Journal of Information Security and Applications 46 (2019) 259–270

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Dynamic privacy leakage analysis of Android third-party libraries

Yongzhong He[a], Xuejun Yang[b], Binghui Hu[b], Wei Wang[a,b,*]

[a] Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China
[b] School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China

ARTICLE INFO

Article history:
Available online 28 March 2019

Keywords:
Privacy leakage
Third-party library
Android Apps

ABSTRACT

The third-party libraries are reusable resources that are widely employed in Android Apps. While the third-party libraries provide a variety of functions, they bring serious security and privacy problems. The third-party libraries and the host Apps run in the same process and share the same permissions. Whether the third-party libraries are compliant with privacy policies is out of the control of App developers. In this work, we identify four types of privacy leakage paths inside Apps with case studies. Based on the Xposed framework, we propose a fine-grained and dynamic privacy-leakage analysis tool to analyze the privacy leakage behaviors of the third-party libraries in real time. Our tool can first identify the third-party libraries inside Apps, and then extracts call chains of the privacy source and sink functions during the execution of Apps, and finally evaluate the risks of privacy leaks of the third-party libraries according to the privacy leakage paths. We evaluate our tool over 150 popular Apps, collecting 1909 privacy data related call chains. We find that many third-party libraries access to private information. Moreover, they set up direct network connections to remote servers, which suggests that the third-party libraries pose a great privacy risk. The experiments results show that our tool can achieve real-time, fine-grained and dynamic privacy leakage analysis on Android Apps.

1. Introduction

Most smartphones are operating on the Android system. It is showed [1] that, as the first quarter of 2017, Android system shares up to 87% in the Chinese smartphone market. The large-scale use of Android phones has driven the rapid development of Android mobile applications. Most of the Apps leak users' private information, but the users know nothing about where the privacy flows to and how it is used. Generally, an App is comprised of the host application and multiple third-party libraries. Third-party libraries provide common functionalities such as advertising push, location-based services, and social services. Accordingly, we classify them as advertising libraries, analysis libraries, location service libraries, social networking libraries. Some examples are listed in Table 1.

We read and analyzed the development guides and privacy policies of these third-party libraries, then summarized seven common privacy requirements of the advertising database and analysis library, as shown in Table 2. The privacy policies of the third-party libraries suggest that they usually require access to Android device information (IMEI, IMSI, MAC, etc.) to generate the Android-device identifier, read geographic information and establish network connectivity, which may expose users to unnoticeable privacy leakage risks.

Many third-party libraries are released in class or jar format without source codes so that even App developers might not be aware of risks involved in them. In the Android platform, the developers of the host Apps and third-party libraries have different interests. While an App is running, the host App and the third-party libraries run in the same process, enjoy the same permissions, and their boundaries are blurred. In essence, the security threat of Android third-party libraries derives from the abuse of permissions of the host Apps. However, existing access control mechanisms of Android system are coarse-grained, which cannot distinguish the subject of any privacy access request, whether it is the host App or a third-party library, therefore the privacy leakage risk is out of control for the App developers and users.

In order to distinguish between a host App and the third-party libraries, it is essential to detect and identify the third-party libraries first. There are two types of methods to detect third-party libraries in Android. The first is whitelisting. For example, AdRisk [2] uses a whitelist of 100 advertisement libraries to reveal potential risks in Apps. Chen et al. [3] has a list of 73 libraries in the whitelist to filter third-party libraries when detecting App clones. This type of methods is scalable to thousands of Apps because it only needs to compare the package names. However, it cannot

* Corresponding author.
E-mail addresses: yzhhe@bjtu.edu.cn (Y. He), 16120440@bjtu.edu.cn (X. Yang), 15120360@bjtu.edu.cn (B. Hu), wangwei1@bjtu.edu.cn (W. Wang).

https://doi.org/10.1016/j.jisa.2019.03.014
2214-2126/© 2019 Elsevier Ltd. All rights reserved.

Table 1
The taxonomy of third-party libraries.

Type	Example
advertising libraries	Admob, Inmobi
analysis libraries	Umeng,Flurry,Analytics
location service libraries	Baidu Location,AmapLocation
social networking libraries	Tencent, Sina Weibo
game engine libraries	Unity3D, Badlogic
development tool libraries	Oauth,Jsoup

resist name obfuscation. The other approach is based on feature vector extraction and matching. LibRadar [4] uses the hash value of API call frequency, the percentage ratio of the total number of API calls to total categories of API calls, to define the fingerprint of a library package. LibD [5] takes advantage of the relationship between the classes and methods of java and the metadata in the App. OSSPOLICE [6] uses string literals, the exported function and control-flow graph in C/C++ source. In Java source, it chooses string constant, normalized class and function centroid. The second method is resistant to obfuscation but time-consuming.

There are some generic tools of privacy leakage analysis for Apps. FlowDroid [7] and TrustDroid [8] are based on static privacy leakage analysis, while TaintDroid [9] and NDroid [10] are dynamic tools and require modifying the Android platform. AppFence [11] is built upon TaintDroid and can block unwanted data transmission. APPIntent [12] is a combination of static and dynamic analysis methods to determine whether the acquisition of private information is consistent with the intention of users. However, these technologies focus on the granularity level of Apps, and they cannot directly analyze the privacy behaviors of third-party libraries inside Apps.

There have been many researches on analyzing privacy leakage of the advertising library, or other types of third-party libraries. Several papers [13–15] examine Android advertising libraries through static analysis and find that many in-app ad libraries collect privacy-sensitive information [14] even without declaring the use of privacy-related permissions in their documentation [15], and such behaviors may be growing over time [13]. Livshits et al. [16] propose an automated approach to identify and place missing permission prompts where third-party libraries may potentially misuse permissions. A few studies employ dynamic analysis to disclose potential risks [17,18]. Brahmastra [17] proposes an automation tool to test the potential vulnerability of third-party libraries embedded into mobile Apps, beyond the reach of GUI-based testing tools. MAdFraud [18] adopts a dynamic analysis approach to detect fake ad clicks by host Apps. In our previsous work

Some researchers have introduced protection mechanisms against permission abusing by third-party libraries. AdDroid [19] and PEDAL [20] separate the advertising library from the main function code and reduce the ad library's authority to prevent the advertising library from leaking the privacy data. In order to refine the granularity of Android access control mechanism, Compac [21] treats the third-party libraries and the host App as separate components and uses the application run-time information to make a decision for the privileged access request. However, all these solutions require modifying the Android system or repackage the applications, so they are not user-friendly. Our tool does not need to modify the Android system or repackage the Apps. In our previous work, we detected malicious Android apps or anomalies with various kinds of features [22–24] and different methods [25–27]. We also discovered Android sensor usage behaviors with data flow analysis [28] and analyzed privacy of analytics libraries in Android ecosystem [29].

In this paper, we study the privacy leakage behaviors of third-party libraries inside Android Apps and identify four kinds of paths for privacy leakage of Android Apps through a case study. Then we propose a privacy leakage analysis framework which can realize fine-grained and dynamic analysis in real-time based on the Xposed [30] framework. Xposed is a framework that allows users to develop third-party plug-ins for customizing the behavior of the Android system and apps without touching any APKs. With the help of the Xposed framework, we can hook the Android privacy-related APIs to identify fine-grained privacy leakage inside Apps. We first design a third-party library detection scheme to identify and locate the third-party libraries in Apps. Then we treat the host App and third-party libraries inside an App as different subjects. Based on hooking of privacy APIs, our tool can answer the question of which specific subject in an App obtains the privacy data and which one leaks it. Finally, we conduct an analysis for 150 Android Apps selected from the application markets and evaluate the privacy leakage risks of some commonly used third-party libraries.

In summary, we make the following contributions.

- We develop a new third-party library detection approach for Android Apps. For the non-obfuscated App, we use the whitelist-based detection. For the obfuscated App, we adopt the motifs-based detection that is resilient to name obfuscation.
- We identify four types of privacy leakage paths inside Android App. We present the design and implementation of the tool combination with static detection and dynamic Xposed framework to analyze privacy leakage behaviors of third-party libraries in real time.
- We evaluate 150 popular Apps, collecting 1909 private information leakage related call chains. We find that many third-party libraries have access to private information, and most of the third-party libraries have direct network connections.

