Detecting Improper Behaviors of Stubbornly Requesting Permissions in Android Applications

Jianmeng Huang, Wenchao Huang, Fuyou Miao, and Yan Xiong
(Corresponding author: Wenchao Huang)

School of Computer Science and Technology, University of Science and Technology of China
Elec-3 (Diansan) Building, West Campus of USTC, Huang Shan Road, Hefei, Anhui Province, China
(Email: huangwc@ustc.edu.cn)

(Received June 13, 2018; Revised and Accepted Nov. 22, 2018; First Online July 16, 2019)

Abstract

Android applications may stubbornly request permissions at initialization: if the user does not grant the requested permissions, these applications would simply exit, refusing to provide any functionalities. As a result, users are urged by this behavior to grant sensitive permissions and users actually lose the power to control their sensitive data, which may cause permission abuse and privacy leakage. In this paper, we propose an approach to automatically detect the improper behaviors of stubbornly requesting permissions. Experiments on real-world applications demonstrate the effectiveness of our approach and reveal that almost 24% analyzed applications contain stubborn permission requests.

Keywords: Android Security; Improper Behaviors Detection; Privacy Leakage; Stubborn Permission Requests

1 Introduction

The dramatic growth of Android applications (*apps* for short) has raised significant security concerns. Android dominated the smart phone market with a share of 85% in 2017 [14]. Most of the apps provide functionalities relying on sensitive user data (such as SMS and contacts), as well as certain system features (such as camera and microphone). However, a number of malicious apps abuse their privileges on private data [9], which threatens users' privacy.

To protect users' privacy, the Android permission mechanism [10] provides control on whether an app is allowed to access certain sensitive resources. On Android 6.0 and higher versions, if an app wants to access sensitive data, it should request corresponding permissions at runtime [7]. The permission mechanism does restrict the improper behaviors to a certain extent. For example, if a user does not need the location services from an app or does not trust the app, the user can deny the permission of accessing GPS to this app. Overall, the newer Android permission mechanism helps users to protect certain sensitive data by granting or denying corresponding permissions at runtime, allowing that only the functionalities related to the denied permissions are restricted.

However, some apps may get their requested permissions by urging users to grant them: if users do not grant the permissions, these apps would exit. Two reasons may account for the purposes of this behavior. First, these apps intentionally collect user's sensitive data. They gain profit from user's privacy, so they need to get corresponding permissions. Second, to reduce workload, the developers of these apps have not implement the codes which handle the exception of not having sensitive permissions. Hence, these apps are stubborn to get the permissions, otherwise the apps may crash. In this paper, we call this behavior as the stubborn permission request, which actually puts pressure on users to grant the requested permissions.

Unfortunately, users are vulnerable to the stubborn permission requests. On the one hand, users are not aware of the stubborn permission request in the app before the app is installed and running. After the app is installed, users may bother to uninstall it or choose another app with similar functionalities. On the other hand, users may be attracted by some functionalities of the app, which makes users ignore the risk of privacy leakage and hence grant the permissions. A survey [11] of 308 Android users and a laboratory study of 25 Android users found that only 17 percent paid attention to permissions. If users yield to the stubborn requests, their privacy is under threat for that the app would get full access to its required sensitive data. At runtime, users are not aware of when and which permissions are used. Therefore, it is necessary to inform users about the stubborn permission requests of an app before the app is installed into the device.

In this paper, we present an approach based on static analysis to detect improper behaviors of stubbornly requesting permissions in Android apps. We first study and model the behaviors of stubborn permission requests: if the requested permissions are not granted, the app would

exit. Then we statically identify the behaviors in the decompiled codes. To the best of our knowledge, our approach is the first to detect stubborn permission requests in Android apps. The experimental result shows that 24% of the tested popular apps contain stubborn permission requests. Such result indicates that users who are using these apps are exposed to the potential risk of privacy leakage, and the app market has not noticed the risk of such improper behavior.

