https://www.ndss-symposium.org/about/

[51] L. Li et al., "Static analysis of android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, 2017.

[52] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, Apr. 2021.

[53] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 45–66, Mar. 2019.

[54] B. He, H. Xu, L. Jin, G. Guo, Y. Chen, and G. Weng, "An investigation into android in-app ad practice: Implications for app developers," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 2465–2473.

[55] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Automated detection and classification of third-party libraries in large scale android apps," *J. Softw. Chin.*, vol. 28, no. 6, pp. 1373–1388, 2017, doi: 10.13328/j.cnki.jos.005221.

[56] L. Jin, B. He, G. Weng, H. Xu, Y. Chen, and G. Guo, "MAdLens: Investigating into android in-app ad practice at API granularity," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 1138–1155, Mar. 2021.

[57] M. Li et al., "Large-scale third-party library detection in android markets," *IEEE Trans. Softw. Eng.*, vol. 46, no. 9, pp. 981–1003, Sep. 2020.

[58] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin, "FLEXDROID: Enforcing in-app privilege separation in android," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[59] X. Zhan et al., "Automated third-party library detection for android applications: Are we there yet?," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 919–930.

[60] W. Tang, P. Luo, J. Fu, and D. Zhang, "LibDx: A cross-platform and accurate system to detect third-party libraries in binary code," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evol. Reeng.*, 2020, pp. 104–115.

[61] Z. Zhang, W. Diao, C. Hu, S. Guo, C. Zuo, and L. Li, "An empirical study of potentially malicious third-party libraries in android apps," in *Proc. 13th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2020, pp. 144–154.

[62] J. Xu and Q. Yuan, "LibRoad: Rapid, online, and accurate detection of TPLs on android," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2020.3003336.

[63] M. Ahasanuzzaman, S. Hassan, and A. E. Hassan, "Studying ad library integration strategies of top free-to-download apps," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.2983399.

[64] M. Ahasanuzzaman, S. Hassan, C.-P. Bezemer, and A. E. Hassan, "A longitudinal study of popular ad libraries in the Google Play store," *Empir. Softw. Eng.*, vol. 25, pp. 824–858, 2020.

[65] G. Chen, W. Meng, and J. Copeland, "Revisiting mobile advertising threats with MAdLife," in *Proc. World Wide Web Conf.*, 2019, pp. 207–217.

[66] P. Salza, F. Palomba, D. D. Nucci, A. D. Lucia, and F. Ferrucci, "Third-party libraries in mobile apps when, how, and why developers update them," *Empir. Softw. Eng.*, vol. 25, pp. 2341–2377, 2019.

[67] B. Li, Y. Zhang, J. Li, R. Feng, and D. Gu, "APPCOMMUNE: Automated third-party libraries de-duplicating and updating for android apps," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reeng.*, 2019, pp. 344–354.

[68] Y. Zhang et al., "Detecting third-party libraries in android applications with high precision and recall," in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 141–152.

[69] P. Salza, F. Palomba, D. D. Nucci, C. D'Uva, A. D. Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *Proc. IEEE/ACM 26th Int. Conf. Program Comprehension*, 2018, pp. 255–265.

[70] F. Dong, H. Wang, L. Li, Y. Guo, G. Xu, and S. Zhang, "How do mobile apps violate the behavioral policy of advertisement libraries?," in *Proc. 19th Int. Workshop Mobile Comput. Syst. Appl.*, 2018, pp. 75–80.

[71] H. Han, R. Li, and J. Tang, "Identify and inspect libraries in android applications," *Wireless Pers. Commun.* vol. 103, pp. 491–503, 2018.

[72] H. Ogawa, E. Takimoto, K. Mouri, and S. Saito, "User-side updating of third-party libraries for android applications," in *Proc. 6th Int. Symp. Comput. Netw. Workshops*, 2018, pp. 452–458.

[73] T. Watanabe *et al.*, "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 14–24.

[74] H. Yu, X. Xia, X. Zhao, and W. Qiu, "Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation," in *Proc. 9th Asia-Pacific Symp. Internetware*, 2017, pp. 1–6.

[75] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2187–2200.

[76] J. Zhan, Q. Zhou, X. Gu, Y. Wang, and Y. Niu, "Splitting third-party libraries' privileges from android apps," in *Proc. Australas. Conf. Inf. Secur. Privacy*, 2017, pp. 80–94.

[77] J. Gui, M. Nagappan, and W. G. J. Halfond, "What aspects of mobile ads do users care about? An empirical study of mobile in-app ad reviews," pp. 1–10, 2017, arXiv:1702.07681.