The rest of this paper is organized as follows. Section 2 describes fine-grained privacy leakage paths in Android platform. In Section 3, we outline the framework of our tool and describe the framework in details. In Section 4, we present the experiments and results. In Section 5, we discuss some potential issues and future research directions. We conclude this paper in Section 6.

Table 2
The privacy permissions of third-party libraries.

Privacy rights	Admob	Inmobi	Youmi	Wanpu	Umeng	Flurry
android.permission.INTERNET	Yes	Yes	Yes	Yes	Yes	Yes
android.permission.ACCESS_NETWORK_STATE	Yes	Yes	Yes	Yes	Yes	Optional
android.permission.ACCESS_WIFI_STATE		Optional	Yes	Yes	Yes	
android.permission.READ_PHONE_STATE			Yes	Yes	Yes	
android.permission.WRITE_EXTERNAL_STORAGE			Yes	Yes		
android.permission.GET_TASKS				Yes		
android.permission.ACCESS_COARSE_LOCATION		Optional	Optional			Optional
android.permission.ACCESS_FINE_LOCATION		Optional				Optional

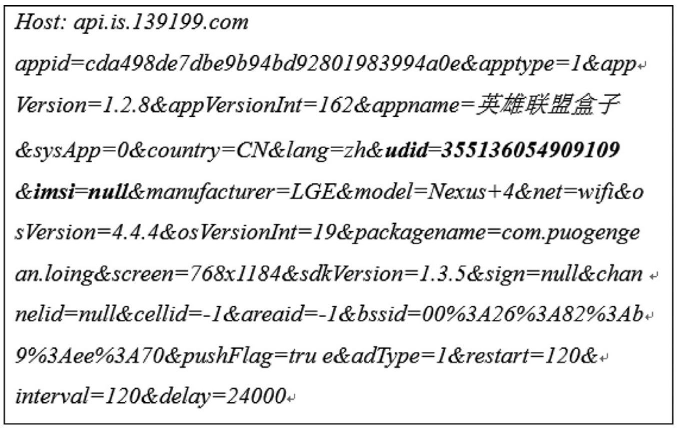

Host: api.is.139199.com
appid=cda498de7dbe9b94bd92801983994a0e&apptype=1&app
Version=1.2.8&appVersionInt=162&appname=英雄联盟盒子
&sysApp=0&country=CN&lang=zh&**udid=355136054909109**
&**imsi=null**&manufacturer=LGE&model=Nexus+4&net=wifi&o
sVersion=4.4.4&osVersionInt=19&packagename=com.puogenge
an.loing&screen=768x1184&sdkVersion=1.3.5&sign=null&chan
nelid=null&cellid=-1&areaid=-1&bssid=00%3A26%3A82%3Ab
9%3Aee%3A70&pushFlag=tru e&adType=1&restart=120&
interval=120&delay=24000

Fig. 1. Network traffic flow containing privacy data.

2. Fine-grained privacy leakage paths in Android

2.1. The method of finding privacy leakage paths

Firstly, we study how the host Apps and third-party libraries invoke methods to collect private information and analyze the privacy leakage paths of Android Apps. We use Tcpdump to get network traffic during the execution of an App. After extracting the network traffic flows, we find the request flow sending to the external server contains many privacy data. We define this kind of request flow as privacy flow. Fig. 1 shows the request flow of an APP that contains privacy data such as IMSI and UDID.

Then we decompile the App package file, search the source code according to the key fields in the privacy data, and locate the function call of sending privacy streams. Based on the inter-function invocation rules, the propagation trails of the privacy data within the App is reversely tracked until the privacy-sensitive API function calls of the Android system are found. Finally, the host App and third-party libraries are distinguished according to the package name structure of the Java source codes. By tracing the source codes, we can reconstruct the paths where privacy leaks occur in the App.

2.2. Classification of privacy leakage paths

By analyzing the propagation path of privacy data inside an App between third-party libraries and the host App, we identify four kinds of privacy leakage paths of the Android App as follows.

The four privacy leakage paths can be represented by the following four Figs 2–5. Solid lines represent function call relationships. Dotted lines represent the privacy getting path or privacy sending path. The privacy leakage path types are illustrated in Table 3.

Case 1: The host App directly calls the sensitive API, obtains and leaks the privacy data.

Case 2: The host App calls the third-party library function, indirectly the third-party library calls the sensitive API, the third-party library passes privacy data to the host App, and finally the host App leaks them.

Table 3
Privacy leakage paths classification.

Collector	Leaker	
Collector Leaker	The host App(directly)	The host App(indirectly)
The host App	Case 1	Case 2
Third-party Library	Case 4	Case 3

Table 4
A snapshot of an App's call stack.

Principal	Call stack
Host app	android.App.Activity.onCreate
Host app	com.example.userApp.MainActivity.onCreate
Third-party library	com.malicious.library.WebCodeRunner.run
Third-party library	com.ImgLib.takePicture
Privacy API	Android.hardware.Camera.takePicture

Case 3: The host App calls the third-party library function, indirectly calls the sensitive API, the third-party library sends the private information to the third-party servers, causing privacy leakage.

Case 4: The host App directly calls the sensitive API, and then the third-party library leaks the privacy data.

For simplicity, the illustration examples only consider one third-party library in a privacy leakage path. However, the classification is also valid for cases that some third-party libraries collaborating in the privacy leakage path. When there are multiple third-party libraries in the privacy leakage path, all the third-party libraries in the privacy getting path are labeled as privacy collectors, and all libraries in the privacy sending path are labeled as privacy leakers. For example, when a third-party library is only in the privacy getting path, but not in the privacy sending path, the privacy leakage path for this library is classified as Case 2.

In Android system, by inspecting the call stack information of the current thread, the sequence of method calls and principal of the target function can be located. By analyzing the call stacks of the privacy-sensitive APIs, we can reconstruct the privacy leakage paths dynamically. A snapshot of an app's call stack is shown in Table 4.

2.3. Analysis of privacy leakage paths

The different paths of privacy leakage have different risks. In Case 1, the privacy leakage path only consists of function calls from the host App. Privacy collector who reads the privacy data, and privacy leaker who sends privacy data out of the phone, are both the host App, so Case 1 is controllable for the App developers. In Case 2 and 3, the host App calls the sensitive API function indirectly through third-party libraries, so the privacy collecting path contains functions from third-party libraries. The main difference between Case 2 and Case 3 is the privacy-leaking path. In this paper, we focus only on the privacy leaking through the network connection. In Case 2, the network connection of privacy leakage is initiated by the host App, while in Case 3, the network connection is initiated by a third-party library. Therefore, it is obvious that Case 3 is the riskiest, uncontrollable situation.

In our case study, there is no privacy leakage situation belonging to Case 4. It is obvious that Apps are not dependent on third-party libraries.

3. Design and implementation

3.1. Privacy leakage risk analysis framework

In this section, we give an overview of our tool. Our tool is based on Xposed to dynamically detect and analyze privacy leakage. However, it also depends on static techniques to identify the third-party libraries at first. The granularity of privacy detection in our tool is refined to distinguish the host App and third-party libraries in the Android App. The framework of our tool is shown in Fig. 6.

Our tool is composed of three modules: the third-party library detection module, Xposed privacy monitor module and privacy risk analysis module.