Our approach can be adopted by Android app markets. The app market could add a label to the app which contains stubborn permission requests. As a result, users could be warned by the label. If a user does not trust the app, he/she could choose to install other alternative apps. With the result of our detection, the market could also conduct further detailed analysis on the apps in order to figure out how the sensitive data are used.

The main contributions of this paper are as follows:

- We study the stubborn permission requests in Android apps. To the best of our knowledge, our work is the first to investigate this kind of behaviors.
- We propose a static analysis approach to detect stubborn permission requests in Android apps.
- We demonstrate the effectiveness of our approach and present our findings.

The rest of this paper is organized as follows: Section 2 describes the background and the current problem. Section 3 presents the design of our approach and Section 4 describes experimental results. Section 5 describes related work and Section 6 concludes the paper.

2 Background and Problem Statement

2.1 Android Permissions

The main purpose of Android permission mechanism is to protect Android end users' privacy. In the Android system, every app runs in a limited-access sandbox. If an app needs to use resources or information outside of its sandbox, the app has to request the appropriate permissions (*e.g.*, contacts, Bluetooth and location). Before Android 6.0, the permissions are declared in the app manifest file and granted by users at install time. This permission mechanism does protect users' privacy to a certain extent. However, the permission model cannot be easily deployed, for that before appropriately using the Android permission model against suspicious apps, users should understand the program behaviors of the apps.

On Android 6.0 and higher versions, the system grants the permission automatically or might prompt the user to approve the request, depending on the level of the permission. Particularly, the Android permissions are categorized into two levels: the normal and dangerous permissions [8]. Some permissions are considered "normal" (*e.g.*, Internet, setting alarm and NFC) so the system immediately grants them upon installation. Other permissions are considered "dangerous" (*e.g.*, SMS, camera and storage) so that apps must explicitly request for users' agreements at runtime. Only dangerous permissions require users' agreements. The proper way of utilizing the dangerous permissions is:

1) Adding the permissions to the manifest;
2) If an app needs a dangerous permission, it must check whether it has that permission every time it performs an operation that requires this permission, because the user can revoke the permission from the app at any time;
3) If the app does not have the permission, the app must prompt the user for that permission using certain APIs provided by Android, which brings up a standard Android dialog that cannot be customized by developers.

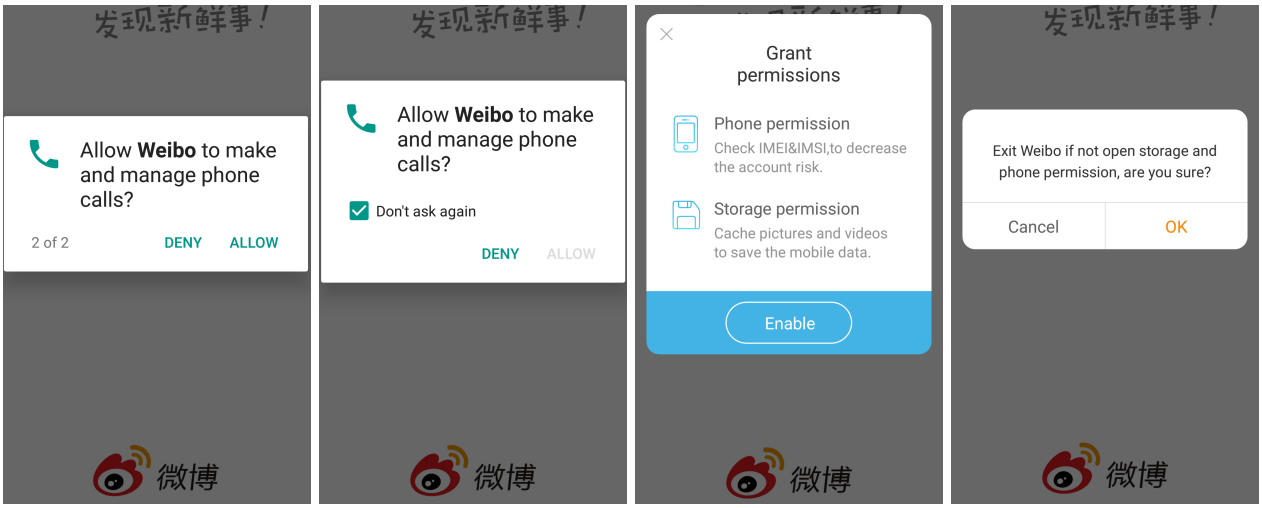
Besides, the permission request should occur at the time that the operation needs the corresponding sensitive resource, so that the user could understand why the app needs the permission in that circumstance and grants the permission to the app if the request is reasonable. After the user responds to an app's permission request, the system invokes method `onRequestPermissionsResult()` in the app, passing the user's response to the app. The app should override this method to find out whether the permission is granted.

