# Detection and Privacy Leakage Analysis of Third-Party Libraries in Android Apps

Xiantong Hao, Dandan Ma, and Hongliang Liang(✉)

TSIS Lab., Beijing University of Posts and Telecommunications, Beijing, China
{xiantonghao,ma21,hliang}@bupt.edu.cn

**Abstract.** Third-party libraries (TPL) make Apps' functionality diversified but introduce severe security risks. Precisely detecting and analyzing TPLs is challenging because their code usually is not publicly available or obfuscated. Prior studies do not perform well in detecting closed-source or obfuscated TPLs and analyzing their privacy risks.

In this paper, we propose a novel approach to detect TPLs in Android Apps and analyze privacy leakage caused by TPLs. The key idea of our approach is that it leverages the call frequencies of different types of APIs as features and conducts a clustering algorithm on these features, our approach works well on obfuscated TPLs, especially those with dead code removal and control flow randomization. We also analyze whether there is privacy leakage in a TPL by dynamically instrumenting privacy-related APIs and inspecting its call stack. We implement our approach in a tool named Libmonitor and evaluate it on 162 obfuscated Apps and 217 real-world Apps. Experimental results show that Libmonitor outperforms two state-of-the-art tools on two datasets. With obfuscated TPLs, Libmonitor improves 394.08% over Libradar and 26.32% over LibD on F1 metric, respectively. With closed-source TPLs, Libmonitor increases 18.66% over Libradar and 150.15% over LibD on F1 metric, respectively. Besides, Libmonitor found 5809 pieces of privacy leakage risks caused by 152 TPLs in 64 real-world Apps.

**Keywords:** Android · Third-party library detection · Clustering · Privacy leakage analysis

## 1 Introduction

Now Android is the most popular mobile operating system [1] while developing a practical Android application is getting more intricate. Today's Apps require many additional functions to assist, such as user portraits, personalized recommendations, socialization, etc. To cope with the ever-increasing demand for App development, existing developers on the Android platform integrate third-party libraries (TPL) into their Apps. These TPLs have complex sources, diverse types, and different functions, including advertising, location access, mobile payment, etc. Related research work [2] shows that each App contains an average of 1.59 TPLs. In extreme circumstances, some Apps use over 30 different TPLs. On

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2023
Published by Springer Nature Switzerland AG 2023. All Rights Reserved
F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 569–587, 2023.
https://doi.org/10.1007/978-3-031-25538-0_30

average, over 60% of sub-packages in an App are from third-party libraries [3]. The TPL can reuse code, reduce development time, improve App quality, and let developers focus on the architecture of the App. However, improper use of TPL will introduce the following problems:

The first problem is the introduction of vulnerable code. If a developer uses a TPL with a vulnerability and does not update it in time, the TPL will introduce the vulnerable code into the App. For example, the SlowMist Security Team published a report in 2018 [4], there is an XSS 0-day vulnerability in a JavaScript library named *TradingView*, which can bypass the Cloudflare defense mechanism. Wu et al. [5] found 13 popular TPLs have open ports and 61.8% open-port actions in their App dataset are caused by TPL. Almanee et al. [6] conducted a study on 200 free Apps on Google Play, and tracked the version iterations of Apps using third-party native libraries between May 2013 and September 2020. The study found that 53 Apps use malicious versions of third-party libraries with known CVEs, of which 14 Apps have not updated the vulnerable versions in 2020. Since multiple Apps may use a same TPL, vulnerabilities in the third-party library will have an enormous impact, like viruses.

The second problem is the leakage of private information. TPLs may collect user information to achieve their functions, but sometimes, they may collect unauthorized or unnecessary data. For example, TPLs provided by Taomike and Baidu have been exposed to security vulnerabilities. They secretly monitor users' behaviors and upload sensitive information to remote servers [7]. According to a survey in 2019 [8], over 40% Apps collect users' personal information beyond their own function requirements.

Unfortunately, developers rarely list third-party libraries in their Apps. Some open-source TPLs can be accessed from Maven [9], GitHub [10], and other websites, however, commercial Apps may use closed-source TPLs. To detect TPLs in mobile Apps, several prior efforts [2,11,12] use the whitelist approach. However, closed-source TPLs and the growing of new TPLs make it difficult for researchers to maintain a complete whitelist. In addition, code obfuscation also makes the whitelist approach ineffective. Some studies [14,16,33] apply similarity comparison techniques. LibID [13] has two library detection schemes, using the textual representation in the basic block and class dependency as features, but it cannot resist dead code removal and control flow randomization. Libpecker [32] uses signature matching to get a similarity score between a library and an App. By internal class dependencies in the library, Libpecker generates strict signatures for each class. To resist the customization and removal of library code as much as possible, it uses fuzzy match when calculating library similarity. However, Libpecker requires pre-collection of TPLs to establish a feature database, so it can't detect closed-source TPL and is time-consuming.

Other studies leverage feature clustering technique. Libradar [15] uses feature clustering, but it requires an exact match of hash values in features, so it cannot deal with TPLs obfuscated with dead code removal. LibD [17] needs to traverse the decompiled code to build the directory tree and homogeny graphs, which is time-consuming and cannot detect TPLs obfuscated with control flow randomization.

As for privacy leakage analysis in Apps, some research efforts use taint analysis technique. For example, Taintdroid [18] and Taintman [19] work on Dalvik and ART, respectively. They use dynamic taint analysis to track data flow, but they need to modify system code and hence are hard to migration. Besides, their analysis target is a whole App instead of a TPL. Some studies analyze network traffic to find privacy leakage. For instance, He et al. [20] capture network packets to match private information and trace back to find the APIs that leak privacy. VULPIX [21] also collects network traffic to match private information, but it only considers network interfaces and ignores other interfaces such as logcat and short-message-service. These methods are either heavy weighted or incomplete for analyzing privacy leakage in TPLs.