[78] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[79] F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, "Stay in your cage! A sound sandbox for third-party libraries on android," in *Proc. 21st Eur. Symp. Res. Comput. Secur.*, 2016, pp. 458–476.

[80] K. Chen *et al.*, "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and iOS," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 357–376.

[81] C. Soh, H. B. K. Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang, "LibSift: Automated detection of third-party libraries in android applications," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf.*, 2016, pp. 41–48.

[82] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, and S. Dienst, "Analyzing ad library updates in android apps," *IEEE Softw.*, vol. 33, no. 2, pp. 74–80, Mar./Apr. 2016.

[83] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[84] M. Wei, D. Ren, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-app ads," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[85] S. Nath, "MAdScope: Characterizing mobile in-app targeted ads," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2015, pp. 59–73.

[86] T. Book and S. W. Dan, "An empirical study of mobile ad targeting," vol. abs/1502.06577, pp. 1–14, 2015.

[87] A. Paturi, P. G. Kelley, and S. Mazumdar, "Introducing privacy threats from ad libraries to android users through privacy granules," in *Proc. Netw. Distrib. Syst. Secur. Symp. Workshop Usable Secur.*, 2015, pp. 1–10.

[88] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond, "Truth in advertising: The hidden cost of mobile ads for software developers," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 100–110.

[89] G. Cho, J. Cho, Y. Song, and H. Kim, "An empirical study of click fraud in mobile advertising networks," in *Proc. 10th Int. Conf. Availability Rel. Secur.*, 2015, pp. 382–388.

[90] M. Kühnel, M. Smieschek, and U. Meyer, "Fast identification of obfuscation and mobile advertising in mobile malware," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 214–221.

[91] A. Narayanan, L. Chen, and C. K. Chan, "AdDetect: Automated detection of android ad libraries using semantic analysis," in *Proc. IEEE 9th Int. Conf. Int. Sensors Sensor Netw. Inf. Process.*, 2014, pp. 1–6.

[92] W. Yang, J. Li, Y. Zhang, J. S. Y. Li, and D. Gu, "APKLancet: Tumor payload diagnosis and purification for android applications," in *Proc. 9th ACM Symp. Inf. Comput. Commun. Secur.*, 2014, pp. 483–494.

[93] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy*, 2014, pp. 25–36.

[94] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu, "Duet: Library integrity verification for android applications," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 141–152.

[95] M. Sun and G. Tan, "NativeGuard: Protecting android applications from third-party native libraries," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 165–176.

[96] I. Ullah, R. Boreli, M. A. Kaafar, and S. S. Kanhere, "Characterising user targeting for in-app mobile ads," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2014, pp. 547–552.

[97] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Impact of ad libraries on ratings of android mobile apps," *IEEE Softw.*, vol. 31, no. 6, pp. 86–92, Nov./Dec. 2014.

[98] R. Bhoraskar *et al.*, "Brahmastra: Driving apps to test the security of third-party components," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 1021–1036.

[99] T. Book and D. S. Wallach, "A case of collusion: A study of the interface between ad libraries and their apps," in *Proc. 3rd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2013, pp. 79–86.

[100] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," in *Proc. Mobile Secur. Technol.*, 2013. [Online]. Available: http://arxiv.org/abs/1303.0857

[101] A. Tongaonkar, S. Dai, and D. S. A. Nucci, "Understanding mobile app usage patterns using in-app advertisements," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, 2013, pp. 63–72.

[102] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 483–492.

[103] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: Balancing privacy in an ad-supported mobile application market," in *Proc. 12th Workshop Mobile Comput. Syst. Appl.*, 2012, pp. 1–6.

[104] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Proc. Workshop Mobile Secur. Technol.*, 2012, pp. 1–10.

[105] N. Vallina-Rodriguez *et al.*, "Breaking for commercials: Characterizing mobile advertising," in *Proc. Internet Meas. Conf.*, 2012, pp. 343–356.

[106] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 492–530, Jun. 2017.

[107] Y. Tang *et al.*, "Demystifying application performance management libraries for android," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 682–685.

[108] Y. Tang *et al.*, "A systematical study on application performance management libraries for apps," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2021.3077654.

[109] Kotlin, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Kotlin_(programming_language)

[110] L. Mario, H. Andrew, and P. Denys, "On automatically detecting similar android apps," in *Proc. IEEE 24th Int. Conf. Program Comprehension*, 2016, pp. 1–10.

