This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android Ecosystem

Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang Xiangliang Zhang

Abstract—While much effort has been made to detect and measure the privacy leakage caused by the advertising (ad) libraries integrated in mobile applications, analytics libraries, which are also widely used in mobile apps have not been systematically studied for their privacy risks. Different from ad libraries, the main function of analytics libraries is to collect users' in-app actions. Hence, by design analytics libraries are more likely to leak users' private information. In this work, we study what information is collected by the analytics libraries integrated in popular Android apps. We design and implement a framework called "Alde". Given an app, Alde employs both static analysis and dynamic analysis to detect the users' in-app actions collected by analytics libraries. We also study what private information can be leaked by the apps that use the same analytics library. Moreover, we analyze apps' privacy policies to see whether app developers have notified the users that their in-app action data is collected by analytics libraries. Finally, we select 8 widely used analytics libraries to study and apply our method to 300 popular apps downloaded from both Chinese app markets and Google play. Our experimental results show that some apps indeed leak users' personal information through analytics libraries even though their genuine purposes of using analytics services are legal. To mitigate such threats, we have developed an app named "ALManager" that leverages the Xposed framework to manage analytics libraries in other apps.

Index Terms—Android, analytics libraries, privacy leakage

1 Introduction

According to the statistical result from AppBrain [2], the number of apps in Google Play has reached 3.5 million. The sheer number of apps that are in the Google Play and the number of new ones added daily only show that the mobile app ecosystem has become a gigantic marketplace that is keeping expanding. It is more and more difficult for app developers to make their apps stand out. Hence, it is increasingly important for developers to understand their users and improve their apps for the users.

For this purpose, developers need a way to collect the data on *user's in-app actions*, such as opening an app, browsing different pages in the app, pressing a button in the app, etc. By analyzing the collected data, developers can understand how the users use their apps, what app functions attract users and what not, and what problems they may be experiencing. Such knowledge can help the developers improve their designs and also fix known problems (as shown in Figure 1), thus enhancing the users' experience. Hence, almost every popular app contains code snippets to collect and analyze users' in-app actions. Some developers implement the collecting and analysis functions by themselves, while others implement these functions with the help of some third-party libraries. We call a third-party library that is used to collect and analyze the users' in-app actions "a third-party analytics library", or "analytics library" in short.

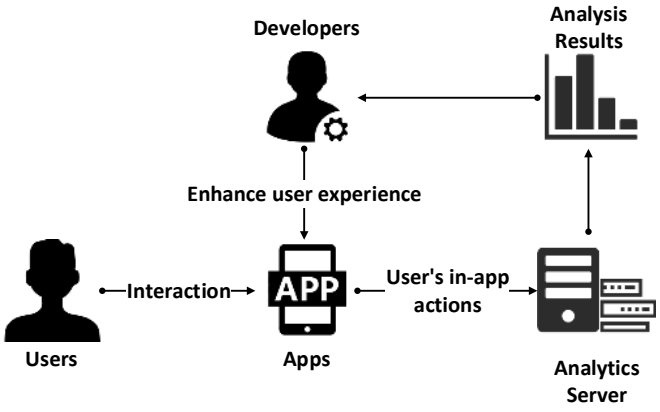

Fig. 1. Mobile applications analytics system.

Analytics libraries are similar to ad libraries in some aspects. For example, both analytics libraries and ad libraries are integrated with the host apps. Host apps and the libraries share privileges and resources. They have the same Linux file access control permissions and Android permissions. Both analytics library and ad library require some permissions that may not be needed by the host app. Therefore, analytics libraries may cause security and privacy issues similar to that caused by ad libraries [3], [4], [5]. However, ad libraries do not require developers to do many settings. Take AdMob's banner ads [6] as an

- *X. Liu, J. Liu and W. Wang, are with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, 3 Shangyuancun, Beijing 100044, China. E-mail: {xingliu, jqliu, wangwei1}@bjtu.edu.cn*
- *S. Zhu is with Department of Computer Science and Engineering, The Pennsylvania State University, PA 16801, USA. E-mail: szhu@cse.psu.edu*
- *X. Zhang is with Division of Computer, Electrical and Mathematical Sciences & Engineering, King Abdullah University of Science and Technology, Saudi Arabia. E-mail: xiangliang.zhang@kaust.edu.sa*

Corresponding author: Wei Wang

This paper is an extension of our previous work, Alde: Privacy Risk Analysis of Analytics Libraries in the Android Ecosystem [1], published in SecureComm 2016

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

example. Developers only need to add an ad view in their apps and set up the corresponding ad unit ID [7]. Then the ad library will automatically request ads and display them in the ad view. Developers do not need to care about the ads' content. Though ad libraries have provided some ad control APIs, many developers do not use them [8], [9]. In contrast, when developers use analytics libraries to collect users' in-app actions, developers need to invoke the tracking APIs provided by the analytics libraries at locations they want [10]. For example, developers may invoke the tracking APIs to collect a user's payment action after the user clicks a payment button. In other words, what information to collect is set by developers. The more a developer wants to profile his users, the more tracking points he needs to set up in his app.

After collecting users' in-app action data, analytics libraries send it to the analytics companies, which analyze the data and present analysis results to developers. Now, curiosities are aroused on what private information is leaked to analytics companies and to app developers, respectively, from the collected data. This problem is exacerbated because analytics libraries may collect unique device information (IMEI, MAC, etc.) that can be used to link the information collected by different apps together to get a more comprehensive record of users' activities. Previous studies on app privacy, however, only concerned the information protected by Android permissions or information input by users (e.g., account number, password) [11], [12]; therefore, they cannot answer the very question we are facing. As a first step in the direction of answering this question, we explore the users' in-app action data collected by the analytics libraries in the popular apps. Specifically, we design and implement an analysis framework called "Alde" (Analytics libraries data explorer), which employs both static analysis and dynamic analysis to discover the users' in-app actions collected by analytics libraries. In the static analysis process, Alde performs a backward taint analysis based on the app's smali codes [13]. This backward taint analysis aims to find out what hardcoded information is sent to the tracking APIs. In the dynamic analysis process, we hook the tracking APIs to explore what information is sent to these APIs at the app's running time. As an extension of our previous work [1], we propose an obfuscated API finder to deal with the apps that have obfuscated the tracking APIs they used. After we obtain the users' in-app actions collected by the analytics libraries, we manually review the data to determine what personal information is leaked to the analytics companies. We also analyze the apps' privacy policies to check whether they notify the users about such data collection. We select 8 widely used analytics libraries for study and apply our method to 300 apps downloaded from both Wandoujia (a Chinese app market) and Google Play. The experimental results show that i) analytics libraries can be exploited by malicious developers to directly collect users' personal information; ii) some apps indeed leak users' personal information to analytics companies even though their genuine purposes of using analytics libraries are legal; iii) users will be deeply profiled if analytics companies link the information collected from different apps, especially in China; iv) developers seldom describe the use of analytics libraries in their apps' privacy policies even though they are requested to do so. To mitigate such threats and give users the ability to control the analytics libraries, as an extension of our previous work [1], we develop an app called "ALManager", which leverages the Xposed framework [14] to manage the analytics libraries. In summary, we make the following contributions in this work:

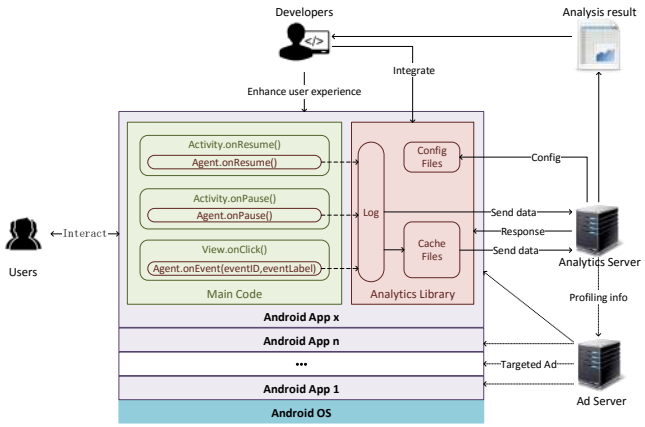

Fig. 2. Structural overview of a mobile third-party analytics library.

- To the best of our knowledge, our work is the first research focusing on understanding information leakage caused by users' in-app action data collected by analytics libraries.
- We design and implement a framework named "Alde" that can be used to discover the users' in-app action data collected by analytics libraries.
- We apply our method to 300 apps downloaded from both Wandoujia and Google Play, and reveal the data collected by the analytics libraries integrated in these apps. Based on the data, we discover various types privacy risks.
- We develop an app named "ALManager" to mitigate the privacy risk caused by analytics libraries.

The remainder of this paper is organized as follows. Section 2 describes the background of Android analytics libraries. Section 3 present the system design and implementation of Alde. Section 4 describes the dataset that we use in this study. Section 5 describes the experimental results. The design and implementation of ALManager and related work are given in Section 6 and 7, respectively. Section 8 concludes our work.

2 Background

2.1 Analytics Libraries

Analytics libraries are important tools that mobile app developers commonly employ in their apps. Through them, analytics companies provide mobile app developers well analyzed data that shows how the users are using their apps. To understand how an analytics library is embedded into an Android app, we provide a simplified structural overview of the mobile analytics library through Figure 2.

Take Umeng [15], the most popular analytics library in China, as an example. In order to integrate this library into

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

their apps and obtain the analysis results, developers need to take the following steps:

1) Register an account at the analytics company and log in. Then, a developer is required to set up the basic information (name, category, etc.) of the app that he wants to track. After the setup for the app, the analytics company will generate a unique AppKey. This AppKey will be utilized to identify the app.
2) Add the SDK provided by the analytics company into the app's build path. Then, edit the app's AndroidManifest file and add the unique AppKey into the app's metadata. Moreover, the developer is required to add the permissions required by the analytics library into the AndroidManifest file.
3) Initialize the analytics library. Commonly, a developer needs to invoke the initialization method provided by the analytics library to initialize the library when the app is launched.
4) Invoke the tracking APIs provided by the analytics library to collect users' in-app action. For example, with the Umeng library, a developer can invoke *MobclickAgent.onResume()* and *MobclickAgent.onPause()* in each activity's *onResume()* and *onPause()* methods to collect each activity's start time and end time. He can invoke *MobclickAgent.onEvent(...)* to collect the users' in-app actions of his interest. For instance, if the developer wants to know how many users are interested in movies in a video app he developed, he can invoke this method (triggered when the users press the "movies" button) and set the parameter *eventID* as "movies". Developers can also set up the analytics library to automatically collect the run-time errors occurred in the app.
5) Upload the app to Android app market(s). When users download and enjoy this app, the analytics company will receive users' in-app actions data, analyze it and present the analysis results to the app's developer through a web interface.