In this paper, we propose a new clustering-based approach to detect TPLs in Android Apps and perform privacy leakage analysis on TPLs. We classify Android APIs into 15 categories and extract API call frequencies for every category as features. These features can resist control flow randomization because each API name is persistent in any call place and call time. Considering that different Apps may import a same TPL, we use fuzzy match to cluster feature vectors into different clusters and use cluster prediction to detect TPLs, which means the feature vectors in a cluster don't have to be exactly the same during cluster partitioning. In this way, the fuzzy match is not sensitive to dead code removal. We use dynamic instrumentation to monitor sensitive APIs and perform privacy leakage analysis. Dynamic instrumentation is light-weighted and hence easily applied on most sensitive interfaces like logcat, network, message and clipboard.

In summary, our contributions are as follows:

- We propose a feature matching approach for detecting TPLs. We divide Android APIs into 15 categories and extract API call frequency for each category as features, then perform fuzzy match to cluster feature vectors and detect TPLs by cluster prediction. Our approach can resist code obfuscation, especially dead code removal and control flow randomization.
- We develop a TPL privacy leakage analysis approach, which uses dynamic instrumentation to monitor source and sink APIs, then performs information match and call stack analysis.
- We implement the proposed approaches in a tool Libmonitor and evaluate it on 162 obfuscated Apps and 217 real-world Apps respectively. Experimental results show that Libmonitor outperforms two state-of-the art tools, Libradar and Libd. Moreover, Libmonitor found 5809 pieces of privacy leakage risks caused by 152 TPLs in 64 Apps.

## 2 Background

### 2.1 Code Obfuscation

Code obfuscation is the act of creating source or machine code that is difficult for humans to understand without destroying behaviors of a program. The purpose

of code obfuscation is to prevent code from being tampered with or reverse-engineered. Code obfuscation is common in Android Apps, for example, Dong et al. [22] found that 43% of Google Play Apps and 73% of third-party market Apps have obfuscated code. Currently common obfuscation strategies in Android apps are: package flattening, identifier renaming, string encryption, control flow randomization, and dead code removal, the later two of which are difficult to deal with when detecting TPLs [28]. Existing well-known tools are Dasho and Proguard. Dasho provides all strategies above while Proguard supports the former two strategies.

### 2.2 Cluster Algorithm

Cluster analysis is the task to group a set of objects so that objects in the same cluster are more similar or more closely related to each other. Clustering is a common statistical data analysis technique. According to the cluster criteria, clustering algorithms can be divided into the following categories:

- *Hierarchical cluster*: samples are more related to objects at closer distances.
- *Centroid-based cluster*: each cluster is marked by a center point and the number of clustering centers needs to be determined in advance.
- *Density-based cluster*: clusters are defined as regions with a higher density than the other objects.

The advantage of cluster analysis over other machine learning algorithms is that cluster analysis is an unsupervised algorithm and is simple to operate. Cluster analysis is more cost-effective and time-efficient than other probability sampling methods, especially for widely distributed samples.

## 3 Design

### 3.1 Overview

In this section, we present Libmonitor, a third-party library detection and privacy leakage analysis system. Libmonitor consists of two modules: TPL detection and privacy leakage analysis.

We observe that a third-party library may be used in several Apps, so we can detect closed-source or obfuscated TPLs by clustering features of numerous Apps. The workflow of TPL detection is shown in Fig. 1. First, a large dataset of real-world Apps is collected from Android App markets, and API call frequencies of different types are extracted as features for each App. Then clustering of fuzzy match is performed, which means that the features are not exactly equal in a cluster, so our system can resist dead code removal.

The workflow of privacy leakage analysis is shown in Fig. 2. We get private information, e.g., AdvertiserID, AndroidID, IMEI, MacAddress, DeviceType, TimeZone, KernelVersion, through android debug bridge (adb) and dynamic instrumentation. They are encoded or encrypted into a processed information

list (PIL). Then we adopt a dynamic instrumentation technique to separately monitor and record the invocations of private information related APIs, i.e., source and sink APIs. Finally, we match PIL with API invocation record and perform call stack analysis to generate privacy leakage analysis report for those TPLs in the TPL list, which is generated in TPL detection module.