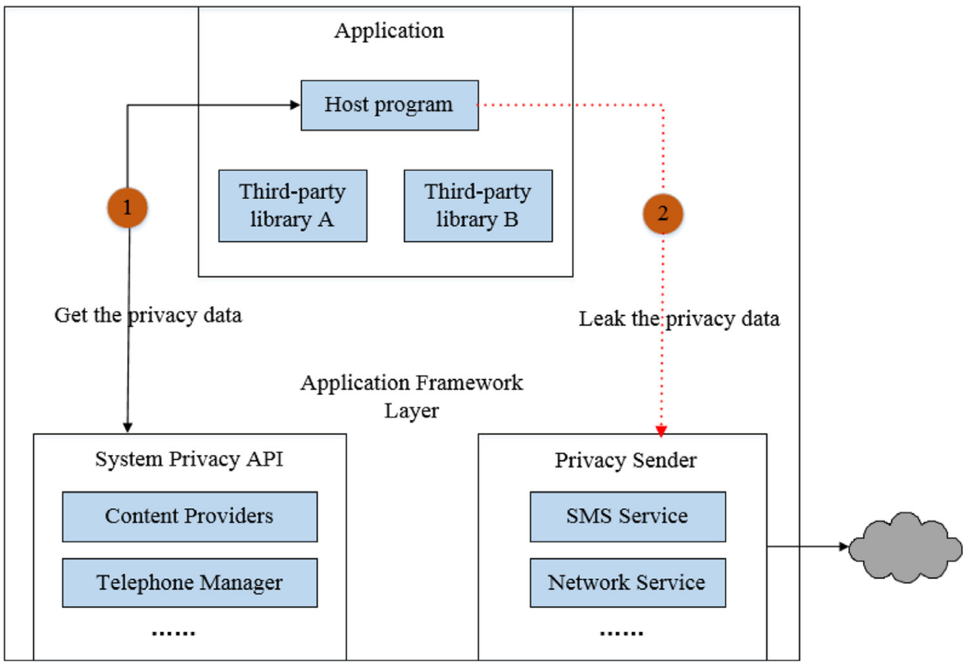


Fig. 2. The process of Case 1.

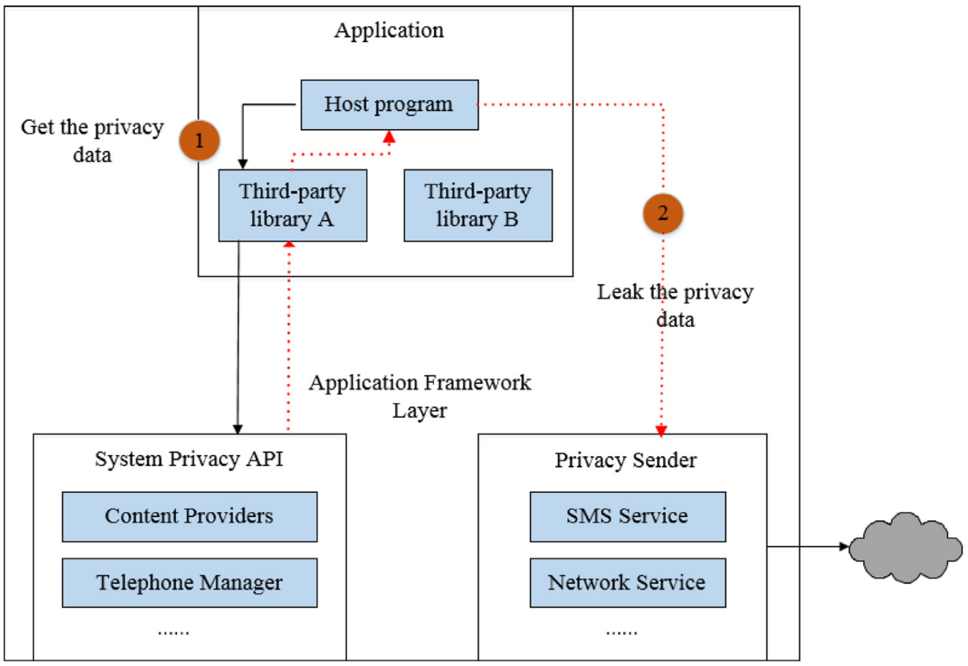


Fig. 3. The process of Case 2.

The third-party library detection module: We check whether an App is obfuscated after decompiling. If the App is obfuscated with rename according to the decision rules, we will use motifs-based detection to identify which libraries are included in the App. Otherwise, it will use whitelist-based detection.

The Xposed privacy monitor module: It is responsible for monitoring the privacy data propagating paths. We design the *BeforeHookedMethod* method that gets the call stack for the privacy source function and extracts the call chains. Then we match the call chain with third-party library list output by the third-party library detection tool and analyzes the caller of privacy source function. Meanwhile, we hook the sink functions to identify the caller of privacy leakage and generate privacy log recording call chains of each privacy access process.

The privacy risk analysis module: Based on privacy log, we develop the risk evaluation criteria for third-party library privacy leakage and classify the privacy risks of the third-party library. Then we use three levels to represent the degrees of risk and generate privacy report.

Our tool is implemented as an App. When we run the App under test and click on its UI to trigger the application-related functionalities. Our tool monitors the App's privacy behaviors in real-time and automatically generates a privacy log and privacy leakage analysis report.

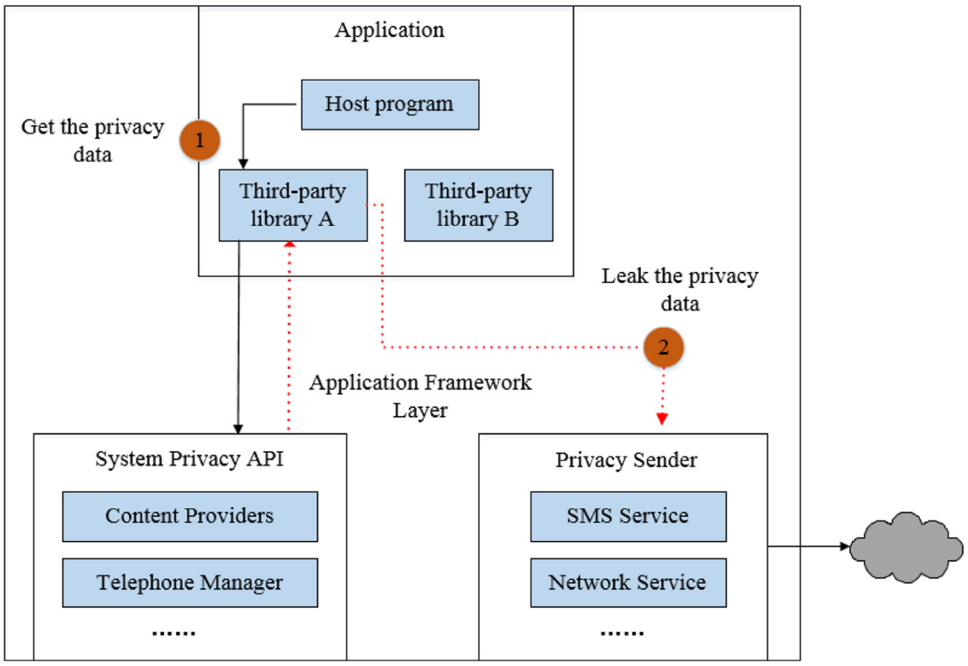


Fig. 4. The process of Case 3.

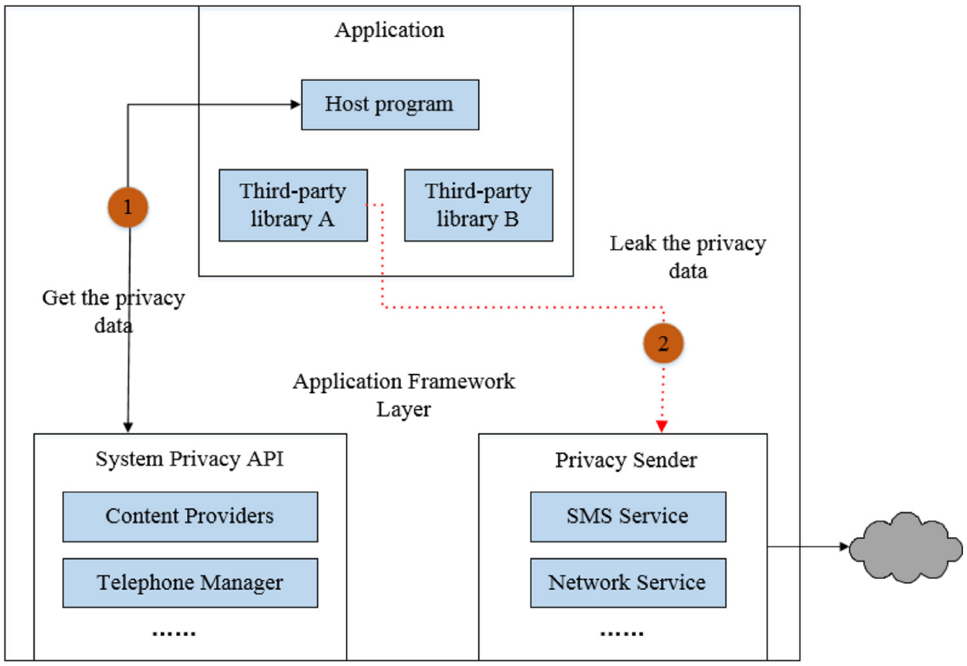


Fig. 5. The process of Case 4.