[111] M. Sun, M. Li, and J. C. S. Lui, "DroidEagle: Seamless detection of visually similar android apps," in *Proc. 8th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2015, pp. 1–12.

[112] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2012, pp. 62–81.

[113] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.

[114] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[115] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina, and E. Moser, "FSquaDRA: Fast detection of repackaged applications," in *Proc. IFIP Annu. Conf. Data Appl. Secur. Privacy*, 2014, pp. 130–145.

[116] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable detection of semantically similar android applications," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 182–199.

[117] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A scalable and accurate two-phase approach to android app clone detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 71–82.

[118] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy*, 2013, pp. 185–196.

[119] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[120] Baksmali, 2021. [Online]. Available: https://github.com/JesusFreke/smali

[121] Androguard, 2020. [Online]. Available: https://github.com/androguard/androguard

[122] Soot, 2021. [Online]. Available: https://github.com/Sable/soot

[123] Google android GMS, 2021. [Online]. Available: https://mvnrepository.com/artifact/com.google.android.gms

[124] Google android library, Dec. 2015. [Online]. Available: https://mvnrepository.com/artifact/com.google.android/android

[125] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "CENTRIS: A precise and scalable approach for identifying modified open-source software reuse," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 860–872.

[126] DroidKungFu, 2021. [Online]. Available: https://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml

[127] DroidBox, Sep. 2015. [Online]. Available: https://github.com/pjlantz/droidbox

[128] TaintDroid, Feb. 2014. [Online]. Available: http://www.appanalysis.org/

[129] Android debug bridge, 2021. [Online]. Available: https://developer.android.com/studio/command-line/adb

[130] Google Admob, 2021. [Online]. Available: https://admob.google.com/home/

[131] Monkey, 2021. [Online]. Available: https://developer.android.com/studio/test/monkey

[132] NVD, 2021. [Online]. Available: https://nvd.nist.gov/

[133] Source Clear, 2021. [Online]. Available: https://sca.analysiscenter.veracode.com/vulnerability-database/search#query=language:java

[134] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: Lightweight provenance for smart phone operating systems," in *Proc. 20th USENIX Conf. Secur.*, 2011, Art. no. 23.

[135] AIDL, 2019. [Online]. Available: https://developer.android.com/guide/components/aidl

[136] JNI, 2020. [Online]. Available: https://developer.android.com/training/articles/perf-jni

[137] Java reflection, 2021. [Online]. Available: http://tutorials.jenkov.com/java-reflection/index.html

[138] Dynamic code execution, 2021. [Online]. Available: https://gerardnico.com/lang/java/dynamic#load_and_run_the_class

[139] R. Burke, "Hybrid recommender systems: Survey and experiments," *User Model. User-Adapted Interact.*, vol. 12, no. 4, pp. 331–370, Nov. 2002.

[140] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[141] J. Harty, H. Zhang, L. Wei, L. Pascarella, M. Aniche, and W. Shang, "Logging practices with mobile analytics: An empirical study on firebase," in *Proc. IEEE/ACM 8th Int. Conf. Mobile Softw. Eng. Syst.*, 2021, pp. 56–60

[142] L. Li, T. F. Bissyande, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, Apr. 2021.

[143] AndroZoo, 2016. [Online]. Available: https://androzoo.uni.lu/

[144] M. Baykara and E. Çolak, "A review of cloned mobile malware applications for android devices," in *Proc. 6th Int. Symp. Digit. Forensic Secur.*, 2018, pp. 1–5.

[145] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Comput. Surv.*, vol. 53, no. 6, pp. 1–36, 2020.

[146] B. Rashidi and C. Fung, "A survey of android security threats and defenses," *J. Wireless Mob. Netw. Ubiquitous Comput. Dependable Appl.*, vol. 6, pp. 3–35, 2015.

[147] Sufatrio, J. Tan, C. Wei, and L. Thing, "Securing android: A survey, taxonomy, and challenges," *ACM Comput. Surv.*, vol. 47, pp. 1–45, 2015.

[148] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, Second Quarter 2015.

[149] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 429–440.

Xian Zhan received the BEng degree in computer science and technology from Wuhan University, Hubei, China, and the doctoral degree under the supervision of Prof. Xiapu Luo, from the Department of Computing, Hong Kong Polytechnic University, Hong Kong. Now, she is a postdoctoral fellow at Hong Kong Polytechnic University. Her research interests include program analysis, mobile privacy and security, Android analysis, third-party library and open-source software analysis, and machine learning. She got an ACM SIGSOFT Distinguished Paper Award at ICSE 2021. Her research work usually target at top-tier conferences and journals, including USENIX Security, CCS, ICSE, ASE, and *IEEE Transactions on Software Engineering*.