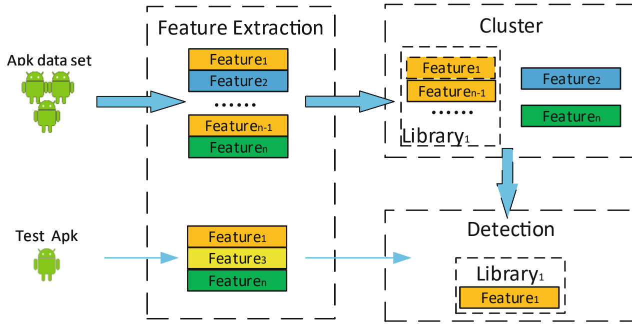


**Fig. 1.** Workflow of Third-party library detection

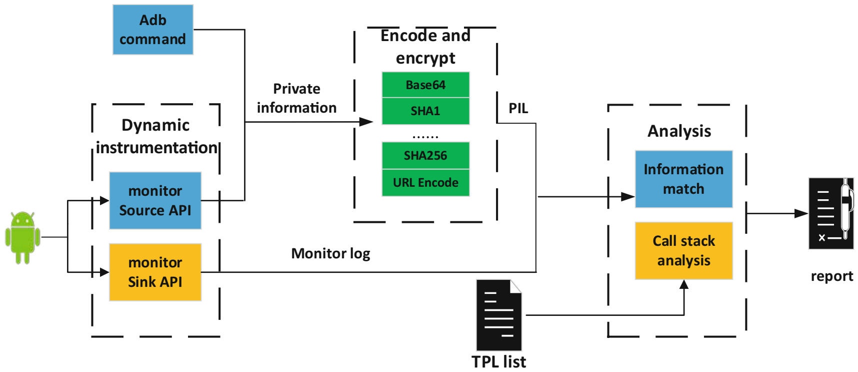


**Fig. 2.** Workflow of privacy leakage analysis

### 3.2 TPL Detection

**Collecting and Unpacking Apps.** We collect 10000 Apps from Androzoo [23], which contains millions of Apps from various App markets. Since Apps on Google play are more secure and popular, we mainly downloaded Google play Apps. After unpacking these Apps with apktool [24], we can analyze them from multiple aspects. Without source code, we need to analyze *smali* code or *dex* file. Generally, the *smali* code is more friendly for people to read. However, conversion to *smali* files is time-consuming, and the analysis of *smali* files is text analysis, which loses some structured information in binary, therefore we unpack Apps to get dex file.

**Feature Extraction.** In this section, we firstly categorize Android APIs into 15 categories according to the functions of the Android APIs, as shown in Table 1, then we parse *dex* files in an App and count the call frequency of each type of API in the App according to Algorithm 1. The call frequencies of these APIs rarely change in the process of obfuscation, and thus can represent the behaviors (or features) of TPLs in the App. As shown in Algorithm 1, we go through all *dex* files and *class* files to extract features. *AT* is a large array that maps an API to its API type in total 15 types, these types are represented by numbers from 0 to 14 respectively, thus the feature vector of each node is an array of length 15. We compute the feature vectors of child nodes in Java hierarchy tree and gradually update the feature vectors of their parent nodes. Finally, we integrate them and store them in the database. The total API list can be obtained from the Android Developer website [25].

**Table 1.** Classification of android APIs

| Type | Description |
|---|---|
| io | Write to and read file |
| Security | Security verification |
| Deviceinfo | Get IMEI, deviceId, etc. |
| Network | Related to network |
| Location | Get location |
| UI | UI and resource file |
| Hardware | Control of hardware |
| Multimedia | Radio, vedio, etc. |
| Application | Package manager and four components |
| Database | Manage database |
| Util | Universal method |
| System | Scheduling and Parallelization, etc. |
| Xml | Parse xml file |
| Time | Timezone, date, etc. |
| Datatype | Object of Int, String, json, etc. |

**Feature Clustering.** Common clustering algorithms include hierarchical clustering, center-based clustering, spectral clustering, etc. The center-based algorithm needs to determine the number of cluster centers in advance, which is unknown to us. Spectral clustering is used to deal with the division of nodes that are related to each other, but we regard each package as an individual node in our system, therefore this algorithm is not appropriate. Considering these facts, we adopt the hierarchical clustering method.

**Algorithm 1.** Extracting features from Dex files

**Require:** $DF$: dex files; $AT$: The array that maps an API to its API type in total 15 types.
**Ensure:** $Dict$: features dictionary;

```
1: dictionary initialize
2: for each file in DF do
3:    Parse and construct dexfile object O
4:    for each class C in O do
5:       C_f=[0]*15
6:       for each method in C.methods do
7:          C_f[AT[method]]+=1
8:       end for
9:       add C_name and C_f to Dict
10:      while C is not root directory do
11:         tmp = C
12:         C = C.parent
13:         if C in Dict then
14:            Dict[C]+ = Dict[tmp]
15:         else
16:            Dict[C] =Dict[tmp]
17:         end if
18:      end while
19:   end for
20: end for
```

Android App code has a certain package structure, the obfuscation of identifier name will not affect the package structure, and the feature of one third-party library in different Apps are the same, for example, when *com.google.gson* is obfuscated into *com.a.b*, its feature value will not change. Even with dead code removal, the feature of TPL is similar within limits. Based on the above observations, we cluster the features at the same package structure level when implementing the feature cluster.

First, we read the feature vector from the feature database built in Sect. 3.2. There may be some features that are too small or simple, which can't represent a TPL. Considering the existence of these noise points, we filter the feature vector data to remove these invalid feature vectors. Then we normalize the feature vector data and use the Birch algorithm [26] to cluster these data. We only keep the clusters each of which has more sample points than a certain threshold. Finally, we build the map between a cluster label and a TPL according to package name that is not obfuscated in the cluster and persist the cluster model.

**TPL Detection.** After feature clustering, we get lots of potential third-party libraries in our cluster model. To detect TPLs in an App, we first extract its features and normalize the feature vector in the same way as Sect. 3.2. Then we conduct cluster prediction based on the models established in the clustering process. Our clustering model uses the hierarchical clustering strategy with

a threshold, which means that the cluster prediction process performs fuzzy matching and is resistant to dead code removal. According to the map between labels and TPLs, we get a list of TPLs used in an App.

### 3.3 Privacy Leakage Analysis

**PIL Collection.** Compared with the user's personal information, the device information is fixed. We can use adb and instrumentation of the source APIs to obtain device-related information. Considering that an App may encrypt or encode the device information when transmitting it, we process acquired device information using common encoding and encryption algorithms, and finally, the processed information list (PIL) is obtained.