Compared with the existing tools for privacy leakage analysis, our tool has the following advantages.

1. Low coupling with Android system: No need to modify the Android platform or repackaging the Apps.

2. Suitable for various types of private information, various applications: Based on the Android system APIs, we can dynamically change the privacy data types managed by our tool, through implementing addition, deletion and modification operations. Moreover, our tool is suitable for privacy leakage analysis of various Apps, regardless of the App functionalities.

3. Dynamic and real-time: Our tool is a dynamic analysis tool based on the monitoring the actual execution of Apps. It is more practical and effective to report the privacy risk of the third-party libraries than the existing static analysis based on analyzing privacy policy and source code of the third-party library.

3.2. *Privacy-sensitive APIs*

We focus on the privacy data protected by Android permission system, which is the most sensitive privacy data for users, and other information that may be misused to expose user privacy, including the following types of information.

Equipment identification information: It can be used to identify phones or users, such as IMEI, IMSI, ICCID (SIM card unique identification), telephone numbers, and user account information.

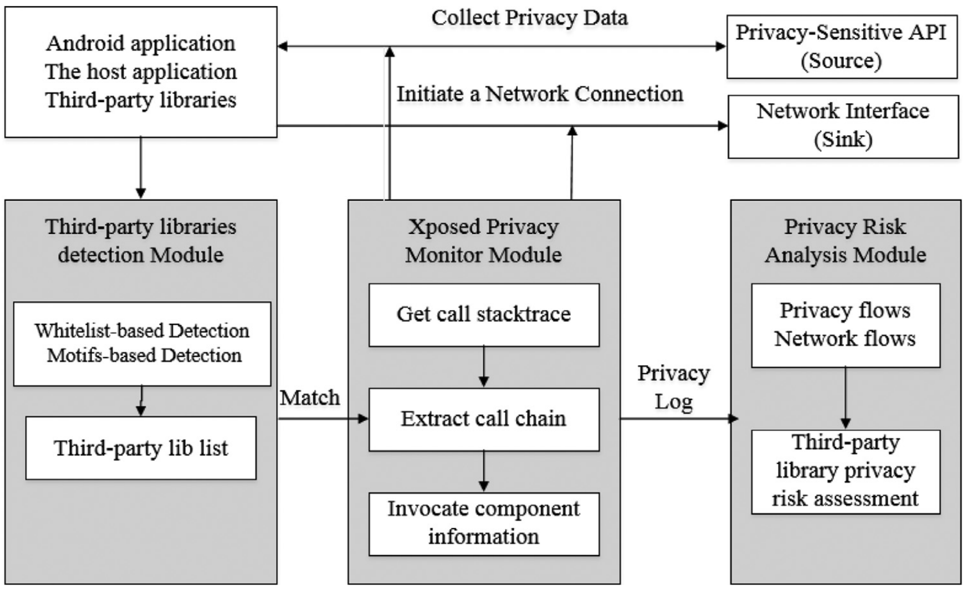

Android application
The host application
Third-party libraries
Collect Privacy Data
Privacy-Sensitive API (Source)
Initiate a Network Connection
Network Interface (Sink)
Third-party libraries detection Module
Whitelist-based Detection
Motifs-based Detection
Third-party lib list
Match
Xposed Privacy Monitor Module
Get call stacktrace
Extract call chain
Invocate component information
Privacy Log
Privacy Risk Analysis Module
Privacy flows
Network flows
Third-party library privacy risk assessment

Fig. 6. The framework of our tool.

Table 5
Privacy-sensitive APIs list.

Privacy type	System class	Method
IMEI	android.telephony.TelephonyManager	getDeviceId
IMSI	android.telephony.TelephonyManager	getSubscriberId
ICCID	android.telephony.TelephonyManager	getSimSerialNumber
Contact	android.provider.ContactsContract	PhoneLookup
Phone Number	android.telephony.TelephonyManager	getLine1Number
SMS	android.telephony.SmsManager	sendTextMessage
MIC	android.media.MediaRecorder	getAudioSource
MIC	android.media.MediaRecorder	startRecording
Camera	android.hardware.Camera	setPreviewDisplay
Sensors	android.hardware.SensorManager	getDefaultSensor
GPS Location	android.location.LocationManager	getLastKnownLocation
GPS Location	android.location.LocationManager	requestLocationUpdates
GPS Location	android.location.LocationManager	getLongitude
Cell Location	android.telephony.TelephonyManager	getCellLocation

Information database: The database stores some privacy data such as address books and text messages.

Location information: Android Apps obtain user location information through GPS, network or cellular base station. GPS positioning has the highest accuracy and needs ACCESS_FINE_ LOCATION permission. Network positioning and base station positioning have slightly reduced accuracy, only need the ACCESS_COARSE_ LOCATION permission.

Sensors: Android supports eight kinds of sensors including acceleration sensor, gyroscope, ambient light sensor, magnetic force sensor, direction sensor, pressure sensor, distance sensor, and temperature sensor. The sensors may be misused to leak users' privacy.

Camera and microphone: Android Apps use the camera to take and view photos, use microphones to record audio, which is privacy sensitive.

We use Apktool and Dex2jar to decompile the Apps from the Android application markets and identify the system API functions related to user privacy data. Some of the sensitive APIs are shown in Table 5.

3.3. The third-party library detection

Because some Apps are obfuscated but many are not, we present a new method to detect third-party libraries inside Android Apps to support detecting both obfuscated and plain Apps with high efficiency and accuracy. There are many types of obfuscation mechanisms such as adding useless codes, changing the logic of codes and so on. Among them, rename obfuscation is the most widely used in Apps. In order to be fast and accurate, our detection tool first decides whether an App is obfuscated after decompiling APK file of an App. If the App is obfuscated with rename according to the decision rules, we will use motifs-based detection to identify which libraries are included in the App. Otherwise, it will use whitelist-based detection. Finally, we generate a file containing a list of all the third-party libraries inside the App, which will be used in Xposed monitor module. Fig. 7 shows the workflow.

3.3.1. Rename obfuscation rules

After decompiling the App, we go through the names of all the packages. Based on our observation and analysis of obfuscated Android Apps, we design the rules to decide rename obfuscation. Given space limitations, we show some rules following.

Rule 1: *The name is a single character such as /a, /b.*

Rule 2: *The name is made up of several same characters such as /aa, /bbbbb.*

Rule 3: *The name is the combination of letters and special numbers such as /a123 and /bb123.*

For example, *c/a/a/a/c* is actually renamed from *uk/co/senab/, photoview/c* in Senab photoview library, and *com/ss* refers to *com/aviary* in Aviary library (Senab photoview is a third-party

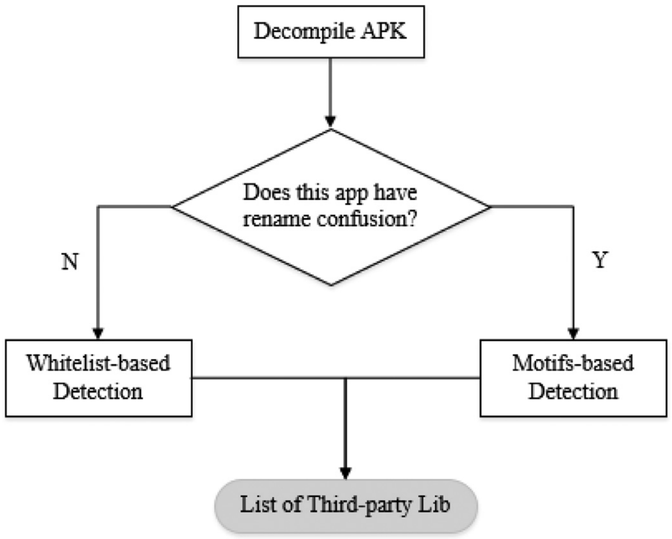

Decompile APK
Does this app have rename confusion?
N
Y
Whitelist-based Detection
Motifs-based Detection
List of Third-party Lib

Fig. 7. The framework of the detection tool.