The steps described above are the common procedures that developers need to follow if they want to use an analytics library. Although most analytics libraries can be used successfully like this, different analytics libraries are different in implementation details. Hence, the processes of integrating different analytics libraries into the apps are not totally the same. Additionally, some new analytics libraries (such as *Appsee*[1] and *UXCam*[2]) use a totally different method to collect users' in-app actions. They do not require developers to invoke tracking APIs to collect users' actions. Instead, they collect all the interactions between users and apps as videos and show the videos to the developers directly. We do not consider this kind of analytics libraries in this paper.

2.2 What Information is Presented to Developers

When users play with apps, their in-app actions data is collected by analytics libraries and sent to the analytics servers. It is analyzed automatically in the analytics servers, which presents the analysis results to developers. In Table 1, we list the information that developers can learn from the analysis results. Besides the basic information shown in this list, analytics companies also present some statistical information, such as *User growth rate*, *User retention*, *User loyalty*, *Event conversion rate*, etc. This statistical information can be presented in different time periods: by day, by week, by month, or by year, which helps developers learn whether their apps are popular or not in a period, or whether the new functions they added in the apps attract more users. Data presented to the developers is the statistical analysis results based on *all* users. In principle, developers cannot get the raw data of an individual user's in-app actions.

TABLE 1
The information that developers obtain about their apps

Categories	Details
Users	Total users, New users, Returning users, Active users, Launch times, Launch frequency, Duration of once use, Activity path, App Versions
Terminals	Devices, Resolutions, OS versions, Carriers, Area, Languages
Events	Event IDs, Event labels, Event times, Event values
Errors	Error summary, Error times, First appearance time, Last appearance time

2.3 Differences between Mobile App Analytics and Web Analytics

Compared with the traditional web analytics, the privacy threat is heightened on mobile app analytics. First, in web analytics, a user is authenticated through cookies and IP addresses, whereas in mobile app analytics, the user is authenticated through device ID. Cookies can be deleted by the user easily, but device ID is hard to change. Besides, people (such as family members) may share the same computer to browse the websites. Web analytics cannot know exactly whether a web page's browsing behavior comes from the same user or not. But, since people usually have their own mobile phones, data collected by mobile analytics on a device usually comes from the same user. Second, in web analytics, browsers are able to distinguish the host websites from the third-party analytics sites based on their domain names. Analytics sites can only access the content presented in the web browser and some basic information about the browser and computer. However, in mobile app analytics, analytics libraries have the same permissions as their host apps. They can access many other resources on the phone with the permissions granted to the host apps. Third, in the web analytics, users often open many pages at the same time and do not close the pages that they are not actively viewing. The analytics sites do not know whether the pages are actually viewed by the user or not, while in the mobile app analytics, the app running in the front is usually actually viewed by the user [16] [17].

1. https://www.appsee.com
2. https://uxcam.com

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

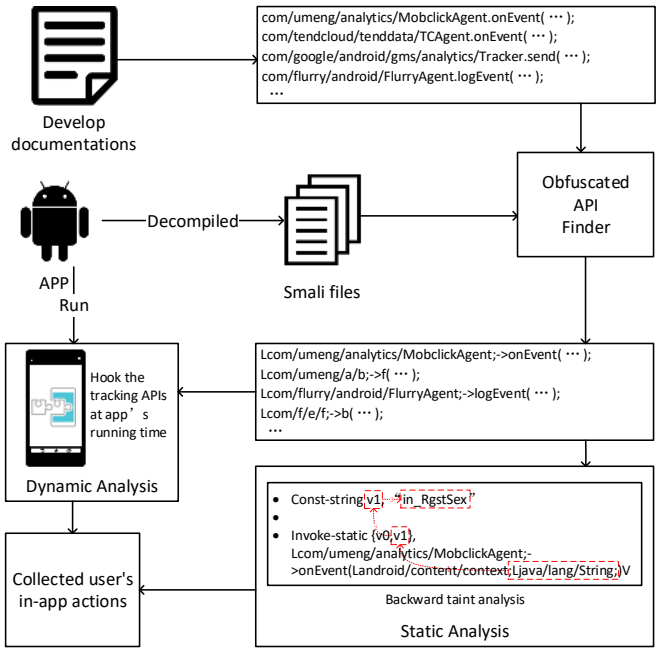

Fig. 3. System Overview of Alde.

3 System Design and Implementation of Alde

To understand what private information can be leaked by the analytics libraries integrated with the popular apps, we develop "Alde", a system for this purpose. Alde uses both static analysis and dynamic analysis to discover the values of the tracking APIs' parameters, which are the users' in-app actions collected by these tracking APIs. The system's overview is illustrated in Figure 3.

3.1 Documentation Analysis

As we described in Section 2.1, developers need to invoke the tracking APIs provided by analytics libraries to collect users' in-app actions. Hence, our first step is to determine the tracking APIs provided by each analytics library. We obtain this information by analyzing the development documentation provided by each analytics library. However, some analytics libraries only give a brief description of the tracking APIs in their development documentation. The complete class names (including class package names) of the tracking APIs that are needed in the following processes are not given. To address this problem, we download some apps that contain these analytics libraries, decompile them with Apktool [18], and find out the complete class names of the tracking APIs from the decompiled code.

3.2 Obfuscated API Finder

Identifier renaming based obfuscation is widely used in popular Android apps. App's package, class, method and field names are obfuscated into meaningless strings. For example, method *MobclickAgent;→onEvent(...)* may be obfuscated into *b;→f(...)*. Although most apps only obfuscate their main code, some apps obfuscate the tracking APIs they used. In order to analyze the apps that have obfuscated the tracking APIs, we design and implement an obfuscated API finder to recognize the obfuscated tracking APIs in these apps. Since the identifier renaming obfuscation does not change an app's method call graph, the obfuscated API finder recognizes the obfuscated tracking APIs by their method call graphs.

TABLE 2
List of instruction categories and their representing characters

category	character	category	character
JUMP	J	NEW	N
INVOKE	I	PUT	P
MATH	M	CONSTANT	C
GET	G	MOVE	V
GOTO	T	SWITCH	W
RETURN	R	ARRAY	A
Other smali instructions			S

For a given analytics library, we extract its tracking APIs' method call graphs from the apps that do not obfuscate these tracking APIs' names. Given an app that uses this analytics library and does not obfuscate the tracking APIs, we decompile the app into smali code files and analyze all the smali files to generate this app's method call graph. The generated method call graph is a directed graph. Nodes represent the app's methods and edges represent the calling relationships from callers to callees. In addition, each node of the method call graph is assigned with a reference string that characterizes the method's content. The reference string is generated from the method's instruction sequence. Each instruction in the method body is mapped to a character. The mapping rule is shown in Table 2. If a method is a system method and does not have analyzable method body, the reference string assigned to this method is an empty string. After the app's method call graph is generated, we extract the tracking API's method call graph from it. As shown in Figure 4, a tracking API's method call graph is a subgraph of the app's method call graph. It is a series of method call sequences start from the tracking API. The maximum length of the method call sequence considered in our work is 4. This is because the length of most method call sequences of each tracking API is equal to or less than 4. For each tracking API in each analytics library, we extract its method call graph. Since there may be some differences in the same tracking APIs from different versions of an analytics library, a tracking API may correspond to more than one method call graph.

Given an app that has obfuscated the tracking APIs it used and a known tracking API, the obfuscated API finder detects whether the given tracking API is used in the given app through the following steps.

1) Decompile the app and generate the app's method call graph.
2) Simplify the method call graph by deleting the nodes that are certainly not tracking APIs, for example, nodes belonging to other known third-party libraries.
3) Assign each node of the simplified method call graph a reference string that characterizes the method's content. The reference string generation rule is the same as described before.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

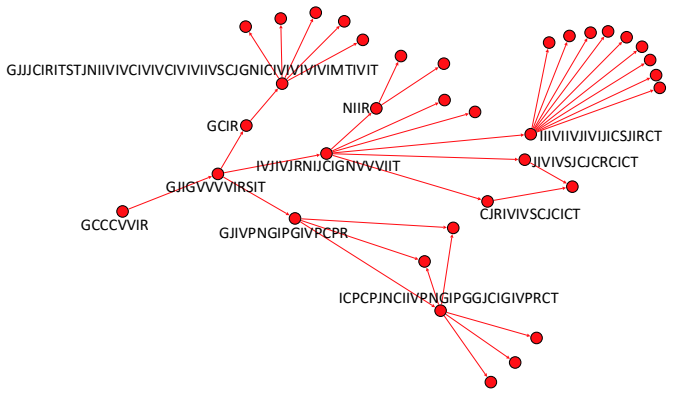

Fig. 4. Method call graph of method *MobclickAgent.onEvent(context,String)* from Umeng analytics library.

4) For each method (node) in the simplified method call graph, the obfuscated API finder compares the method's reference string and parameter type with that of the given tracking API. If they match, the obfuscated API finder extracts this method's method call graph (the maximum length of the method call sequences is 4, too) and compares its method call graph with that of the given tracking API. The algorithm for comparing the two graphs is shown in Algorithm 1. In this algorithm, the function *getMethodCallSequenceList()* convents the method call graph into a method call sequence list (as shown in Figure 5). If their method call graphs also match, the obfuscation API finder records this method as an obfuscation of the given tracking API.