**Dynamic Instrumentation.** Sensitive APIs can be divided into source APIs and sink APIs as shown in Table 2. We record the return values of the source APIs called by an App and the parameters of the sink APIs as logs, and then collect the call stack information for subsequent analysis. We leverage dynamic instrumentation technique to monitor sensitive APIs as shown in Fig. 3. Specifically, dynamic instrumentation is composed of an Android device and a PC-side controller. The device is installed with frida server, a typical instrumentor for Android, and the PC-side controller is implemented as some scripts in JavaScript which are injected into the Android device. We use Monkey [27] to simulate a user's operations on the App. During the App's execution, the scripts record the running log for later analysis.

**Table 2.** Example source APIs and sink APIs

| Example | Type |
|---|---|
| android.location.LocationManager.getLastKnownLocation | Source |
| android.content.ClipboardManager.getPrimaryClip | Source |
| android.content.ContentResolver.query | Source |
| android.telephony.TelephonyManager.getDeviceId | Source |
| android.util.Log.i | Sink |
| okhttp3.OkHttpClient.newCall | Sink |
| android.telephony.SmsManager.sendDataMessage | Sink |
| android.util.LogPrinter.println | Sink |
| android.content.ClipboardManager.setPrimaryClip | Sink |

**Private Information Matching.** Private information can be divided into two categories: device information and user-specific personal information. The former includes deviceID, WifiInfo, etc., while the latter mainly includes user name, age, home address, etc. Part of the device information is fixed and can be obtained in advance, and another part of the information such as location information can

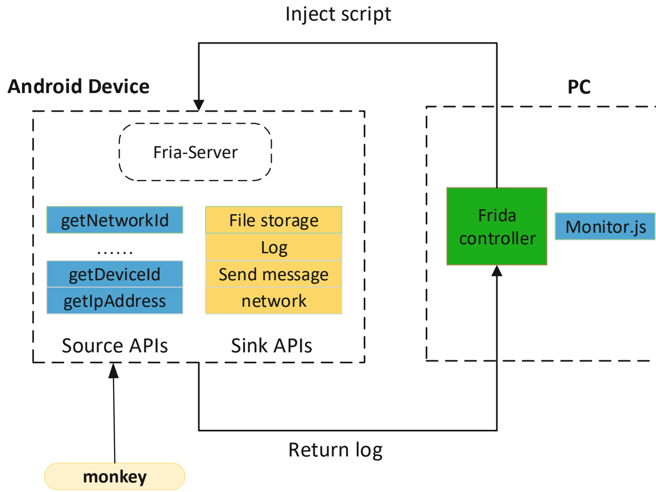


**Fig. 3.** Dynamic instrumentation

be obtained by instrumentation of the source APIs, this information is processed in Sect. 3.3 and we can get PIL. For logs of sink APIs, we match logs with PIL to determine whether there is private information flowing out of the sink APIs. For logs of source APIs, as long as source APIs are called, we define that the private information is matched but with low risk. Device information is generally a specific string, so it can be matched directly in the monitor log. For user-specific personal information, it is not fixed information but has a certain format, we can use regular expressions to match user-specific personal information.

**Call Stack Analysis.** Since our research object is third-party libraries in the Android App, we need to analyze the acquisition and transmission of private information by the third-party library. In the third-party library detection module, we can get a list of third-party libraries used in the application. The analysis of the call stack can help us know about the privacy leakage through the third-party library. As long as the privacy information is successfully matched, we will evaluate the privacy leakage risk of the third-party library by parsing the call stacks in logs of source APIs and sink APIs. In this paper, we divide the privacy leakage risk of third-party libraries into four levels according to the following rules:

- Level 0: No private information is matched;
- Level 1: Any TPL calls a source API directly or indirectly;
- Level 2: Any TPL indirectly calls a sink API, and any private information is matched in the API monitor log;
- Level 3: Any TPL directly calls a sink API, and any private information is matched in the API monitor log.

## 4 Evaluation

### 4.1 Dataset and Environment

To compare with state-of-the-art TPL detection tools, we get an App dataset from [28], which contains the ground truth of TPLs in Apps. The dataset consists of two parts: $\text{Dataset}_{ob}$ for evaluating obfuscation resistance capability and $\text{Dataset}_{per}$ for evaluating performance on precision and time cost. $\text{Dataset}_{ob}$ consists of Apps processed by Proguard and Dasho. However, the ground truth in the dataset only considers open-source TPL, we extend it to include closed-source TPLs and get a complete ground truth.

For privacy leakage analysis of TPL, there are no open-source tools for comparison and no widely used datasets, so we randomly selected 64 Apps from $\text{Dataset}_{per}$, these applications include categories of reading, video, shopping, image processing, etc.

The program runs on the ubuntu 18.04 operating system and the feature clustering algorithm is implemented with the scikit-learn toolkit. We use frida of version 15.1.14 for dynamic instrumentation. We wrote Libmonitor in python version 3.6.9.

### 4.2 TPL Detection

We compare our tool with two state-of-the-art clustering-based tools, i.e., Libradar and Libd. We do not consider those tools based on similarity comparison [13,14,16,32,33] because they can not detect closed-source TPLs. Both Libradar and LibD use the cluster-based approach to detect TPL, and we can get the profile database or source code of these two tools online. We design experiments to answer these questions:

**RQ1** How is Libmonitor's capability to resist obfuscation?
**RQ1** How does Libmonitor perform on detecting TPLs?
**RQ1** Is Libmonitor efficient compared to other tools?