Table 6
The whitelist of this paper.

Library	Library
com/amplitude/security	org/libreoffice
com/amulyakhare/textdrawable	org/mixare
com/andexert/expandablelayout	org/ligi/tracedroid
com/andreabaccega/formedittext	org/litepal
com/andreabaccega/widget	org/mailboxer/saymyname

library to zoom Android Image View, and Aviary is a library to edit the images in the Adobe Creative SDK).

3.3.2. *Whitelist-based detection*

We build the library whitelist in three ways. Some libraries are selected from the existing papers [31,32], and some from popular source code hosting website such as GitHub. Most libraries are identified by LibRadar [4] and LibD [5] from tens of thousands of Apps. Table 6 shows a part of the whitelist.

We match the names of the decompiled App's directories with the whitelist. For example, if an App has a directory named *org/mixare*, we mark this App as using library *org/mixare*(mix Augmented Reality Engine library), which is a free open source augmented reality browser and available for Android and for the iPhone3GS and above.

3.3.3. *Motifs-based detection*

We use motifs-based detection to identify third-party libraries with obfuscation. The overall framework of this method is illustrated in Fig. 8.

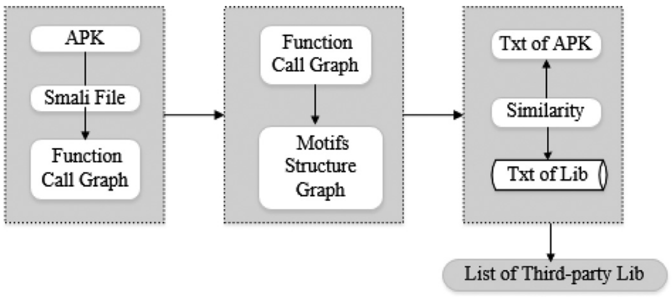


Fig. 8. The framework of motifs-based detection.

Table 7
The relationship between operation code and attributes.

Operation code	Attribute name	Operation code	Attribute name
move	V	invoke-super	I
move/16	V	invoke-direct	I
move-wide	V	invoke-interface	I
move-wide/16	V	invoke-static	I
move-result	V	const-wide/16	C
move-result-wide	V	const-wide/32	C
move-result-object	V	const-wide	C

First, we decompile the App code into smali code and rank all the operands in the user function as attributes of the nodes in order to generate the function call graph of the application. We note that in order to execute the App correctly, obfuscation tools should not change the basic operation code names. Then, we represent the function call graph with motifs subgraph structure stored as a text file. Finally, our detection tool can compare the attributes of nodes and the calling relationships of nodes. If the similarity between the App's file and a third-party library's file in the database is high, we can decide that the third-party library is included in the App.

1. Generating function call graph

We first define the function call graph as follows.

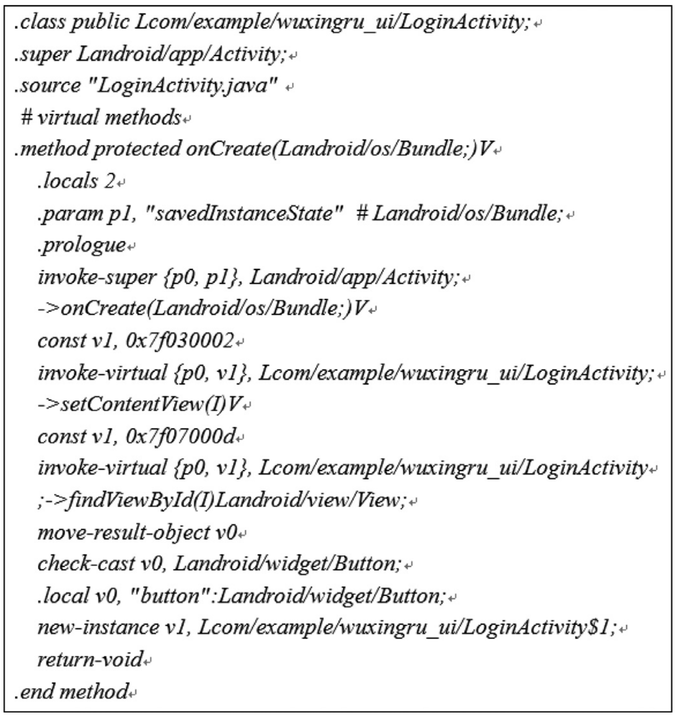
Definition 1: The function call graph is denoted by a directed graph $G = <V, E>$. V represents the set of all vertices (or nodes) in graph G and each vertex represents a user-defined method in the App. Each functionâs name consists of its class name, method name, and parameter types. Every node has an attribute that is comprised of operation code in this function. E represents the set of all directed edges in graph G. If node u is invoked before node v, we think there is a directed edge from node u to node v, where u is the calling node and v is the invoked node.

First, we decompile the app with Apktool to get the smali codes, resource files, and smali configuration files. Then we iterate through each smali file. Each function in the smali file is a node whose id is the class name, the method name and the parameter type of the method. Next, we use the operation code sequence in the function code block as the attribute of the node. We analyze 221 operation codes commonly used in smali codes. Table 7 shows the attribute values of some of the operation codes. Finally, according to the *invoke* keyword in the smali file, the calling relationship between function nodes is determined

For example, Fig. 9 is an *onCreate* method of the smali file, which is a node. This method has eight opcodes: *invoke-super, const, invoke-virtual,* move-result-object, *return-void*, and so on. Based on the correspondence between the opcodes and attribute values in Table VII, this node attribute is *ICICIVNR*. There are three opcodes for the invoke family in the *onCreate* method, so there are three edges from the node. The function call graph for this method is shown in Fig. 10.

2. Representing the function call graph by the subgraph structure of motifs.

It is shown that all graphs can be represented by the subgraph structure of motifs graphs. Milo et al. [33] has shown that the network graph can also use motifs subgraph structural patterns to represent the structural and functional characteristics of graphs. There are thirteen kinds of subgraph structures of motifs graph. However, ten of the motifs are rarely or never occurred in the software, so that we choose three structures with the highest frequency to represent the function call graph, which is shown in Fig. 11. Finally, we stored each function call as a text file, where each row in the file represents a subgraph structure of the motifs, and the contents of each row are the nodes in the subgraph, the node attributes, and the invocation relationships between the nodes.



```
.class public Lcom/example/wuxingru_ui/LoginActivity;
.super Landroid/app/Activity;
.source "LoginActivity.java"
 # virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
   .locals 2
   .param p1, "savedInstanceState"  # Landroid/os/Bundle;
   .prologue
   invoke-super {p0, p1}, Landroid/app/Activity;
   ->onCreate(Landroid/os/Bundle;)V
   const v1, 0x7f030002
   invoke-virtual {p0, v1}, Lcom/example/wuxingru_ui/LoginActivity;
   ->setContentView(I)V
   const v1, 0x7f07000d
   invoke-virtual {p0, v1}, Lcom/example/wuxingru_ui/LoginActivity
   ;->findViewById(I)Landroid/view/View;
   move-result-object v0
   check-cast v0, Landroid/widget/Button;
   .local v0, "button":Landroid/widget/Button;
   new-instance v1, Lcom/example/wuxingru_ui/LoginActivity$1;
   return-void
.end method
```

Fig. 9. Smali code.