Algorithm 1 Compare whether two graphs are the same

Input: G_1 and G_2
Output: **true**: G_1 and G_2 are the same. **false**: G_1 and G_2 are not the same.

```
L1 ← G1.getMethodCallSequenceList()
L2 ← G2.getMethodCallSequenceList()
if L1.length() ≠ L2.length() then
    return false
end if
for l in L1 do
    if l in L2 then
        continue
    else
        return false
    end if
end for
return true
```

In order to evaluate the effectiveness of this obfuscated API finder, we conduct an experiment on 50 open source apps downloaded from F-Droid, an Android open source project sharing website[3]. We add some tracking APIs into each app and configure each app's *proguard-rules.pro* file to obfuscate these tracking APIs. Then, we use our obfuscated API finder to detect these obfuscated tracking APIs. All of these obfuscated tracking APIs are successfully detected,

3. https://f-droid.org/en/

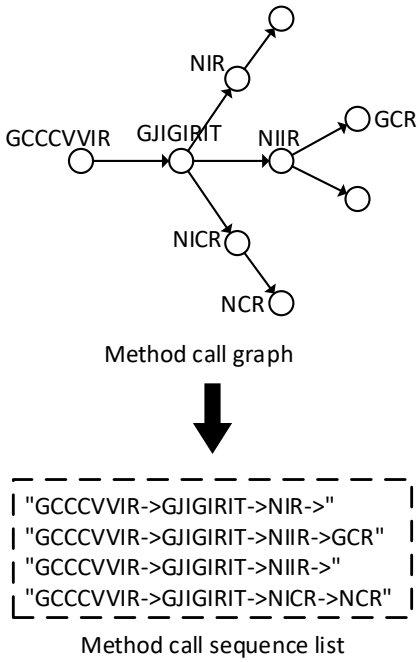

Fig. 5. Convert a method call graph into a method call sequence list.

which indicates that this obfuscated API finder can effectively identify the tracking APIs obfuscated via identifier renaming.

This approach aims to detect identifier renaming based obfuscation processed by ProGuard [19]. Other obfuscation techniques that modify the call graph structure (inlining, outline, reflection injection) indeed affect the detection results. The effectiveness of this approach in practice will be discussed in Section 4.2.

3.3 Static Analysis

Some information collected by the tracking APIs is hardcoded in the app's source code, such as names of some buttons. Static analysis aims to discover the users' in-app actions hardcoded in the app's source code. Alde performs static backward taint analysis to find out the values of the tracking APIs' parameters based on the app's smali code. As shown in Figure 3, given an app, Alde carries out the following backward taint analysis.

1) Alde decompiles the app into smali code files with Apktool. Then, it analyzes the smali code files to generate the app's method call graph and each method's control flow graph.
2) Alde finds out the corresponding smali codes of the tracking APIs and identifies the registers that store the values of the tracking APIs' parameters. Then, Alde starts a backward taint analysis from the identified registers. Alde searches the smali code in the reverse order to backward propagate taint. The taint propagation rules are based on the semantic of Dalvik bytecode [20]. For example, consider the Dalvik instruction: *move v1, v2*. The semantics of this instruction specify that the value in register *v2* is moved to register *v1*. In our backward taint analysis, if the value of register *v1* is tainted, then

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

taint register $v2$. If a tainted register is assigned with an insensitive value, the taint will be removed from this register and the backward taint analysis of this register will stop.

3) This backward taint propagation process will not stop until Alde finds a constant value is assigned to the tainted register or Alde traces into a method that cannot be analyzed by Alde. Last, the final constant value and the backward taint propagation path are reported. As shown in Figure 3, the final constant value of $v1$ is "in_RgstSex".

The code snippet shown in Figure 3 appeared in a fitness app. When users go to the *Gender* setting page, this code snippet will run and collect this in-app action.

3.4 Dynamic Analysis

Though the above static analysis can explore the in-app actions defined and hardcoded in an app's source code, some information is only generated at app's running time, so it cannot be captured by static analysis. Hence, Alde also performs dynamic analysis on the app.

In the dynamic analysis process, Alde runs the app for 5 minutes with the help of AndroidViewClient [21]. Developed with python, AndroidViewClient is a test framework for Android apps and is more powerful than monkeyrunner [22]. AndroidViewClient runs on a computer and connects to an Android device through USB debugging. AndroidViewClient can be used to obtain the UI structure of a running app and sent simulated user operations to the device. We write a python script based on AndroidViewClient to automatically run Android apps. Given an app, Alde runs it according to the process described in Figure 6. A "view" means an element in an activity, such as button, text-area, picture, etc. At the same time, the tracking APIs are hooked by Alde with the help of Xposed framework.

Xposed Framework is a widely used framework running on the rooted Android device. With the APIs provided by the Xposed Framework, we can develop modules to hook any Java methods running on the device for monitoring, modifying or replacing specific methods. In Android platform, both the app process and the system service process are hatched by the Zygote process. When Zygote process starts, it will load some necessary resources such as some Android core classes and some Java runtime libraries. App processes and system service processes will inherit these resources when they are hatched by the Zygote process. The code executed by the Zygote process is located in file */system/bin/app_process*. Through the root permission, Xposed Framework replaces the original *app_process* file with its own customized *app_process* file to control the Zygote. This makes the Zygote process load a jar file named *XposedBridge.jar* provided by Xposed Framework during startup. Since other app processes and system service processes will inherit the resources loaded by Zygote, *XposedBridge.jar* is embedded into every app. Through the code in *XposedBridge.jar*, Xposed Framework modifies the method needed to hook as a native method *xposedCallHandler*. Native methods will be invoked before Java methods. Native method *xposedCallHandler* will invoke Java method *handleHookedMethod*, and *handleHookedMethod* will invoke two call back methods named *beforeHookedMethod* and *afterHookedMethod*, which are implemented by developers. These two methods are invoked before and after the hooked method, respectively. Methods that are needed to be hooked and the modifications to the hooked methods are defined in Xposed modules. Xposed modules are installed into the device as apps. Xposed Framework will load the selected modules after device reboot.

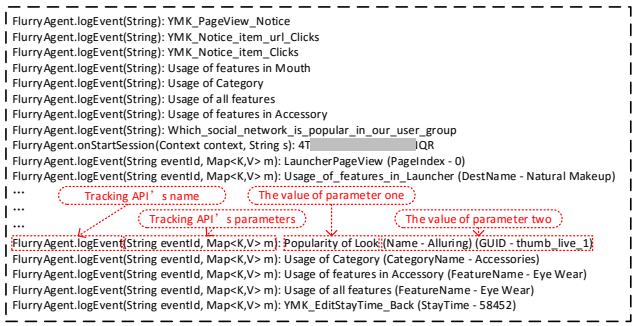

Fig. 7. Part of the analysis results of app "YouCamMakeUp".

We develop a module for Xposed framework to hook the tracking APIs. When the app under analysis invokes a tracking API, the values of the API's parameters will be captured by Xposed framework and stored in the files located in the phone's external SDCard. When the app stops running, we pull these files from the phone. Through this method, we get the users' in-app actions that are collected by the analytics libraries at the app's running time.

For the apps that ask the users to register an account, we register the account manually and then analyze it with Alde. After the entire analysis process of an app is finished, we merge the analysis results from both static analysis and dynamic analysis to get the final analysis results (as shown in Figure 7).

4 Dataset

In this section, we describe the dataset that we use in this study. We download 200 popular apps from a Chinese app market named "Wandoujia" and 100 popular apps from Google Play. All these apps are free apps.

4.1 Analytics Libraries

We focus on 8 widely used analytics libraries, listed in Table 3. To select these widely used analytics libraries, we search the Internet for popular analytics libraries and also learn from previous studies [23]. After this, we get a list of popular analytics libraries. Then, we search these analytics libraries' package names in the smali code that is decompiled from the apps we downloaded. If an analytics library's package name appears in an app, we conclude that the app uses this library. Finally, we select 8 most widely used analytics libraries in our dataset. Four of them are mainly used by the apps in the Chinese app markets and the other four are mainly used by the apps in Google Play. In the rest of this paper, we call them analytics libraries China and analytics libraries Google Play, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

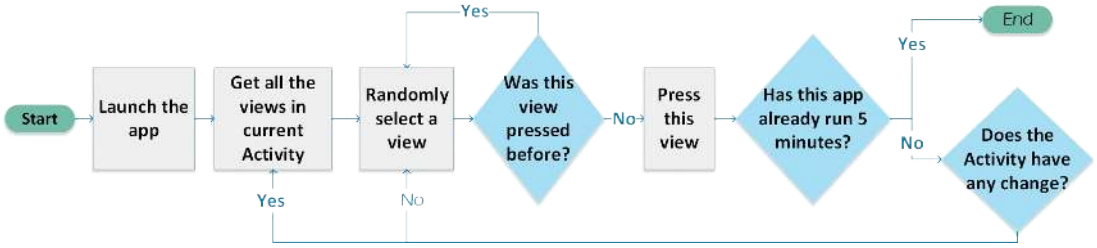

Fig. 6. Flowchart of dynamic app running in Alde.

TABLE 3
Analytics libraries' required permissions and optional permissions. "✔" means required permission and "●" means optional permission.

	Umeng	Talkingdata	Tencent Analytics	Baidu Analytics	Flurry	Adjust	Localytics	Google Analytics
INTERNET	✔	✔	✔	✔	✔	✔	✔	✔
ACCESS_WIFI_STATE	✔	✔	✔	✔		●		
ACCESS_NETWORK_STATE	✔	✔	✔	✔	●			✔
READ_PHONE_STATE	✔	✔	✔	✔				
WRITE_EXTERNAL_STORAGE		✔	✔	✔				
WRITE_SETTINGS				✔				
GET_TASKS		●		✔				
READ_EXTERNAL_STORAGE				✔				
MOUNT_UNMOUNT_ FILESYSTEMS				✔				
ACCESS_FINE_LOCATION		●		●	●			
ACCESS_COARSE_LOCATION		●			●			
BLUETOOTH				●				
WAKE_LOCK							✔	

Table 3 also shows the permissions required by these 8 analytics libraries as well as their optional permissions. Analytics libraries China commonly require more permissions. This is because they need the device information (IMEI, MAC, etc.) to generate the ID that is used to identify the individual device. They also need to know the network state and WIFI state in order to adjust the interval of sending collected data to their servers. They may also need to store some cache files in the external storage. Analytics libraries Google Play can do the similar things with the help of Google Play Service, which is not available in China. However, these permissions also empower the analytics libraries from Chinese app markets to collect more information than they need.