**RQ1: How Is Libmonitor's Capability to Resist Obfuscation?** To evaluate the capability of resisting code obfuscation, we perform TPL detection on $\text{Dataset}_{ob}$ with 162 Apps, which are obfuscated respectively by Proguard and Dasho. We compare our tool with Libradar and Libd, and the statistical results are shown in Table 3. The results show that Libmonitor has best performance on F1 metrics for Apps obfuscated both by Dasho and Proguard, Libradar performs well for Proguard Apps, and Libd performs well for Dasho Apps.

We analyze the experimental results. On the one hand, different obfuscation tools have different effects on the results. Proguard generally employs identifier renaming while Dasho performs dead code removal and control flow randomization for Apps. Libradar requires an accurate comparison of hashes of API call frequencies and is sensitive to dead code removal. Libd needs to construct the control flow graph (CFG) to generate features, and thus is sensitive to control

flow randomization caused by Dasho. Compared to exact matching detection methods or methods based on CFG, our clustering method uses fuzzy matching and thus performs better. On the other hand, since our method is based on clustering, dataset selection is important. Our clustering dataset contains a more updated version of Apps, so it has better detection results (Table 3).

**Table 3.** Experiment Result on $dataset_{ob}$

| Tool | Proguard | | | Dasho | | |
|---|---|---|---|---|---|---|
| | Precise | Recall | F1 | Precise | Recall | F1 |
| Libmonitor | 71.31% | 74.91% | 0.7307 | 91.50% | 63.76% | 0.7515 |
| Libradar | 94.08% | 58.19% | 0.7191 | 84.21% | 8.36% | 0.1521 |
| Libd | 54.06% | 61.50% | 0.5754 | 60.29% | 58.71% | 0.5949 |

**RQ2: How Does Libmonitor Perform on Detecting TPLs?** To measure the precision of third-party library detection for real-world applications, we use another dataset $dataset_{per}$ for evaluation, and the results can be seen in Table 4. Moreover, we extend the ground truth of the original dataset with 245 closed-source TPLs and count the experimental results of three tools respectively. The results show that Libmonitor outperforms other tools on F1 metrics when detecting closed-source TPLs. The F1 values of three tools are low according to original ground truth, because the clustering-based approach can detect many closed-source TPL, and the original ground truth doesn't consider about it. Libd hashes basic blocks and concatenates features, it can resist code obfuscation but doesn't perform well in restoring package names, so it causes many false positives. Compared with Libradar, our feature cluster process is based on the fuzzy match, hence feature vectors are easier to cluster and more clusters can be generated, so Libmonitor can detect more third-party libraries (Table 4).

**Table 4.** Experimental results on $\text{Dataset}_{per}$ and extended Dataset

| Tool | $\text{Dataset}_{per}$ | | | Extended Dataset | | |
|---|---|---|---|---|---|---|
| | Precise | Recall | F1 | Precise | Recall | F1 |
| Libmonitor | 30.43% | 71.80% | 0.4275 | 79.07% | 83.98% | 0.8145 |
| Libradar | 37.77% | 50.74% | 0.4330 | 95.78% | 53.48% | 0.6864 |
| Libd | 9.52% | 8.70% | 0.0909 | 24.35% | 49.12% | 0.3256 |

**RQ3: Is Libmonitor Efficient Compared to Other Tools?** To evaluate the efficiency of Libmonitor, we run it and two baseline tools on $\text{Dataset}_{per}$ to compare their runtime cost. The experimental results are listed in Table 5. The performance of Libmonitor is significantly better than Libd, and slightly

weaker than Libradar. In order to represent the performance of these tools more intuitively, we plot their run time on top 40 Apps in Fig. 4. Obviously, Libmonitor and Libradar perform comparably while Libd is the most time-consuming.

**Table 5.** Run time on $\text{Dataset}_{per}$

| Tool | All | Average |
|---|---|---|
| Libmonitor | 6280.1 s | 28.9 s |
| Libradar | 5375.8 s | 24.8 s |
| Libd | 27477.0 s | 126.6 s |

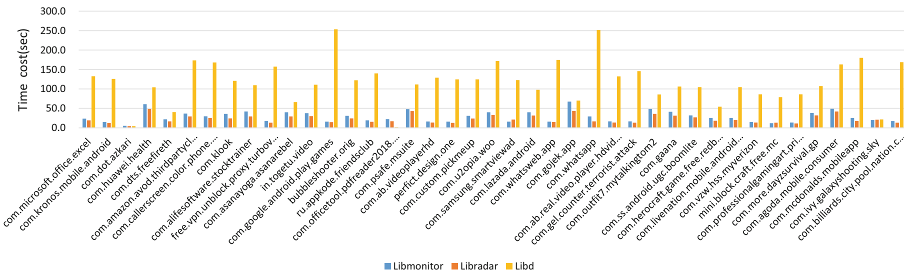


**Fig. 4.** Run time on top 40 apps in $\text{Dataset}_{per}$.

We explain the reasons behind the results as follows. First, Libd needs to build CFG and concatenates features of basic blocks, which is relatively time-consuming. Both Libradar and Libmonitor use the frequency of API calls as feature by parsing the dex files, and hence run quickly. Second, Libmonitor divides Android's APIs into more fine-grained types than ones in Libradar, causing slightly more run time.

### 4.3 Privacy Leakage Analysis

We use dynamic instrumentation technique to monitor sensitive APIs, then we perform information match and call stack analysis. Finally, we generate privacy leakage analysis reports for third-party libraries. We perform privacy leakage analysis on 64 Apps and obtain the following experimental results.