Tianming Liu received the MS degree in cyberspace security from the Beijing University of Posts and Telecommunications, Beijing, China, in 2020. He is currently working toward the PhD degree in the Faculty of Information Technology, Monash University, Melbourne, Australia. His research interests include mobile security and Android application analysis.

Lingling Fan received the BEng and PhD degrees in computer science from East China Normal University, Shanghai, China, in June 2014 and June 2019, respectively. She is an associate professor with the College of Cyber Science, Nankai University, China. In 2017, she joined Nanyang Technological University (NTU), Singapore as a research assistant and then had been as a research fellow of NTU since 2019. Her research focuses on program analysis and testing, software security, and Android and application analysis and testing. She got two ACM SIGSOFT Distinguished Paper Awards at ICSE 2018 and ICSE 2021. For more information, please visit https://lingling-fan.github.io/.
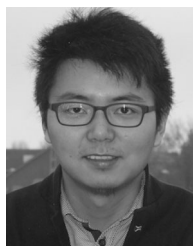
Li Li received the PhD degree in computer science from the University of Luxembourg, Esch-sur-Alzette, Luxembourg, in 2016. He is a senior lecturer (a.k.a. associate professor) and a PhD supervisor with Monash University, Australia. Prior to joining Monash University, he spent 1.5 years as a research associate with the Serval Group, SnT, University of Luxembourg. His research interests include the fields of Android security and reliability, static code analysis, machine learning, and deep learning. He received an ACM Distinguished Paper Award at ASE 2018, a FOSS Impact Paper Award at MSR 2018 and a Best Paper Award at the ERA track of IEEE SANER 2016. He is an active member of the software engineering and security community serving as reviewers or co-reviewers for many top-tier conferences and journals such as ASE, ICSME, SANER, *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering and Methodology*, *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Dependable and Secure Computing*, *ACM Transactions on Privacy and Security*, *Empirical Software Engineering*, *Journal of Systems and Software*, *Information and Software Technology*, etc. For more information, please visit http://lilicoding.github.io.

Sen Chen (Member, IEEE) received the PhD degree in computer science from the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in June 2019. He is an associate professor with the College of Intelligence and Computing (School of Cybersecurity), Tianjin University, China. Before that, he was a research assistant professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Previously, he was a research assistant of NTU from 2016 to 2019 and a research fellow from 2019–2020. His research focuses on security and software engineering such as mobile security, AI security, open-source security, and intelligent development and testing. He has published broadly in top-tier security (IEEE S&P, USENIX Security, CCS, *IEEE Transactions on Information Forensics and Security*, and *IEEE Transactions on Dependable and Secure Computing*) and software engineering venues including ICSE, FSE, ASE, *ACM Transactions on Software Engineering and Methodology*, and *IEEE Transactions on Software Engineering*. For more information, please visit https://sen-chen.github.io/.

Yang Liu received the bachelor's and PhD degrees from the National University of Singapore, Singapore, in 2005 and 2010, respectively. In 2012, he joined Nanyang Technological University as a nanyang assistant professor. He is currently a full professor, director of the Cybersecurity Lab, program director of HP-NTU Corporate Lab, and deputy director of the National Satellite of Excellence of Singapore. In 2019, he received the University Leadership Forum chair professorship at NTU. He specializes in software engineering, cybersecurity, and artificial intelligence. His research has bridged the gap between the theory and practical usage of program analysis, data analysis and AI to evaluate the design and implementation of software for high assurance and security. By now, he has more than 400 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA fellowship, TRF fellowship, Tan Chin Tuan fellowship, Nanyang Research Award 2019, ACM distinguished speaker, NRF Investigatorship, and 15 best paper awards and one most influence system Award in top software engineering conferences like ASE, FSE, and ICSE. For more information, please visit http://www.ntu.edu.sg/home/yangliu/.

Xiapu Luo received the PhD degree in computer science from the Hong Kong Polytechnic University, Hong Kong. He is an associate professor with the Department of Computing and an associate researcher with Shenzhen Research Institute, Hong Kong Polytechnic University. He was a postdoctoral research fellow with the Georgia Institute of Technology. His research focuses on smartphone security and privacy, network security and privacy, and Internet measurement. For more information, please visit https://www4.comp.polyu.edu.hk/~csxluo/.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.