4.2 App Selection

As described in Section 3, our method needs to know where the tracking APIs are invoked. If an app obfuscates the tracking APIs it used, we need to identify the names of the obfuscated tracking APIs. We perform an **API search process** (i.e., searching tracking APIs' names in apps' smali code) to filter out the apps we can analyze directly. If a tracking API provided by an analytics library appears in an app's smali code, we consider this app as using this analytics library and it can be analyzed by our method. To understand how many apps are missed by our method, we carry out another **file search process** to determine the analytics libraries used by each app. In this file search process, we launch each app on a device and determine what analytics libraries it uses based on the files generated at the app's running time. This is because different analytics libraries generate different files (such as database files, cache files, Shared_prefs files) at their running time. The generated files' names are not influenced by code obfuscation. For the apps that are not found by the API search process but found by the file search process, we try to identify the obfuscated tracking APIs they use with the obfuscated API finder described in Section 3. We present the filtering result in Figure 8. For each analytics library in the figure, there are two numbers (in red color) shown besides its group of four bars, which represent the number of apps that get the same analysis results via both API search process and file search process, in Chinese app market and in Google Play, respectively.

Figure 8 shows that our method can analyze most of the apps. With the obfuscated API finder, we identify 25 apps that have obfuscated the tracking APIs' names and use the obfuscated API names in the subsequent analysis process. Through these apps, we find that the impact of identifier renaming based obfuscation on third-party analytics library usually has the following three aspects. First, the names of the tracking APIs are changed to single characters or other meaningless strings. Second, some unused tracking APIs and related classes may be removed. For example, Umeng provided seven different tracking APIs. App *com.iyd.reader.ReadingJoy Version 6.0*[4] only used two of

4. Some Chinese apps do not have the corresponding English name, we use their package names instead.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

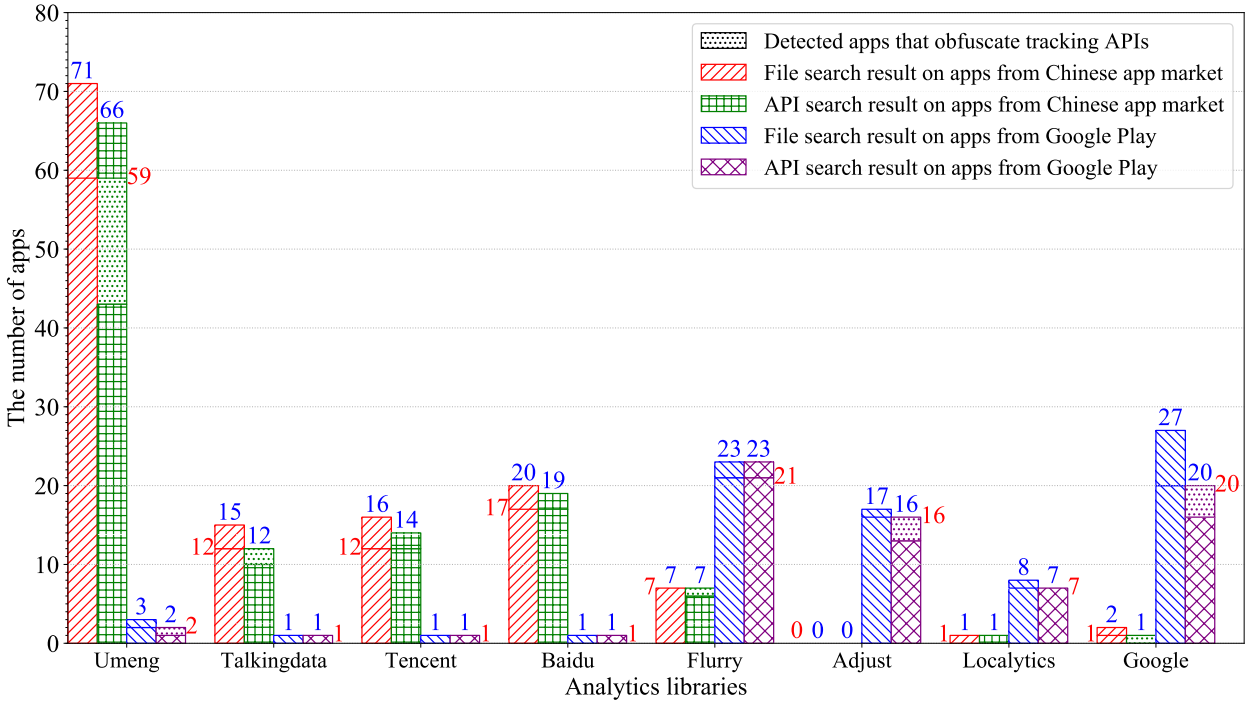

Fig. 8. The number of popular apps containing each of the analytics libraries.

them. After obfuscating, five unused tracking APIs and some classes related to these APIs were removed from the app's code. Third, the package structure of the analytics library may be changed. For example, the original tracking API *MobclickAgent.onEvent* from Umeng library was located in package *com.umeng.analytics*. However, in app *com.hipu.yidian Version 5.0.1*, this tracking API was obfuscated to *on.a* located in the root package.

In order to assess the effectiveness of the obfuscated API finder, we manually analyze the apps that are found by the file search process but not found by the API search process. We find that these tracking APIs were not identified due to the following four reasons. First, the versions of an obfuscated analytics library may be different from versions of its corresponding un-obfuscated analytics library. The same tracking API in different versions of the same analytics library may have different method call graphs. For a given analytics library, we extract its tracking APIs' method call graphs from the apps that do not obfuscate these tracking APIs' names. Since we only analyze 300 popular apps, the tracking APIs' method call graphs generated from the un-obfuscated analytics libraries may not cover all versions of the tracking APIs. We find not only the method call graphs of some obfuscated tracking APIs but also part of the readable strings in them are different from those of the un-obfuscated tracking APIs. Hence, these obfuscated tracking APIs belong to other versions of analytics libraries, because obfuscation does not change the readable strings in the code into different readable strings.

Second, while some apps use analytics libraries, they do not use the tracking APIs we care about. For example, app *Spotify Version 6.0* has used Google analytics library. However, it only uses this analytics library to track which campaigns and traffic sources are attracting users to download this app from the Google Play Store other than user's in-app actions. Third, several apps invoke the tracking APIs via Java reflection, which is not handled by our method. Fourth, some apps are protected by packing. These apps cannot be successfully decompiled by Apktool, and therefore, they cannot be analyzed by our method. The number of apps that have failed to analyze for each of these four reasons is shown in Table 4. We believe failure reason #1 can be mitigated by collecting and analyzing more apps that have not obfuscated the analytics libraries they used. In Section 3.2, the obfuscated apps that were used to test the obfuscated API finder were generated by ourselves. We have the corresponding un-obfuscated versions of the analytics libraries used by these apps. Hence, the obfuscated API finder is able to detect all the obfuscating apps.

Meanwhile, since dynamic running cannot cover all the code in an app, some apps do not generate the corresponding files in the file search process even they contain the tracking APIs. For such apps, we only consider their static analysis results in the following analysis. Finally, we select 100 popular apps from Chinese app market and 56 popular apps from Google play to analyze.

5 Experimental Results

We analyze the selected popular apps with our proposed system and then review the analysis results manually. Based on the information collected by the analytics libraries, we classify the apps into three levels: App level, Activity level and User level (See Table 5).

In Table 5, "App level" refers to the app that only uses analytics libraries to collect the information that reflects the running status of the whole app, such as what Activities are visited by the users. "Activity level" refers to the app that uses analytics libraries to collect the running status of each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

TABLE 4
The number of apps that failed to analyze

Analytics Library	Total Number[a]	Identified[b]	Failed			
			Reason One	Reason Two	Reason Three	Reason Four
Umeng	30	17	7	0	1	5
Talkingdata	5	2	0	0	0	3
Tencent Analytics	4	0	3	0	1	0
Baidu Analytics	3	0	1	0	0	2
Flurry	3	1	0	0	0	2
Adjust	4	3	0	1	0	0
Localytics	1	0	1	0	0	0
Google Analytics	13	5	5	2	0	1

[a] The number of apps that are found by the file search process but not found by the API search process.
[b] The number of apps that are identified the usage of obfuscated tracking APIs through the obfuscated API finder.

TABLE 5
The number of apps in each level of information collection

	Umeng	Talkingdata	Tencent Analytics	Baidu Analytics	Flurry	Adjust	Localytics	Google Analytics
App level	10	2	4	4	9	5	2	6
Activity level	46	9	7	15	15	10	3	13
User level	12	2	4	1	6	1	3	2

activity in an app, such as which "view" in the activity is pressed by the users. "User level" refers to the app that uses analytics libraries to collect the data generated by the users. For instance, how long time a user spends on a song in a music app. Table 5 shows most apps belong to the Activity level.

In order to detail the results we found in our analysis, we organize them as the answers to the following four questions.

Q1: Do analytics libraries leak users' personal information to app developers?

As the developers cannot get the raw data of the collected information, it is hard for them to profile individual users. However, developers can exploit these analytics libraries to collect users' private data directly.

For example, *Wo Mailbox Version 6.3.0* is a mailbox app that helps users manage their emails. It is developed by China Unicom and has more than 2.6 million active users in February 2016 [24]. Our tool found out that through the analytics library, this app automatically recorded users' e-mail addresses, recipients' email addresses, email addresses that users carbon copy emails to, emails' subjects and users' IP addresses.

We also found the analytics libraries did not check the information collected by the developers. They just performed some statistical analysis and presented the analysis results to the developers. This made it possible to collect users' sensitive information through these analytics libraries. In order to test and verify this vulnerability, we developed two apps with Umeng and Talkingdata [25], respectively. We disguise these two apps as Communication apps, so it is reasonable for these apps to request READ_CONTACTS permission. When users open their contacts book with our apps, these apps read users' contacts and display to them. But, these apps also secretly collect their contacts and send the contacts out through analytics libraries by invoking *MobclickAgent.onEvent(Context ctx, String eventId, Map eventValue)* for Umeng and *TCAgent.onEvent(Context ctx, String eventId, String eventValue)* for Talkingdata. The contacts are embedded in the parameter *eventValue*. Both Umeng and Talkingdata successfully collect users' contact information and present them to us, although the invoked tracking APIs were originally designed to collect user's in-app actions only. We tested these two apps with Flowdroid [26], a state-of-the-art static taint analysis tool for Android apps, but none of these two apps was reported to send user's contacts through the Internet. Although we have not found any real-world apps that have the similar behaviors, we think this is clearly a big security vulnerability.