**Case Studies.** Some Apps may leak private information through interfaces other than the network interface. For instance, as shown in Fig. 5, *Cartola* App prints Advertiser_ID information through logcat interface. Besides, we found a serious security threat: *DoorDash* App transmits the passwords and email of users in plain text, which means that one can see users' password and email

through logcat, as indicated in Fig. 6, where we hide some user data such as mobile phone and email. The App developers should take care of these behaviors when using TPLs.

{'method': "android.util.Log.v.overload('java.lang.String', 'java.lang.String')", 'srctype': 'Advertiser_ID', 'srcval': 'b5db678b-ab72-4a9f-973c-a6e7202679c5', 'formatch': 'Sending hit to service PATH: https: PARAMS: cd=Parciais_dos_times, a=2094971135, ht=1647671318431, an=Cartola FC, tid=UA-20936277-33, cd2=, cd1=, uid=, adid=b5db678b-ab72-4a9f-973c-a6e7202679c5, ate=1, t=appview, av=4.5.0, v=1, _u=.4nL, ul=zh-cn, aid=br.com.mobits.cartolafc, sr=1440x2392, cid=decef45a-96e6-4824-97f7-0827ed785d36, '}

**Fig. 5.** Case study: privacy leakage analysis on Cartola App.

{'method': "android.util.Log.d.overload('java.lang.String', 'java.lang.String')", 'srctype': 'email_pt', 'srcval': 'xxxxxxxxxxxxxxxxxxxxxx',
'formatch': '{"last_name":"nickname","phone_number":"xxxxxxxxxxx",
"password":"0123456789","first_name":"test","
email":"xxxxxxxxxxxxxxxxxxxxxx"}'}

**Fig. 6.** Case study: privacy leakage analysis on DoorDash App.

**Experimental Results.** We define a piece of privacy leakage data that one TPL participates in as a privacy leak risk. For example, if a privacy leakage behavior involves two different TPLs, we will judge it as two privacy leakage risks. Libmonitor found 5809 pieces of privacy leakage risks caused by 152 TPLs in 64 applications, and the risks in level 1 to level 3 are 3118, 797 and 1894 respectively. Generally, private information is not necessarily leaked after obtained through source APIs, and may be used as user profile, so source APIs are called more frequently, i.e., risks in level 1 are the most. Risks of level 2 and level 3 represent privacy leakage through sink APIs called indirectly and directly by TPLs, respectively. Therefore, 46.3% of found risks involves privacy leakage caused by TPLs.

We count third-party libraries with more leakage risks, and record the types of private information that are leaked more frequently, as shown in Table 6 and Table 7. TPL will participate in the leakage of private information. It can be seen from the experimental results that *com.google.android.gms* has the most risks of privacy leakage, and AdvertiserID is the most easily leaked type of private information. Most of the applications in the dataset integrate Google's advertising service, so AdvertiserID is easy to be leaked. Most Apps on Google play rely on google mobile service(gms), so *com.google.android.gms* is used frequently.

**Table 6.** Top 10 privacy leaked by 152 TPLs

| Info. type | Freq. |
|---|---|
| AdvertiserID | 1163 |
| BuildNumber | 342 |
| DeviceType | 323 |
| Country | 280 |
| Date of birth | 99 |
| Timezone | 81 |
| AndroidID | 44 |
| Email | 34 |
| Uid | 14 |
| Kernel version | 12 |

**Table 7.** Top 10 TPLs leaking privacy

| PkgName | L1 | L2 | L3 | Total |
|---|---|---|---|---|
| com.google.android.gms | 1064 | 26 | 365 | 1455 |
| com.mopub | 93 | 269 | 341 | 703 |
| com.ironsource | 7 | 0 | 599 | 606 |
| com.google.firebase | 71 | 310 | 179 | 560 |
| io.fabric.sdk.android | 246 | 11 | 22 | 279 |
| com.facebook | 102 | 11 | 150 | 263 |
| com.crashlytics.android | 252 | 0 | 0 | 252 |
| com.applovin | 144 | 2 | 6 | 152 |
| com.unity3d | 23 | 61 | 58 | 142 |
| com.free.ads | 108 | 0 | 0 | 108 |

We counted the frequency of source APIs and sink APIs called in these privacy leakage risks, as shown in Table 9 and Table 8. As for sink APIs, APIs in *android.util.Log* are the interface of logcat whose call frequency is 328. Others are the interface of network whose call frequency is 3490. The proportion of logcat interface is 8.6%, which is ignored in [20,21], because they just collect network traffic information. In addition, there are SMS and clipboard interfaces. We did not find calls of these two types of interfaces in this experiment. As to source APIs, *android.provider.Settings$Secure.getString* is accessed by most TPLs, because androidID is a unique identifier which can be accessed easily without permission. Besides, *android.content.ContentResolver.query* is accessed frequently to query GSFID, image and vedio. Apps that integrate the Google framework are likely to access GSFID, while images and videos usually are accessed by multimedia related Apps.