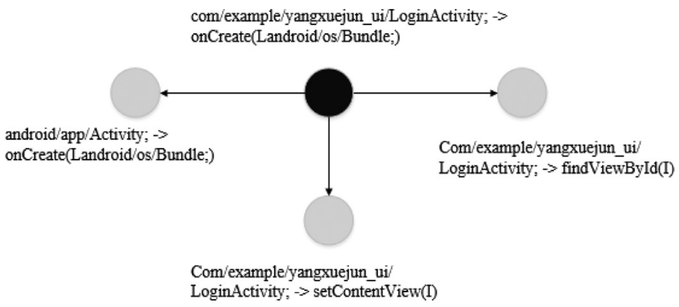

com/example/yangxuejun_ui/LoginActivity; -> onCreate(Landroid/os/Bundle;)
android/app/Activity; -> onCreate(Landroid/os/Bundle;)
Com/example/yangxuejun_ui/ LoginActivity; -> findViewById(I)
Com/example/yangxuejun_ui/ LoginActivity; -> setContentView(I)

Fig. 10. Function call graph of *OnCreate*.

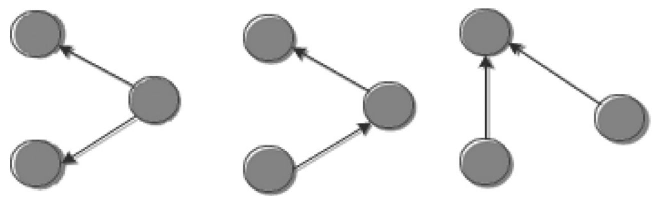


Fig. 11. The three highest frequency structures.

3. Calculating similarity

The similarity between a potential third-party library in an App and a library in our whitelist library database can be determined by comparing the similarity between the two function call graphs. When judging the similarity between the two graphs, we compare the two structure subgraphs of motifs by calculating the number of similar subgraphs. Specifically, we compare how many rows are similar between the two text files, and then calculate the similarity of the function call graph using a similarity formula defined in the following:

Similarity = *Len(S)* / *Len(Lib)*

Lib represents all subgraph structures in a third-party library, and *S* represents the same subgraph structure between a potential third-party library and a real third-party library. *Len (Lib)* is the number of all subgraph structures in a third-party library, and *Len(S)* is the number of the similar subgraph structure between a potential third-party library and a real third-party library.

3.4. Source and sink of privacy leakage

In the Android system, different types of private information are obtained through different APIs. We decompile Apps from the Android App markets and search system APIs that can read private information. We treat these APIs as a *source* of privacy flows.

Privacy data may leave the Android device through the network, SMS, Bluetooth, etc. Among them, the network is the most common leakage port, so we only consider network interface APIs as the *sink*. Our tool hooks functions of the *source* and *sink*, monitoring the invocation of privacy reading and leaking from mobile devices.

3.5. Analyze the caller

3.5.1. Extracting call chains

The *StackTraceElement* class is supported in the *Java.lang* package and can be used to get call stack information for a method. By traversing through the *StackTraceElement* array, we can get the inter-method invocation procedure, and then get the current method and its caller's method name, and so on.

When reaching the privacy *source* and *sink* functions during the execution of an App, the App will trigger hook function in our tool to obtain the call stack through *getStackTrace* method. We treat the host App and third-party libraries as different components (or packages) to analyze the call chain.

The call chain can accurately reflect the invocation relationship between different components in an App. Only knowing the direct caller of a sensitive API is not enough. For example, component *A* can invoke another component *B*, and *B* then invokes *C*, which directly obtains privacy data. To make a correct analysis, we need to know the entire call chain *A->B->C*, instead of *C* alone. In order to make the call chain more clear and concise, we make some restrictions on the call chain log. Method invocation within the same module will not be recorded in the call chain, for example, if a method in Package *A* invokes another method in the same package, which in turns invokes a method in Package *B*, the call chain will be *A->B*.

For each function call, we use the *getClassName* method in *StackTraceElement* to get the package name from the class information. Then, we compare the package name of the current function call to that of the previous call. If they are the same, this function call is not recorded in the call chain; otherwise, we store it in the call chain log.

3.5.2. Inferring the caller

According to the construction rules of the call chain, we analyze the position of the caller of each privacy-sensitive API in the call chain and obtain the callerâs information. Android system libraries and standard develop kits are considered secure and trustworthy. Therefore, in our tool, we match the call chain only with the whitelist of third-party libraries, and Android system libraries and SDKs are excluded in the privacy log.

3.6. Risk assessment of privacy leakage

Based on privacy log, which reflects the third-party libraries' privacy read and leakage path, we develop the risk evaluation criteria for third-party library privacy leakage. We classify the privacy risks of the third-party library and use three levels 0, 1, 2 to represent the degrees of risk, which is shown in Table 8.

Table 8
Privacy leakage risk assessment criteria.

Risk level	Network traffic initiated	No network traffic initiated
Read privacy data	2	1
No privacy data read	0	0

Table 9
The test phone configuration.

Parameter	Configuration
AVD Device Model	Nexus 4
CPU	Snapdragon APQ8064 1536MHZ
Operating system	Android 4.4.4(API-19)
RAM	2GB
Internet connection	192.168.1.232
ROOT	ROOT permission has been obtained

Table 10
Test App set.

Application category	Health	Finance	System tool
Count	50	50	50

A third-party library that does not read private information is considered secure. There is no risk of privacy leakage, so the risk level of this third-party library is labeled as 0. For a third-party library that reads s privacy data but does not initiate a network connection, its privacy leakage risk is low, so it is labeled as 1. For a third-party library that reads the privacy data, and it also initiates a third-party network connection there is a high risk of privacy leakage, so it is labeled as 2. The risk level of a third-party library can be used in fine-grained privacy access control, we will discuss it in more detail in Section 5.

4. Evaluation

4.1. Experiment design

We get root permission for Nexus 4, install Xposed and our tool, and then run our tool as an App. The configuration of the mobile phone for the experiment is shown in Table 9.

We choose the AnZhi App market as a representative to analyze the privacy leakage behaviors of third-party libraries in Apps. According to the functional categories, we download top-ranking Apps and manually test 150 Apps. The App categories are listed in Table 10.

Fig. 12 shows the testing process. For an App in the test set, our tool generates the third-party libraries name list that will be stored in the specified directory of the mobile phone. Then we run the App under test and click on its UI to trigger the application-related functionalities. The test time for an App is 5 minutes. Our tool monitors the App's privacy behaviors in real-time and automatically generates a privacy log and privacy leakage analysis report. After stopping the App under test, we choose the next App in the test set to repeat the test process as before.

4.2. Statistics of third-party library privacy leakage risk

Each privacy leakage path of an App is recorded in detail in the privacy log, and the exposure risk of each privacy leakage path is analyzed and classified according to the privacy leakage scenario, as shown in Fig. 13.

We analyze 150 Apps' privacy logs and categorize privacy flows based on privacy type and privacy leakage path. After removing the duplications, we get 1909 privacy-related call chains. Specific classification statistics are shown in Table 11.

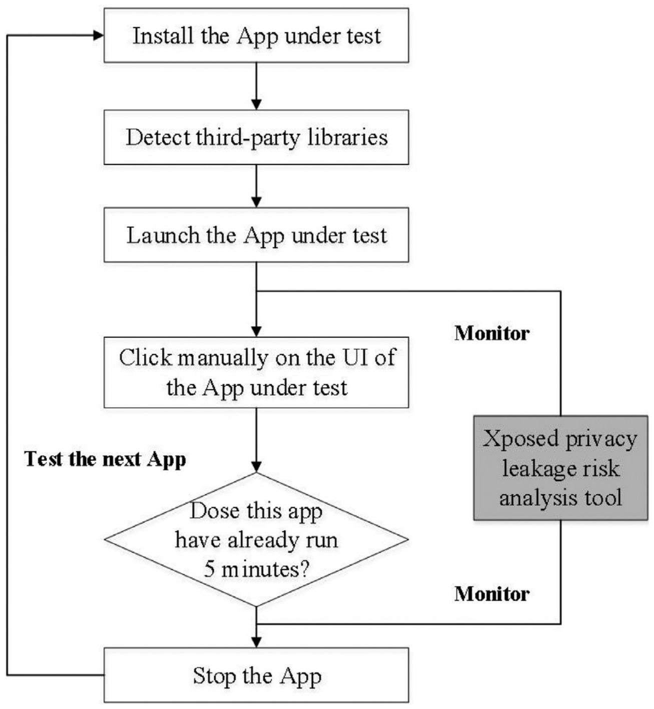
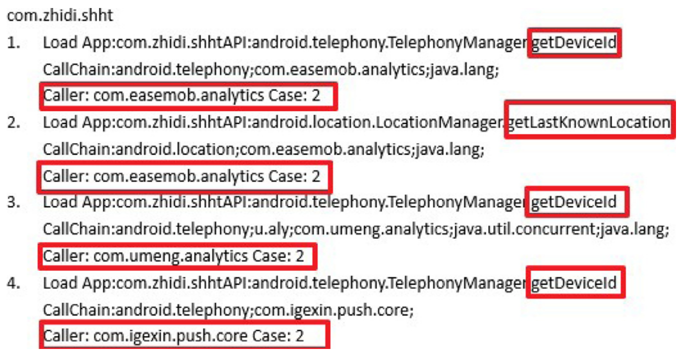