Q2: Do analytics libraries leak users' personal information to analytics companies?

Since analytics companies own the raw data of the collected information, compared with the information leaked to the developers, information leaked to the analytics companies is much more serious.

For example, *com.culiukeji.huanletao Version 3.0* is a shopping app and *com.tadu.android Version 3.21* is a reading app. These two apps ask users to select their genders before using and collect users' gender information via Umeng. The developers' purpose is to understand whether their apps are popular among male or female. They can get the percentage of female users and percentage of male users from Umeng. However, Umeng gets each app user's gender information through this way. Since Umeng collects the user's device identifier (IMEI, MAC, etc) at the same time, it gets each corresponding device user's gender. *com.autohome.usedcar Version 5.2.1* is a used car trading app. It leaks the user's fine location to Umeng. Apps in Google play also have

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

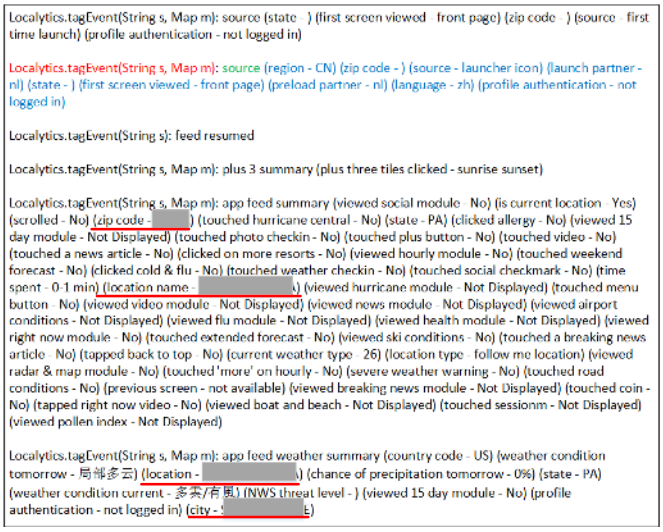

Fig. 9. Example of user's in-app actions collected by Localytics in The Weather Channel app.

similar behaviors. *Skype Version 6.15.0.1162* sends call ended time and message sent time to Flurry. *Text Free Version 5.6* sends user's fine location, rough number of user's contacts and rough length of every message to Flurry, and also sends the device's IMEI to Adjust. *The Weather Channel Version 6.0.0* leaks user's location to Localytics (as shown in Figure 9). Due to space limitation, we will not list all the apps that have the similar behaviors here. In Table 6, we summarize the privacy issues caused by the third-party analytics libraries in the apps being analyzed.

Besides, some analytics libraries collect users' data secretly. *Talkingdata Version 2.1.37* is a well-known analytics library in China. We find this analytics library reads the smartphone's sensors data without givinging any notice to users and even developers. When developers invoke the tracking APIs provided by this analytics library to collect users' in-app actions, this analytics library will read the sensors' data (includes ambient temperature sensor, relative humidity sensor, rotation vector sensor, pressure Sensor, light sensor and magnetic field sensor) and send the data to the analytics server. The collected sensor data will not be presented to the developers, and Talkingdata does not describe this behavior in its development documentation. Hence, neither the developers nor the users know about this behavior. This is not a direct privacy risk, but the data indeed can be used to infer users' surrounding environment. Differently, Talkingdata released a special version of its SDK for Google Play. In this version, it removed the code snippet for collecting sensor data. Lotuseed is another Chinese analytics library. This analytics library is not very popular, so we do not study it deeply. But in our preliminary work, we found this analytics library collected the list of apps installed in the user's phone secretly.

Q3: What will analytics companies know about the users if they link the information collected from different apps?

As we mentioned before, the privacy risk caused by analytics libraries is exacerbated if analytics companies link the data collected from different apps together to profile the users. Analytics companies can do this work easily because they collect device identifiers together with users' in-app actions. They know which apps are installed in the same device and used by the same user. The more popular the analytics library is, the more information it can gain. Take Umeng as an example, it is the most widely used analytics library in China. Apps that have integrated Umeng inside almost cover all the app categories (See Table 7). As these apps are popular apps, it is very possible that more than one of them is installed in the same phone.

TABLE 7
App categories that Umeng collects data from

Category	Number of apps	Category	Number of apps
Health & Fitness	1	Lifestyle	6
Photography	3	Tools	8
Weather	4	Music & Audio	3
Media & Video	11	News & Magazines	2
Entertainment	2	Books & Reference	4
Personalization	3	Finance	1
Travel & Local	2	Communication	5
Education	6	Shopping	5

We review the information that these apps' developers choose to collect through Umeng to see what user's personal information will be inferred by Umeng if a user installs these apps. First, Umeng is able to use all the Android permissions granted to these different host apps. For example, if an app used Umeng and required permission $\{p\_1, p\_2\}$ while another app on the same device also used Umeng and required permission $\{p\_3, p\_4\}$, then Umeng will be able to use permissions $\{p\_1, p\_2, p\_3, p\_4\}$ although it does not require these permissions by itself. Second, Umeng knows which apps integrating it are installed in the same phone. According to the previous study [5], [27], this app installation and usage pattern will leak the user's information. If the app is developed for a special user group, more information is leaked to Umeng. For example, *com.xtuone.android.syllabus* is developed for undergraduate students, *cn.haoyunbang* is developed for pregnant women and new mothers, etc. When these apps collect users' in-app actions through Umeng, Umeng also gets the users identity information. Third, data sent to Umeng often has clear semantics. Umeng can learn user's gender and reading habits from a reading app, user's location and approximate income level from a used car trading app, user's video watching habits from a video app, and user's health condition from a health app, etc. If Umeng analyzes and links this rich data, it can characterize app users in various aspects.

Q4: Do users know their in-app actions are collected by third-party analytics companies?

According to a previous study [28], 90% users cared if apps shared their personal data with third parties and 45% users believed the apps should never share their personal data with third parties without their explicit confirmation. This inspires us to see whether users know their in-app actions are collected by third-party analytics companies. Hence, we review these analytics libraries' privacy policies

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

TABLE 6
Summarize of the privacy issues caused by third-party analytics libraries in the apps being analyzed

Leaked information	Analytics library							
	Umeng	Talkingdata	Tencent Analytics	Baidu Analytics	Flurry	Adjust	Localytics	Google Analytics
Location	2	0	0	0	1	0	2	1
Gender	3	0	0	0	0	0	0	0
Ad content	1	0	0	1	0	0	0	0
IM account	0	0	2	0	0	1	0	0
Interested news	0	2	0	0	0	0	0	0
Favorite songs or videos	2	0	0	0	1	0	1	1
Favorite websites	3	0	0	0	0	0	0	0
Specific app function used duration	0	0	0	0	2	0	0	0

manually to discover what information they claimed they will collect. Table 8 describes what information is collected from users' Android devices declared by each analytics library in their privacy policies.

In these analytics libraries' privacy policies, we find that some analytics companies have listed what information they will collect and ask the developers to show the use of analytics libraries as well as the information collected by analytics libraries in their apps' privacy policies. However, after we review the privacy policies of the apps we select, we find only a handful of apps follow this rule. In the 100 apps from Chinese app market, only five apps clearly describe the using of third-party analytics service in their privacy policies, and only one of them gives the name of the analytics library it uses. In the 56 apps from Google Play, there are only 19 apps clearly describe the using of third-party analytics service and 5 apps in them give the name of the analytics library it uses. As collecting users' in-app actions do not need permissions, we believe that most users do not know their in-app actions are collected by third-party analytics libraries.

Discussion Our study shows the privacy risk from the analytics libraries. Given the results above, the discussion needs to focus on the reasons for these results and future privacy protection methods in this environment.

First, in today's Android phones, users' private information is not limited to the information protected by Android permissions. Due to the lack of a clear definition of what information is users' personal information, some developers cannot decide what information should not be collected. Second, some developers totally disregard user's privacy, as can be seen from their apps' privacy policies. Only a few apps describe the use of analytics libraries in their privacy policies. Third, some analytics companies do not specifically provide their privacy policies for mobile analytics, which makes mobile app developers hard to understand the privacy risk caused by these analytics libraries.

To protect users' privacy in this situation, we think the first step is to let the users know what information is leaked by the analytics libraries in each app. Then they can choose to use the app or choose a similar one. We believe the app market is the most important role to realize this goal. App markets can ask the developers to write a clear description about the using of analytics libraries and the information collected by analytics libraries in their apps' privacy policies. Alde can be used by app markets to explore the information collected by analytics libraries. We also design and implement an Android app called "ALManager" to help end users to manage the analytics libraries in their devices. ALManager will be introduced in Section 6.

Limitations In the static analysis process, Alde uses the methods provided by Apktool to decompile Android apps, so we cannot analyze the apps that cannot be decomiled by Apktool. Android inter-component communication and inter-process communication are not handled in the static analysis process, which may misses some results. In the dynamic analysis process, we cannot cover all the execution path. This is a common shortcoming of dynamic analysis. Although we take measures to identify the obfuscated tracking APIs' names, we cannot find the obfuscated tracking APIs in some apps.

6 Analytics Library Manager

As described in the previous section, today's analytics libraries in Android may leak users' private information. In this section, we design and implement an Android app called "ALManager" that aims to address this threat.

6.1 Design of ALManager

ALManager is designed for the Android platform. It aims to give a user the capability to control the information collected by analytics libraries. It achieves the following two goals: 1) it allows the users to examine the information collected by the analytics libraries. 2) it allows the users to specify the apps that can collect data through the analytics libraries and blocks the analytics libraries in other apps.