**Table 8.** Top 10 sink APIs

| API | type | frequency |
|---|---|---|
| java.net.URL.$init.overload[2] | Network | 1336 |
| java.net.URL.$init.overload[0] | Network | 1315 |
| com.android.okhttp.internal.huc.HttpsURLConnection-Impl.setRequestProperty | Network | 742 |
| android.util.Log.i.overload[0] | Logcat | 219 |
| android.util.Log.d.overload[0] | Logcat | 89 |
| okhttp3.RequestBody.create.overload[2] | Network | 71 |
| android.util.Log.v.overload[0] | Logcat | 20 |
| okhttp3.RequestBody.create.overload[1] | Network | 12 |
| java.net.URL.$init.overload[5] | Network | 10 |
| org.apache.http.client.methods.HttpGet.$init.overload[0] | Network | 4 |

**Table 9.** Top 10 source APIs

| API | Frequency |
|---|---|
| android.provider.Settings$Secure.getString | 2589 |
| android.content.ContentResolver.query%GSFID | 572 |
| android.hardware.SensorManager.getDefaultSensor | 417 |
| android.content.ClipboardManager.getPrimaryClip | 97 |
| android.net.wifi.WifiInfo.getMacAddress | 86 |
| android.os.BatteryManager.getIntProperty | 84 |
| android.location.LocationManager.getLastKnownLocation | 49 |
| android.net.wifi.WifiInfo.getIpAddress | 27 |
| android.content.ContentResolver.query%image | 16 |
| android.content.ContentResolver.query%video | 16 |

## 5 Discussion

Libmonitor has limitations. First, Libmonitor can also detect open-source TPLs but it cannot accurately identify the specific version of each TPL. As a future work, we will combine clustering method and similarity comparison method to address this issue. Second, our approach to analyze privacy leakage cannot obtain complete data flow from source to sink, which needs to be improved in future work.

## 6 Related Work

### 6.1 Third-Party Library Detection

On third-party library detection, some prior studies use the simplest whitelist method. For example, Liu et al. [12] collected a whitelist of 400 SDKs to analyze the privacy leakage of third-party libraries and classified them according to the functions of the third-party libraries. The whitelist-based method generally requires manual collection and needs to be continuously updated as new third-party libraries appear. In addition, the code obfuscation in Android applications makes this method invalid.

Other research efforts perform feature extraction on Android applications and detect third-party libraries through similarity comparison. LibScout [16] uses class hierarchy analysis to construct a Merkle tree with a fixed depth of 3 as the configuration file of each library and proposes a matching algorithm to calculate the similarity with collected libraries. They collected 800 libraries (9623 versions in total) and constituted a tangible database to ensure accurate detection results. OSSPolice [29] uses normalized signatures and function centroids [30] as features, then uses a hierarchical indexing scheme to compare the similarity between the feature files and the tens of thousands of source files in

the benchmark database. When analyzing the library version, OSSPolice uses the software birthmark [31]to accurately detect OSS versions. According to the identified third-party SDK version, OSSPolice reports the third-party SDK version that contains security vulnerabilities or violates the open-source agreement. OSSPolice has also built a third-party SDK whitelist, which contains 110 Java libraries authorized under GPL and AGPL terms.

ATVHUNTER [33] conducts candidate TPL decoupling by class dependency and uses features of two granularities for TPL matching, the coarse-grain feature is serial numbers assigned in the control flow graph, and the fine-grained feature is the hash of opcode in the basic block. ATVHUNTER builds a TPL database and can detect the version of the TPL. LibID [13] has two library identification schemes, designed for scalability and accuracy, respectively named LibID-S, LIbID-A. LibID-S uses textual representation in basic blocks as the feature, LidID-A makes use of class dependency to get accurate matching of the TPL version, besides, LibID uses LSH and minihash to speed up class matching. These similarity-based tools require building TPL profile in advance and can't detect closed-source TPLs.

### 6.2 Privacy Leakage Analysis

FlowDroid [34] is a static taint tracking system based on Soot, which simulates the complete Android application life cycle, and expresses the control flow call relationship between the Activity life cycle and all callback functions as a control flow graph(CFG). FlowDroid uses CFG to track the sensitive data flow from the source point to the sink point, but FlowDroid cannot analyze data flow of inter-component communication and uses static analysis which is prone to false positives. Enck et al. modified the source code of the Android system and designed the dynamic taint tracking system TaintDroid [18]. By modifying the Dalvik VM interpreter, the corresponding taint label was added to the private data, and four levels of taint were defined. TaintDroid provides logs to record private information behaviors and sends messages in the notification bar to warn users of privacy leakage. However, TaintDroid only implements data flow tracking but does not analyze control flow and native methods. Taintman [19] performs static instrumentation on target Apps and system libraries, and uses taint analysis technique to track data flow and control flow. In addition, Taintman uses an execution environment reconstruction technology called reference hijacking, which allows the target application to reference and modify the system library, so it can run on Android devices without root privilege. However, Taintdroid and Taintman need to modify system code, which is inconvenient for migration and adaptation.

## 7 Conclusion

In this paper, we present a novel approach to detect third-party libraries (TPLs) in Android Apps and analyze potential privacy leakage in TPLs. We implement

a tool named Libmonitor. It leverages the call frequencies of different types of APIs as features and uses fuzzy match strategy when performing the clustering algorithm, and thus works well on obfuscated TPLs.

Experimental results of TPL detection show that Libmonitor outperforms two state-of-the-art TPL detection tools (i.e., Libradar and LibD) on two datasets. Besides, Libmonitor found 5809 pieces of privacy leakage risks caused by 152 TPLs in 64 real-world Apps.