Install the App under test
Detect third-party libraries
Launch the App under test
Monitor
Click manually on the UI of the App under test
Test the next App
Xposed privacy leakage risk analysis tool
Dose this app have already run 5 minutes?
Monitor
Stop the App

Fig. 12. Testing process.



```
com.zhidi.shht
1.   Load App:com.zhidi.shhtAPI:android.telephony.TelephonyManager getDeviceId
     CallChain:android.telephony;com.easemob.analytics;java.lang;
     Caller: com.easemob.analytics Case: 2
2.   Load App:com.zhidi.shhtAPI:android.location.LocationManager.getLastKnownLocation
     CallChain:android.location;com.easemob.analytics;java.lang;
     Caller: com.easemob.analytics Case: 2
3.   Load App:com.zhidi.shhtAPI:android.telephony.TelephonyManager getDeviceId
     CallChain:android.telephony;u.aly;com.umeng.analytics;java.util.concurrent;java.lang;
     Caller: com.umeng.analytics Case: 2
4.   Load App:com.zhidi.shhtAPI:android.telephony.TelephonyManager getDeviceId
     CallChain:android.telephony;com.igexin.push.core;
     Caller: com.igexin.push.core Case: 2
```

Fig. 13. Xposed privacy log.

Table 11
Privacy data leakage statistics.

Privacy type	Case 1	Case 2 & 3	Overall
IMEI	174	502	676
IMSI	125	392	517
Location	31	366	397
Sensor	77	95	172
ICCID	37	38	75
Camera	9	4	13
SMS	1	0	1
Account	12	9	21
Audio	0	2	2
Phone Number	16	19	35

The statistics of the categories of privacy data and privacy leakage paths are shown in Figs 14 and 15. We can see that IMEI, IMSI, location information, are the most frequently leaked information. IMEI information is often used to track users, and Apps usually obtain geographic location information through third-party libraries to ensure a more stable location service. The observation indicates that the previous privacy data protection schemes which deny third-party libraries the permissions to access privacy data are likely to affect the normal functions of the App, so that

Table 12
Privacy behavior statistics of third-party libraries.

Third-party library	IMEI	IMSI	Location	ICCID	Phone number	Sensor	Account	Network flow	Leakage risk
com.igexin.push.core	Yes	Yes	Yes					Yes	2
com.baidu.mobads	Yes	Yes	Yes					Yes	2
com.mob.tools	Yes							Yes	2
com.xiaomi.mipush.sdk	Yes						Yes	Yes	2
com.umeng.analytics	Yes		Yes					Yes	2
cn.jpush.android.service	Yes	Yes	Yes	Yes				Yes	2
com.hyphenate.analytics	Yes		Yes					Yes	2
com.baidu.location	Yes	Yes	Yes			Yes		Yes	2
com.flurry.sdk	Yes		Yes					Yes	2
com.qihoo.util	Yes	Yes		Yes	Yes			Yes	2
com.sina.weibo.sdk	Yes	Yes		Yes				Yes	2
com.baidu.mobstat	Yes		Yes					Yes	2
com.talkingdata.sdk	Yes		Yes	Yes				Yes	2
com.iflytek.cloud	Yes	Yes	Yes					N	1
com.tencent.tauth	Yes					Yes		N	1
com.qq.e.ads.splash	Yes		Yes					N	1

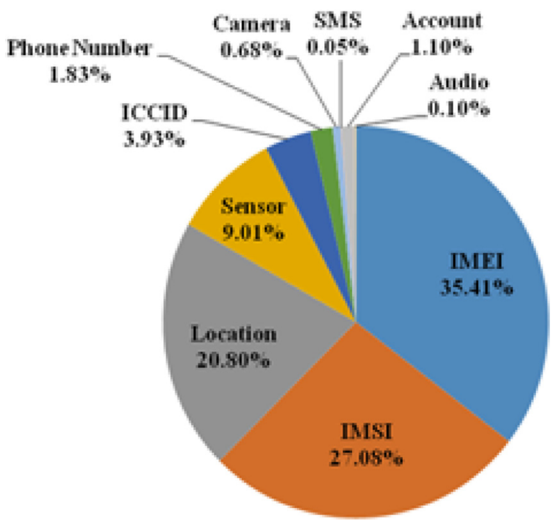


Fig. 14. Privacy data in privacy flows.

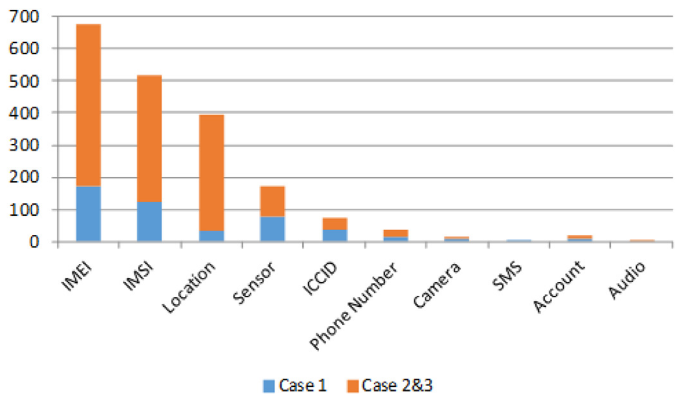


Fig. 15. Leakage path of privacy flows.

these schemes may cause confusion to users in the practical scenario.

We summarize the leaked privacy data of popular third-party libraries' privacy flows, as shown in Table 12.

4.3. Third-party library privacy risk assessment

Take a specific App as an example, according to the privacy leakage risk assessment criteria, we can conduct a privacy risk analysis for the third-party libraries in an App, as shown in Table 13.

4.4. Accuracy evaluation

Based on analysis of the privacy log, 10 applications involving more privacy types are selected from the test set, and static analysis of the privacy leakage path is carried out manually, which is considered as the ground truth. The manual analysis is compared with the results of our tool to calculate accuracy, which is shown in Table 14.

Our tool discovers 156 privacy leakage path. There are 152 correct paths, 4 false paths, and the correct rate reaches 97.4%. The main reason for the errors is that the third-party library detection tools cannot accurately identify the third-party libraries integrated into the application. Matching the call chain with an inaccurate third-party library list results in errors in the analysis result.

Through the complete manual analysis of the API functions used to read privacy in the App, we find that our Xposed privacy monitor module can cover all these APIs, accurately identify the leakage path of privacy data in the running of an App, and will not miss out any privacy flow. So this module can attain high accuracy.

4.5. Performance evaluation

In our tool, the running time of the module of detecting all third-party libraries of an App takes around 10–40 seconds. However, this module executed only once for an App before launching. Meanwhile, the other two modules of monitor and analysis are running while the Apps are running, so we focus on discussing the performance of these two modules.

The privacy log has the records of the time when the privacy APIs are called and when the tool detects the privacy leakage. The time delay of leakage detection is about 28.8 milliseconds, which means that the tool introduces small latency. In addition, we evaluate the overall performance by an Android benchmark tool with four indicators including RAM performance, CPU integer performance, CPU floating point performance and millions of floating point operations per second. The results of the comparison between the system installing our tool and the original system are shown in Table 15. The experiment shows that our tool has little impact on system performance and does not degrade user experience significantly.

5. Discussion

In this section, we discuss a few important issues not addressed in this paper and the future research directions.