By default, ALManager blocks all of the user's in-app action data collected by the analytics libraries. But, it stores this intercepted information in a database. Users can search the database to find out what information is collected by which analytics library in which app. If a user confirms that the information collected by the analytics libraries in one app does not contain his/her personal information, he can remove this app from ALManager's app block list. Since then, the information collected by the analytics libraries in this app will be sent to the analytics servers as usual.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

TABLE 8
Information that these analytics libraries declare to collect in their privacy policy

Analytics library	Information collected
Umeng[a]	SDK versions, browsers version, ISP, IP, platform, time stamp, app identifier, app version, app distribution channels, device identifiers, MAC, IMEI, device type, terminal manufacturers, OS version, session start/stop time, language, location, time zone, network status, hard disk, CPU and battery usage, etc. Maybe also includes user's identifier of the app, longitude and latitude, gender, age, event triggered by the user, error, and page views.
Talkingdata[b]	SDK or API version, platform, time stamp, app identifier, app version, app distribution channels, Android advertiser ID, MAC, IMEI, device type, terminal manufacturers, OS version, session start/stop time, language, location, mobile network/Country code, time zone, network status, hard disk, CPU and battery usage, etc. Maybe also includes user's gender, age, geographic location, specific event triggered by user, error reporting and page views, etc.
Tencent Analytics	Do not find the privacy policy specific for mobile analytics service
Baidu Analytics	Do not find the privacy policy specific for mobile analytics service
Flurry[c]	device ID, IP address, time spent, links clicked, your location, apps on the device, or advertisements viewed on those apps.
Adjust[d]	Anonymized (hashed) IP address, anonymous identifiers such as Google Advertising ID or similar identifiers, installation and first opening of an app on your mobile device, user interactions within an app (e.g. in-app purchases, registration), information regarding which advertisements the user has seen or clicked on.
Localytics	Do not find the privacy policy specific for mobile analytics service
Google Analytics	Do not find the privacy policy specific for mobile analytics service

[a] https://www.umeng.com/policy.html
[b] http://www.talkingdata.com/privacy.jsp?languagetype=en
[c] https://policies.yahoo.com/xa/en/yahoo/privacy/topics/analytics/index.htm
[d] https://www.adjust.com/privacy-policy/

6.2 Implementation of ALManager

We implement our design on Android (as shown in Figure 10). ALManager is also based on the Xposed framework and it consists of four parts:

- **Xposed framework module**. The Xposed framework module is the core of ALManager. By hooking the tracking APIs in other apps with this Xposed framework module, ALManager intercepts the information collected by the third-party analytics libraries. The identification of a tracking API, including the API's class name, the API's method name and the API's parameter type is read from the configuration file. When a hooked tracking API is invoked, ALManager captures the following information: package name of the app that invokes this tracking API, time when this tracking API is invoked, values for this tracking API's parameters. This captured information is sent to ALManager's data storage service via Android Intent. Then, ALManager checks whether the app that invokes this tracking API is allowed to collect user's in-app action through third-party libraries. If this app is in ALManager's app block list, ALManager will replace the values of this tracking API's parameters with empty values. In this way, ALManager prevents user's personal information from being leaked to third-party analytics companies.
- **Data storage service**. The data storage service is an Android Service continuously running in the background. It receives the Intents sent from ALManager's Xposed framework module and stores the data in these Intents into a database.
- **Configuration file**. The configuration file stores ALManager's configuration, including a list of tracking APIs and a list of blocked apps. The list of tracking APIs can be automatically updated from the configuration server at each time ALManager starts. Users can also pull the latest tracking API list from the configuration server at any time through one click in ALManager. The configuration server is maintained by researchers. Therefore, updates of the tracking API list can be made timely when the tracking APIs of new third-party analytics libraries are added or the tracking APIs of old third-party analytics libraries are changed.
- **User Interface**. Users can check out the information collected by the third-party analytics libraries through ALManager's user interface. If a user believes that the information collected by the third-party analytics libraries in an app is not sensitive, he/she can remove this app from ALManager's app block list through the user interface.

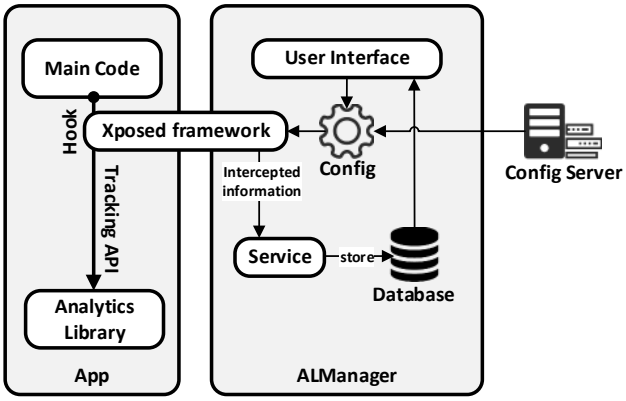

Fig. 10. The framework of ALManager.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

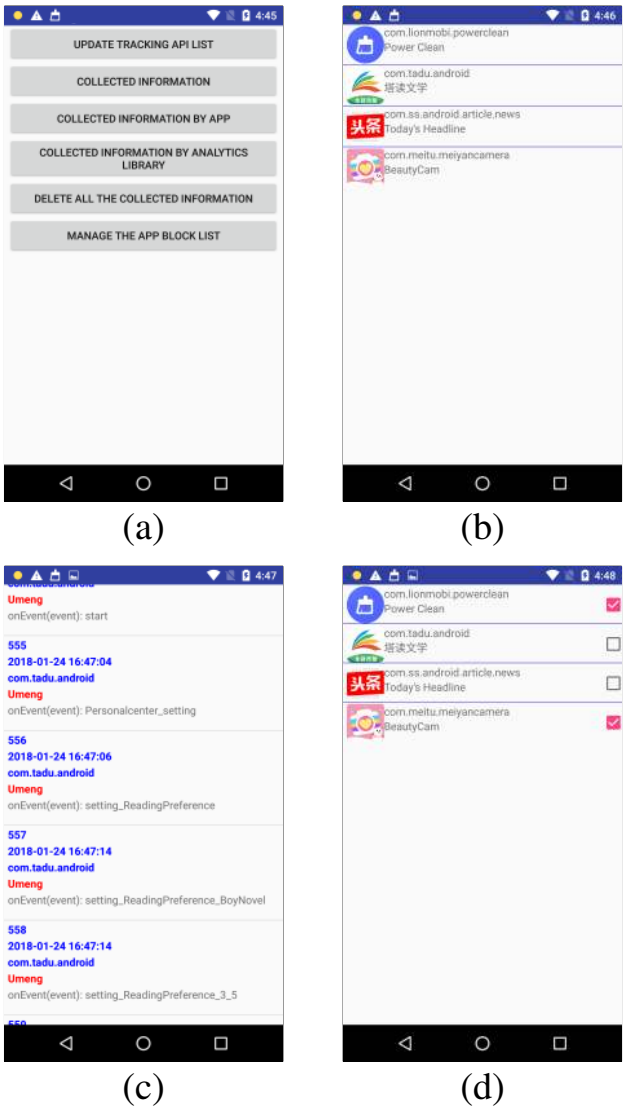

Fig. 11. A Demo of ALManager.

We implement ALManager and run it on a Google Nexus 5 smartphone. The Android version of this phone is 6.0.1 and the Xposed framework version is 86. Figure 11 (a) is the main activity of ALManager. Figure 11 (b) shows the apps that have collected user's in-app actions through third-party analytics libraries. By clicking on an app in the list, ALManager will display the information collected by the analytics libraries in this app (as shown in Figure 11 (c)). After the user checks out the information and thinks it is insensitive, he can choose to remove the app from ALManager's app block list. As shown in Figure 11 (d), some selected apps are removed from ALManager's app block list.

Limitations ALManager works only after successfully hooking the tracking APIs. Hence, it will fail due to the tracking APIs used by other apps are not listed in the configure file. ALManager identifies the tracking APIs that are needed to be hooked through the tracking APIs' identifiers (including class name, method name and parameter type) listed in the configure file. As we described in Section 4.2, some apps have obfuscated the tracking APIs they use, and these obfuscated tracking APIs may not be identified by our obfuscated API finder. Thereby, the identifiers of these obfuscated tracking APIs are not added in the configure file. This finally leads ALManager to fail to identify and hook the target tracking APIs.

6.3 Performance Overhead of ALmanager

ALManager incurs the biggest overhead when an app invokes the tracking APIs. Hence, we evaluate the runtime delay caused by ALManager when an app invokes a tracking API. We develop a test app to invoke a tracking API for 1000 times and record the time spent with and without ALManager, respectively. Results in Table 9 show that ALManager incurs about 2.1 ms overhead on each API invocation. Considering that the tracking APIs are invoked only when the user performs some specific operations, ALManager has very small impact on user experience. Through an Android app performance monitoring tool named *Emmagee*[5], we evaluate the impact of ALManager on the user experience in practice. Emmagee monitors the CPU time, the memory space used by the target app (not the whole system) when the app is running. Emmagee also monitors the target app's UI refresh rate. The app's UI refresh rate can reflect the performance smoothness of the app. The performance smoothness of the app will impact the users' experience. Experimental results on 16 apps from different categories indicate that ALManager has little impact on user experience. These 16 apps are selected from the apps that use Umeng library. As mentioned in Section 5, apps that have integrated Umeng inside cover 16 categories (as shown in Table 7). From each of these 16 categories, we choose one app to test the impact of ALManager on the user experience in practice. Considering a reading app named *Tadu literature* as an example, the app has little difference in CPU usage, memory usage and UI refresh rate with and without ALManager (as shown in Figure 12). As these 16 apps belong to 16 different categories and all of them were popular apps, we believe that the experiments conducted on these apps can generally represent most of apps in the app markets.

Low power consumption is necessary for mobile apps. Therefore, we further measure the power consumption overhead of ALManager. Note that the measured power consumption is not obtained with a power meter. It is given to show an estimate of the power overhead. We develop another test app to invoke a tracking API every second. We run this app for 30 minutes with and without ALManager. Other conditions, such as the screen brightness and the network connection, are kept the same. The experimental results show that the test app consumes 4% of the phone battery in 30 minutes no matter ALManager is used or not. This indicates the power consumption overhead of ALManager is negligible.

7 Related Work

We categorize the previous work into the following categories based on their main purposes.