## References


1. IDC. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/os
2. Lin, J., Liu, B., Sadeh, N., et al.: Modeling users mobile app privacy preferences: restoring usability in a sea of permission settings. In: Proceeding SOUPS 2014 Proceedings of the Tenth USENIX Conference on Usable Privacy and Security, vol. 199 (2014)
3. Wang, H., Guo, Y., Ma, Z., Chen, X.: WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 71–82. ACM, Baltimore (2015).https://doi.org/10.1145/2771783.2771795
4. Slowmist Knowledge-Base. https://github.com/slowmist/Knowledge-Base/blob/master/tradingview-xss-vul.md
5. Wu, D., Gao, D., Chang, R.K.C., He, E., Cheng, E.K.T., Deng, R.H.: Understanding open ports in android applications: discovery, diagnosis, and security assessment. In: Proceedings 2019 Network and Distributed System Security Symposium. Internet Society, San Diego (2019). https://doi.org/10.14722/ndss.2019.23171
6. Almanee, S., Unal, A., Payer, M., Garcia, J.: Too quiet in the library: an empirical study of security updates in android apps' native code. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1347–1359. IEEE, Madrid (2021). https://doi.org/10.1109/ICSE43902.2021.00122
7. Reardon, J., Feal, Á., Wijesekera, P.: 50 ways to leak your data: an exploration of apps' circumvention of the android permissions system, vol. 19 (2019)
8. Mobile application (App) data security and personal information protection white paper. http://www.caict.ac.cn/kxyj/qwfb/bps/201912/P020191230332039577332.pdf
9. Maven Repository. https://mvnrepository.com/
10. GitHub: Where the world builds software. https://github.com/
11. Lin, J., Sadeh, N., Amini, S., Lindqvist, J., Hong, J.I., Zhang, J.: Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In: Proceedings of the 2012 ACM Conference on Ubiquitous Computing - UbiComp 2012, p. 501. ACM Press, Pittsburgh (2012). https://doi.org/10.1145/2370216.2370290
12. Liu, B., Liu, B., Jin, H., Govindan, R.: Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, pp. 89–103. ACM, Florence (2015). https://doi.org/10.1145/2742647.2742668
13. Zhang, J., Beresford, A.R., Kollmann, S.A.: LibID: reliable identification of obfuscated third-party Android libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 55–65. ACM, Beijing (2019). https://doi.org/10.1145/3293882.3330563

14. Wang, Y., Wu, H., Zhang, H., Rountev, A.: ORLIS: obfuscation-resilient library detection for Android. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft 2018, pp. 13–23. ACM Press, Gothenburg (2018). https://doi.org/10.1145/3197231.3197248
15. Ma, Z., Wang, H., Guo, Y., Chen, X.: LibRadar: of third-party libraries in Android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion - ICSE 2016, pp. 653–656. ACM Press, Austin (2016). https://doi.org/10.1145/2889160.2889178
16. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS 2016, pp. 356–367. ACM Press, Vienna (2016). https://doi.org/10.1145/2976749.2978333
17. Li, M., et al.: LibD: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 335–346 (2017). https://doi.org/10.1109/ICSE.2017.38
18. Enck, W., et al.: TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. Commun. ACM. **57**, 99–106 . https://doi.org/10.1145/2494522
19. You, W., Liang, B., Shi, W., Wang, P., Zhang, X.: TaintMan: an ART-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. IEEE Trans. Depend. Secure Comput. **17**, 209–222 (2020). https://doi.org/10.1109/TDSC.2017.2740169
20. He, Y., Yang, X., Hu, B., Wang, W.: Dynamic privacy leakage analysis of Android third-party libraries. J. Inf. Secur. Appl. **46**, 259–270 (2019). https://doi.org/10.1016/j.jisa.2019.03.014
21. Wongwiwatchai, N., Pongkham, P., Sripanidkulchai, K.: Comprehensive detection of vulnerable personal information leaks in android applications. In: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 121–126. IEEE, Toronto (2020). https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163043
22. Dong, S., et al.: Understanding android obfuscation techniques: a large-scale investigation in the wild. In: Beyah, R., Chang, B., Li, Y., Zhu, S. (eds.) SecureComm 2018. LNICST, vol. 254, pp. 172–192. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01701-9_10
23. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 468–471. ACM, Austin (2016). https://doi.org/10.1145/2901739.2903508
24. A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/
25. Android Developer. https://developer.android.com/reference/packages
26. Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: an efficient data clustering method for very large databases. SIGMOD Rec. **25**, 103–114 (1996). https://doi.org/10.1145/235968.233324
27. Monkey. https://developer.android.com/studio/test/monkey
28. Zhan, X., et al.: Automated third-party library detection for Android applications: are we there yet? In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 919–930. ACM, Virtual Event Australia (2020). https://doi.org/10.1145/3324884.3416582

29. Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W.: Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2169–2185. ACM, Dallas (2017). https://doi.org/10.1145/3133956.3134048
30. Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, pp. 175–186. ACM Press, Hyderabad (2014). https://doi.org/10.1145/2568225.2568286
31. The protection of computer software - Its technology and applications: edited by Derrick Grover, 2nd Edition, 1992 (British Computer Society Monographs in Informatics - Cambridge University Press, Softcover), 307pp, £17.95 (US $32.95), ISBN 0-521-42462-3. Computer Law & Security Review. 8, 204 (1992). https://doi.org/10.1016/0267-3649(92)90069-L
32. Zhang, Y., et al.: Detecting third-party libraries in Android applications with high precision and recall. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 141–152. IEEE, Campobasso (2018). https://doi.org/10.1109/SANER.2018.8330204
33. Zhan, X., et al.: ATVHunter: reliable version detection of third-party libraries for vulnerability identification in android applications. In: ICSE (2021)
34. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2014, pp. 259–269. ACM Press, Edinburgh (2013). https://doi.org/10.1145/2594291.2594299