Table 13
Third-party libraries privacy risk analysis.

Third-party library	Privacy flow	Network flow	Leakage risk
com.igexin.push.core	IMEI	Yes	2
com.xiaomi.pushsdk	IMEI	Yes	2
com.umeng.message	IMEI,IMSI	Yes	2
com.baidu.mobads	IMEI,IMSI Location	Yes	2
com.mob.tools	IMEI,Location	Yes	2
com.qq.e.ads.splash	IMEI,Location	Not Found	1
com.taobao.accs	IMEI,IMSI	Not Found	1
com.ijinshan.cloud config	Not Found	Yes	0
com.nostra13.universalimageloader	Not Found	Yes	0

Table 14
Accuracy analysis result.

App numbers	Privacy reading paths numbers	Privacy type	Correct numbers	Wrong numbers
10	156	6	152	4

Table 15
Performance evaluation.

Performance indicators	Xposed privacy analysis tool	Android system without our tool
RAM	1737	1641
Mflops	839.79	828.524
CPU Integer	3760	466
CPU Floating Point	1620	1505

In motifs-based library detection, we set 0.5 as the threshold. The logic behind is that if there are more than half the same codes between a potential third-party library and a real third-party library, it indicates that the App is including this library. But when the App uses less than half codes of the real third-party library, the detection tool will fail to report. In the future, we will design a detection tool with the finer grained comparison of similarity.

At present, we analyze the function call to assess the privacy risk of third-party libraries, and we can also add data flow tracking in the future. When the third party library calls the network interface, we will analyze the network traffic flows and detect whether there is privacy data in the flows, which can report the privacy leakage more accurately.

In this paper, we only design monitor and analysis modules, but the privacy protection mechanism is not mentioned. Based on the results of the fine-grained analysis, we will make a more comprehensive assessment of the privacy leakage risk of the Apps, and design practical privacy configuration and enforcement mechanism for users.

6. Conclusions

Libraries inside Apps play an important role in privacy leakage. After we identify four types of privacy leakage paths between components of an App, we propose a dynamic and fine-grained tool for privacy leakage analysis based on Xposed to analyze the privacy leakage behaviors of the third-party libraries in Apps in real time. Our tool first identifies the third-party libraries inside an App, and then extracts call chains of the privacy source and sink functions. By distinguishing between the host App and the third-party libraries, our tool evaluates the privacy leakage risk of the third-party libraries according to the risk assessment criteria. We conduct experiments on 150 Apps and the third-party libraries inside them. The experimental results show that the third-party libraries have different behaviors in privacy leakage, and pose different privacy leakage risk to both App developers and users.

Acknowledgments

The work reported in this paper is supported in part by National Key Research and Development Program of China, under grant 2017YFC0820100, 2017YFB0802805, and in part by Natural Science Foundation of China, under Grant U1736114.

References

[1] http://www.199it.com/archives/592840.html.
[2] Grace MC, Zhou W, Jiang X, Sadeghi A-R. Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on security and privacy in wireless and mobile networks (WISEC'12); 2012.
[3] 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); 2014.
[4] Ma Z, Wang H, Guo Y, Chen X. Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th international conference on software engineering (Demo Track), ser. ICSE'16 Companion Volume; 2016. p. 653–6.
[5] Li M, Wang W, Wang P, et al. Libd: scalable and precise third-party library detection in android markets. In: International conference on software engineering; 2017. p. 335–46.
[6] Duan R, Bijlani A, Xu M. Identifying open-source license violation and 1-day security risk at large scale. In: Proc. ACM conference on computer and communications security, CCS; 2017.
[7] Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[c]. In: ACM Sigplan conference on programming language design and implementation. ACM; 2014. p. 259–69.
[8] Zhao Z, Osono FCC. Trustdroid: preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking[c]. In: International conference on malicious and unwanted software. IEEE; 2012. p. 135–43.
[9] Enck W, Gilbert P, Chun BG, et al. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones[j]. ACM Trans Comput Syst 2014;32(2):1–29.
[10] Qian C, Luo X, Shao Y, et al. On tracking information flows through jni in android applications[c]. In: Dependable systems and networks (DSN), 2014 44th annual IEEE/IFIP international conference on. IEEE; 2014. p. 180–91.
[11] Hornyack P, Han S, Jung J, Schechter S, Wetherall D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: In Proc. ACM conference on computer and communications security. CCS; 2011.
[12] Yang Z, Yang M, Zhang Y, et al. Appintent:analyzing sensitive data transmission in android for privacy leakage detection[c]. In: ACM Sigsa conference on computer & communications security. ACM; 2013. p. 1043–54.
[13] Book T, Pridgen A, Dan SW. Longitudinal analysis of android ad library permissions[j]. Computer science; 2013.
[14] Grace MC, Zhou W, Jiang X, et al. Unsafe exposure analysis of mobile in-app advertisements[c]. In: ACM conference on security and privacy in wireless and mobile networks. ACM; 2012. p. 101–11.
[15] Stevens R, Gibler C, Crussell J, et al. Investigating user privacy in android ad libraries[j] In IEEE MOST 2012; 2012.
[16] Livshits B, Jung J. Automatic mediation of privacy-sensitive resource access in smartphone applications[c]. In: Usenix conference on security. USENIX Association; 2013. p. 113–30.
[17] Bhoraskar R, Han S, Jeon J, et al. Brahmastra: driving apps to test the security of third-party components[c]. In: The, Usenix security symposium; 2014. p. 1021–36.
[18] Crussell J, Stevens R, Chen H. MADfraud: investigating ad fraud in android applications[m]. In: Proceedings of the 12th annual international conference on mobile systems, applications, and services; 2014.
[19] Pearce P, Felt AP, Nunez G, et al. Addroid: privilege separation for applications and advertisers in android[c]. In: Proceedings of the 7th ACM symposium on information, computer and communications cecurity. ACM; 2012. p. 71–2.

[20] Liu B, Jin H, et al. Efficient privilege de-escalation for ad libraries in mobile apps[c]. In: The, international conference; 2015. p. 89–103.
[21] Wang Y, Hariharan S, Zhao C, et al. Compac: enforce component-level access control in android[c]. In: ACM conference on data and application security and privacy. ACM; 2014. p. 25.
[22] Wang X, Wang W, He Y, Liu J, Han Z, Zhang X. Characterizing android apps' behavior for effective detection of malapps at large scale. Future Gener Comput Syst 2017;75:30–45.
[23] Wang W, Wang X, Feng D, Liu J, Han Z, Zhang X. Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans Inf Forensics Secur. 2014;9(11):1869–82.
[24] Su D, Liu J, Wang X, Wang W. Detecting android locker-ransomware on chinese social networks. In: IEEE Access 7; 2019. p. 20381–93.
[25] Wang W, Li Y, Wang X, Liu J, Zhang X. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener Comput Syst 2018;78:987–94.
[26] Wang W, Gao Z, Zhao M, Li Y, Liu J, Zhang X. Droidensemble: detecting android malicious applications with ensemble of string and structural static features. IEEE Access 2018;6:31798–807.
[27] Wang W, Liu J, Pitsilis G, Zhang X. Abstracting massive data for lightweight intrusion detection in computer networks. Inf Sci 2018;433(434):417–30.
[28] Liu X, Liu J, Wang W, He Y, Zhang X. Discovering and understanding android sensor usage behaviors with data flow analysis. World Wide Web 2018;21(1):105–26.
[29] Liu X, Liu J, Zhu S, Wang W, Zhang X. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. IEEE Trans Mob Comput 2019. doi:10.1109/TMC.2019.2903186.
[30] https://repo.xposed.info/.
[31] Li L, Bissyande TF, Klein J. An investigation into the use of common libraries in android apps[c]. In: International conference on software analysis. IEEE; 2016. p. 403–14.
[32] Hu W, Octeau D, McDaniel P.D, Liu P. Duet: library integrity verification for android applications. In: Proceedings of the 2014 ACM conference on security and privacy in wireless & mobile networks. 141–152.
[33] Milo R, Shen-Orr S, et al. Network motifs: simple building blocks of complex networks. Science 2002;298(824).