5. https://github.com/NetEase/Emmagee

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

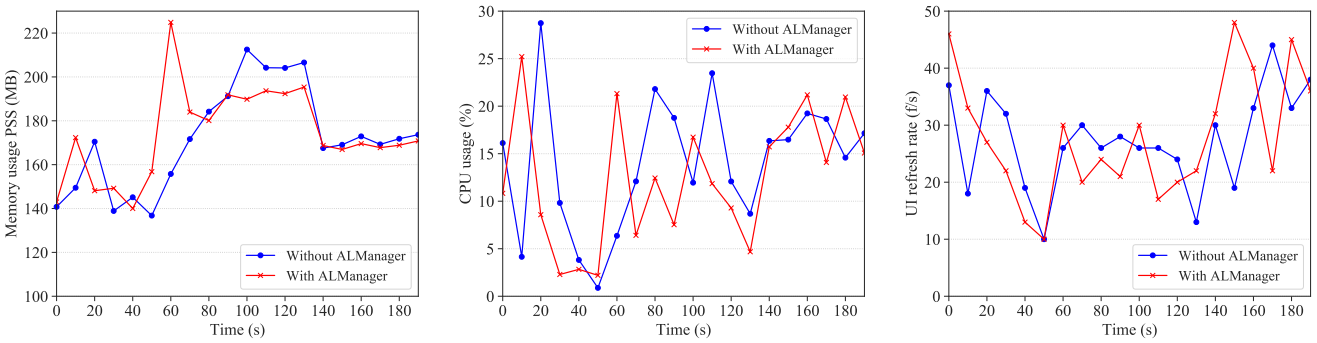

Fig. 12. Performance comparison of Android app *Tadu literature* with and without ALManager

TABLE 9
Performance on API invocations

Analytics library	1000 tracking API invocations		Overhead per API invocation
	Without ALManager	With ALManager	
Umeng	35 ms	1628 ms	1.6 ms
Talkingdata	38 ms	1099 ms	1.1 ms
Tencent Analytics	44 ms	2570 ms	2.5 ms
Baidu Analytics	82 ms	3151 ms	3.1 ms
Flurry	76 ms	2880 ms	2.8 ms
Adjust	70 ms	2096 ms	2.0 ms
Localytics	73 ms	2606 ms	2.5 ms
Google Analytics	177 ms	1608 ms	1.4 ms

Privacy and mobile advertising There are many studies focusing on the privacy issues associated with the advertising libraries in mobile apps. Grace *et al.* [4] studied potential privacy and security risks caused by in-app ad libraries. They analyzed 100,000 Android apps and found that most existing ad libraries collected private information. Book *et al.* [8] studied how app developers used the APIs through which a host app can send private information about the users to ad servers. They found that although most apps did not make use of these privacy-related APIs, the number of apps that used these APIs is not negligible. The information collected by these APIs can be simply identified by the APIs' names. They [29] also studied mobile ad targeting using simulated user profiles and found that a large portion of mobile ads are targeted based on app, location, time, and profiles built around actual users. Nath [9] studied what targeting information was sent to ad networks by mobile apps and how effectively the information was used by ad networks to target users. Demetriou *et al.* [5] developed a tool called "Pluto" that can be used to analyze apps and discover whether they leak targeted user data. They also studied what ad networks can learn from the list of apps installed in a phone. Meng *et al.* [30] studied what ad networks know about the user's interest and demographic information. They also studied whether the host apps could conversely use the targeted ads to infer some of the user information collected by the ad network. Taylor *et al.* [31] studied the privacy risk caused by *intra-library collusion* in Android environment. The *intra-library collusion* means that a single library embedded in more than one app on a device leverages the combined set of permissions available to it to pilfer sensitive user data. By analyzing historical data, they found that risks from intra-library collusion have significantly increased compared to two years ago.

Privacy and mobile analytics service Han *et al.* [28] studied how real-world users were tracked by the apps running on their Android smartphones. They employed dynamic information flow tracking to monitor when sensitive information was sent off the device. They recruited 20 volunteers to participate in this study. They found advertising and analytics were embedded in 57% of the apps and every participant in their study was tracked multiple times. However, they only studied the information protected by Android permissions. Chen *et al.* [32] studied the leakage of user's sensitive information through the vulnerabilities in mobile analytics services. They also studied how the ads served to users can be influenced by modifying the user profiles generated by these analytics services. Their experiments, conducted on Google Mobile Analytics and Flurry, validated the information leakage problem they described. Seneviratne *et al.* [33] studied the third-party trackers in popular paid apps. They categorized the trackers into three categories: advertising, analytics and utilities. They found 60% of the paid apps contained at least one tracker that collected personal information. Vallina *et al.* [34] analyzed mobile advertising and tracking ecosystem. They collected users' network traffic data through an app named "ICSI Haystack". Based on the collected data, they identified mobile advertising and tracking domains. Binns *et al.* [35] studied the distribution of third-party trackers on both websites and Android apps. They proposed a metric for measuring a tracking company's tracking capability. Based on the collected data and proposed method, they analyzed the concentration of third-party tracking and discussed the impact on user privacy. Binns *et al.* [36] also specifically studied the third-party tracking in the mobile ecosystem. They identified and analyzed the third-party trackers on 959,000 apps from the US and UK Google Play stores. The analysis results showed that third-party tracking was found in most apps and several tracking companies were dominating the market.

Analysis of third-party tracking scripts Third-party

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2903186, IEEE Transactions on Mobile Computing

tracking in the web environment has been widely studied. Pan *et al.* [37] proposed an anti-tracking browser called "TrackingFree". TrackingFree automatically generated different identifiers for different web sites. With these identifiers, third-party trackers could not link a user's requests sent from different web sites together to track the user. Wu *et al.* [38] proposed a system named "DMTrackerDetector", which automatically generated the blacklist of third-party trackers. Leveraging the different usage of JavaScript between trackers and non-trackers, DMTrackerDetector detected third-party trackers via supervised machine learning technique. Experimental results showed that 97.8% of the third-party trackers in the test dataset were detected correctly. Lerner *et al.* [39] developed a tool named "TrackingExcavator", which leveraged the Internet Archive's Wayback Machine to measure third-party tracking behaviors from 1996 to 2016. They found that third-party tracking were becoming more and more popular and complicated. Englehardt *et al.* [40] proposed an open-source web privacy measurement tool named "OpenWPM". With OpenWPM, they performed a large and detail measurement of online tracking. By analyzing data crawled from top 1 million websites, they found some fingerprinting techniques never measured before. Merzdovnik *et al.* [41] evaluated the effectiveness of many third-party tracker blockers and discussed the challenges of effectively blocking third-party trackers.

Privacy and mobile app's privacy policy Balebako *et al.* [42] studied how app developers made decisions about privacy and security. They interviewed 13 app developers to get information about privacy and security decision-making. And they tested what they found with 228 app developers online. One important thing they found was that although third-party ads and analytics services are pervasive, developers are not aware of the data collected by these tools. Yu *et al.* [43] developed a tool called "AutoPPG" that can be used to automatically construct correct and readable descriptions about the collection of user's private information. AutoPPG is able to generate the descriptions of third-party libraries used in apps; however, the information it focuses on is limited to the information protected by Android permissions. Yu *et al.* [44] also developed a tool called "PPChecker" that employed natural-language processing and static program analysis techniques to identify problems in Android app's privacy policy. They defined three kinds of problems in privacy policy: incomplete, incorrect and inconsistent. They analyzed 1,197 popular apps with PPChecker and found 282 apps had at least one kind of problems in privacy policy. Slavin *et al.* [45] proposed a semi-automated framework to detect privacy policy violations in Android apps. They constructed a policy terminology-API map that linked policy phrases to API functions. Then they used this map to find the APIs and perform an information flow analysis. They analyzed 501 top Android apps and discovered 63 potential privacy policy violations. Story *et al.* [46] analyzed the metadata of more than one million apps from Google Play to examine which apps had privacy policy. They worked out a logistic regression model to predict an app will have privacy policy or not. They found some categories of the apps, such as the apps had high rating and in-app purchases, were more likely to have privacy policy.

Android app analysis tools Arzt *et al.* [26] proposed "FlowDroid", a well-known static taint analysis tool for Android. FlowDroid conducted precise static taint analysis on Android apps and generated the propagation path for sensitive data. FlowDroid needed developers to predefine the source and sink APIs of the sensitive data. Holavanalli *et al.* [47] developed a static analysis tool named "Blue Seal", which could be used to analyze data sent inter-and intra-apps. Based on Blue Seal, they proposed an extension to Android permission mechanism called "Flow Permissions". Wei *et al.* [48] presented a static analysis framework called "Amandroid", which performed data flow and data dependence analysis at Android component level. Amandroid was implemented with Scala and could run distributed. Enck *et al.* [49] proposed a well-know dynamic analysis tool for Android named "TaintDroid". TaintDroid modified Android's virtualized execution environment for tracking taint data at an app's runtime. You *et al.* [50] presented "TaintMan", a dynamic taint analysis tool that supported Android RunTime (ART). By statically instrumenting the taint code into the target apps and the system libraries, TaintMan could run on un-rooted devices. One thing these tools had in common was that they all required the predefined taint sources to start the analysis. The sources were usually the system APIs used to read the taint data. However, in our work, the sinks are clearly the tracking APIs, but the sources are hard to define. Sources are no longer limited to system APIs that are used to read data protected by Android permissions from the system. Some readable strings in some apps may be sensitive to the users, like disease names in some health related apps.

In our previous work [51], [52], [53], [54], we extracted a big number of features from apps to detect their malicious behaviors. We also studied privacy issues in the Android single sign-on protocol [55] and embedded sensors [56]. Different from our previous work [1], we have additionally proposed a method call graph based "obfuscated API finder" to deal with the apps that have obfuscated the tracking APIs they use. The obfuscated API finder is used to identify the real names of the tracking APIs that are obfuscated by identifier renaming. With this obfuscated API finder, we have additionally analyzed 25 apps that we could not analyze before. The "Cydia substrate" is an API hook tool used in Alde's dynamic analysis process in our previous work. We have replaced it with the "Xposed framework". The Xposed framework is available on all Android versions while the Cydia substrate is only available on Android versions 2.3 through 4.3. In order to mitigate the privacy risk caused by the analytics libraries, we have developed an app named "ALManager" that leverages the Xposed framework to mitigate the privacy risk caused by analytics libraries. We have evaluated the runtime delay and the power consumption overhead caused by ALManager.

8 Conclusion

In this paper, we studied the information leakage caused by analytics libraries that collect users' in-app actions information. We developed a tool named "Alde" to explore the users' in-app actions. Through experiments on 8 popular analytics libraries and 300 apps downloaded from both Chinese app market and Google play, we found that some

apps leaked users personal information to analytics libraries without notifying users. We also found that popular analytics companies have the capability to characterize and profile users. To mitigate this kind of privacy risk, we developed an app named "ALManager" that leverages Xposed framework to manage analytics libraries. In the future work, we plan to improve our tool by making it more automated and more suitable for large-scale analysis. Then we will make it an online service to help users and app markets understand the information collected by analytics libraries.

Acknowledgments

The work reported in this paper was supported in part by National Key R&D Program of China, under grant 2017YFB0802805, in part by Natural Science Foundation of China, under Grant U1736114 and 61672092. Zhu's work was supported in part by the National Science Foundation (NSF) under grants CNS-1618684.

References

[1] X. Liu, S. Zhu, W. Wang, and J. Liu, "Alde: Privacy risk analysis of analytics libraries in the android ecosystem," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2016, pp. 655–672.

[2] appbrain, "Appbrain stats," http://www.appbrain.com/stats, 2018-01.

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

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

[5] S. Demetriou, W. Merrill, W. Yang, and et al., "Free for all! assessing user data exposure to advertising libraries on android," *NDSS 2016*, 2016.

[6] admob, "Admob by google," https://www.google.com/admob/, 2016.

[7] ——, "Admob quick start," https://developers.google.com/admob/android/quick-start?hl=en, 2016.

[8] T. Book and D. S. Wallach, "A case of collusion: A study of the interface between ad libraries and their apps," in *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*. ACM, 2013, pp. 79–86.

[9] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 59–73.

[10] flurry, "Custom events with flurry analytics for android," https://developer.yahoo.com/flurry/docs/analytics/gettingstarted/events/android/, 2016.

[11] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "Supor: Precise and scalable sensitive user input detection for android apps." in *USENIX Security Symposium*, 2015, pp. 977–992.

[12] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications." in *USENIX Security Symposium*, 2015, pp. 993–1008.

[13] JesusFreke, "Smali," https://github.com/JesusFreke/smali, 2016.

[14] rovo89, "Xposed framework," https://github.com/rovo89/Xposed, 2018-01.

[15] umeng, "Umeng," https://www.umeng.com/, 2016.

[16] B. Adler, "The key difference between mobile and web analytics metrics," http://info.localytics.com/blog/the-key-differences-between-mobile-and-web-analytics-metrics, 2014-3-20.

[17] B. Dykes, "Web analytics vs. mobile analytics: Whats the difference?" http://www.analyticshero.com/2013/07/24/web-analytics-vs-mobile-analytics-whats-the-difference/, 2013-7-24.

[18] apktool, "Apktool," http://ibotpeaches.github.io/Apktool/, 2016.

[19] guardsquare, "proguard," https://www.guardsquare.com/en/products/proguard, 2018-11.

[20] Google, "Dalvik bytecode," https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html, 2018.

[21] D. T. Milano, "Androidviewclient," https://github.com/dtmilano/AndroidViewClient, 2016.

[22] Android, "Monkey runner," https://developer.android.com/studio/test/monkeyrunner/index.html, 2016.

[23] mobyaffiliates, "The best app analytics tools list," http://www.mobyaffiliates.com/guides/best-app-analytics-tools-list/, 2015.

[24] eguan, "Eguan mobile apps top list," http://qianfan.analysys.cn/user-radar/view/ranking/topRanking.html, 2016-03.

[25] talkingdata, "Talkingdata," https://www.talkingdata.com/, 2016.

[26] S. Arzt, S. Rasthofer, C. Fritz, and et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[27] J. S. Atkinson, J. E. Mitchell, M. Rio, and G. Matich, "Your wifi is leaking: What do your mobile apps gossip about you?" *Future Generation Computer Systems*, vol. 80, pp. 546–557, 2018.

[28] S. Han, J. Jung, and D. Wetherall, "A study of third-party tracking by mobile apps in the wild," Technical report, University of Washington, Tech. Rep., 2012.

[29] T. Book and D. S. Wallach, "An empirical study of mobile ad targeting," *arXiv preprint arXiv:1502.06577*, 2015.

[30] W. Meng, R. Ding, S. P. Chung, and et al, "The price of free: Privacy leakage in personalized mobile in-app ads," in *NDSS Symposium 2016*, 2016.

[31] V. F. Taylor, A. R. Beresford, and I. Martinovic, "Intra-library collusion: A potential privacy nightmare on smartphones," *arXiv preprint arXiv:1708.03520*, 2017.

[32] T. Chen, I. Ullah, M. A. Kaafar, and et al., "Information leakage through mobile analytics services," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 2014, p. 15.

[33] S. Seneviratne, H. Kolamunna, and A. Seneviratne, "A measurement study of tracking in paid mobile applications," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 7.

[34] N. Vallina-Rodriguez, S. Sundaresan, A. Razaghpanah, R. Nithyanand, M. Allman, C. Kreibich, and P. Gill, "Tracking the trackers: Towards understanding the mobile advertising and tracking ecosystem," *arXiv preprint arXiv:1609.07190*, 2016.

[35] R. Binns, J. Zhao, M. Van Kleek, and N. Shadbolt, "Measuring third party tracker power across web and mobile," *arXiv preprint arXiv:1802.02507*, 2018.

[36] R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," *arXiv preprint arXiv:1804.03603*, 2018.

[37] X. Pan, Y. Cao, and Y. Chen, "I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2015.

[38] Q. Wu, Q. Liu, Y. Zhang, P. Liu, and G. Wen, "A machine learning approach for detecting third-party trackers on the web," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 238–258.

[39] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016." in *USENIX Security Symposium*, 2016.

[40] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1388–1401.

[41] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A large-scale study of tracker-blocking tools," in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 2017, pp. 319–333.

[42] R. Balebako, A. Marsh, J. Lin, and et al.h, "The privacy and security behaviors of smartphone app developers," 2014.

[43] L. Yu, T. Zhang, X. Luo, and et al., "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 39–50.

[44] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 2016, pp. 538–549.
[45] R. Slavin, X. Wang, M. B. Hosseini, and et al., "Toward a framework for detecting privacy policy violations in android application code," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 25–36.
[46] P. Story, S. Zimmeck, and N. Sadeh, "Which apps have privacy policies?" 2018.
[47] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 652–657.
[48] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, p. 14, 2018.
[49] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.
[50] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: an art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices," *IEEE Transactions on Dependable and Secure Computing*, no. 1, pp. 1–1, 2017.
[51] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang, "Droidensemble: Detecting android malicious applications with ensemble of string and structural static features," *IEEE Access*, vol. 6, pp. 31 798–31 807, 2018.
[52] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting android malicious apps and categorizing benign apps with ensemble of classifiers," *Future generation computer systems*, vol. 78, pp. 987–994, 2018.
[53] X. Wang, W. Wang, Y. He, J. Liu, Z. Han, and X. Zhang, "Characterizing android apps behavior for effective detection of malapps at large scale," *Future generation computer systems*, vol. 75, pp. 30–45, 2017.
[54] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.
[55] X. Liu, J. Liu, W. Wang, and S. Zhu, "Android single sign-on security: Issues, taxonomy and directions," *Future Generation Computer Systems*, vol. 89, pp. 402–420, 2018.
[56] X. Liu, J. Liu, W. Wang, Y. He, and X. Zhang, "Discovering and understanding android sensor usage behaviors with data flow analysis," *World Wide Web*, vol. 21, no. 1, pp. 105–126, 2018.

Xing Liu is a Ph.D. student in School of Computer and Information Technology, Beijing Jiaotong University, China. He received his B.S. degree from Beijing Jiaotong University in 2012. He visited the department of Computer Science and Engineering, The Pennsylvania State University, during October 2015-October 2016. His main research interests lie in mobile security and privacy.

Jiqiang Liu received his B.S. (1994) and Ph.D. (1999) degree from Beijing Normal University. He is currently a Professor at the School of Computer and Information Technology, Beijing Jiaotong University. He has published over 70 scientific papers in various journals and international conferences. His main research interests are trusted computing, cryptographic protocols, privacy preserving and network security.

Sencun Zhu received the B.S. degree in precision instruments from Tsinghua University, Beijing, China, in 1996, the M.S. degree in signal processing from the University of Science and Technology of China, Graduate School at Beijing, in 1999, and the Ph.D. degree in information technology from George Mason University, Fairfax, VA, USA, in 2004. He is an Associate Professor with Penn State University. His research interests include wireless and mobile security, network and systems security and software security. Among his many academic services, he is the editor-in-chief of EAI Transactions on Security and Safety and an associate editor of IEEE TMC.

Wei Wang is currently a full professor in the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, China. He earned his Ph.D. degree in control science and engineering under the supervision of Prof. Xiaohong Guan from Xi'an Jiaotong University, in 2006. He was a postdoctoral researcher in University of Trento, Italy, during 2005-2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, during 2007-2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, during 2009-2011. He visited INRIA, ETH, NTNU, CNR, and New York University Polytechnic. He is an editorial board member for Computer & Security and a young AE of Frontiers of Computer Science Journal. He has authored or co-authored over 80 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer and network security.

Xiangliang Zhang is currently associate professor of Computer Science and directs the Machine Intelligence and kNowledge Engineering (MINE)(https://mine.kaust.edu.sa) Laboratory in the Division of Computer, Electrical and Mathematical Sciences & Engineering, King Abdullah University of Science and Technology (KAUST). Prior to this, she was an assistant professor from August 2011 to June 2017 and was a research scientist at KAUST from September 2010 to August 2011. She earned her Ph.D. degree in computer science with great honors from INRIA-University Paris-Sud 11, France, in July 2010. She visited IBM T. J. Watson Research Center, Texas A&M University, University Paris-Sud 11, Concordia University, Microsoft Research Asia, and the University of Luxembourg. She has authored or co-authored over 100 refereed papers in various journals and conferences. She is an Associate Editor of Information Sciences. Her main research interests and experiences are in diverse areas of machine intelligence and knowledge engineering, such as complex system modeling, big data processing, and data security.