The new permission mechanism further organizes permissions into groups related to a device's capabilities or features. Under this mechanism, permission requests are handled at the group level and a single permission group corresponds to several permission declarations in the app manifest. For example, the SMS group includes both the `READ_SMS` and the `RECEIVE_SMS` declarations. If an app gets the permission of `SMS` group, it has both the two permissions in this group.

2.2 Problem Statement

The permission mechanism on Android 6.0 and higher versions helps users better understand the behaviors of using sensitive permissions in an app and hence is more effective in protecting users' privacy. However, proper deployment of the permission mechanism requires higher workload of developers, since the app should always check for and request permissions at runtime to guard against runtime errors. Even if the user grants an app dangerous permissions, the app cannot always rely on having them. Because the user has the option to disable permissions in system settings.

Some apps urge users to grant the requested permissions when these apps begin to run. If the app has the dangerous permissions, the app would continue to run; Otherwise, the app would prompt the user for the permissions. If the user refuses to grant the permissions,



(a) The first permission request. (b) The second permission request. (c) The explanation of permission usage. (d) The final notification.

Figure 1: The permission request of an stubborn app.

the app would immediately exit. Figure 1 illustrates a real-world app which would exit if it does not get the required permissions. Figure 1a and Figure 1b show the standard Android dialogs which are used for requesting permissions. If the user denies the request twice, and chooses "Don't ask again", the dialog would not show at following runs of the app. However, this app prompts its own dialog (Figure 1c) which tells the users how to grant the required permissions in system settings. If the user denies all these requests, this app would inform the user that it would exit if not granted the permissions (Figure 1d).

The permission requests at the initialization of an app divorce from the original intention of the new Android permission mechanism and stubborn permission requests (*i.e.*, if the user does not grant the permissions, the app would exit) hurt the user's rights of utilizing functionalities which do not require dangerous permissions. It is reasonable for the app to disable the functionality that relies on the required permission (not granted), but it is not reasonable for the app to disable all the functionalities if it does not get all of the requested permission.

Challenge: The challenge of automatically detecting stubborn permission requests in Android apps is how to precisely and concisely model this kind of improper behaviors. To the best of our knowledge, the behaviors of stubbornly requesting permissions have not been investigated by existing researches.

Hence, adequate investigations from Android apps are required. Then, to automatically detect such improper behaviors in Android apps, a precise and concise model should be proposed. First, the model should be precise so that the detection result could achieve high precision. Second, the model should be concise so that the detection could be efficient.

3 Design and Implement

To detect the stubborn permission requests in Android applications, we propose a static analysis approach to find situations that if not granted required permissions, the app exits. This section describes our design and implement.

3.1 Overview

Our insight of detecting the improper of stubbornly requesting permission is based on the observation that all such kind of improper behaviors share the same subprocess that if the requesting result is "not granted", the app would finally finish all the activities. Hence, our detection is implemented by searching such patterns in the app.

Definition 1. *A **stubborn permission request** is an improper app behavior that if the user does not grant an app requested dangerous permissions, the app would not provide any functionality, including functionalities which do not need the requested permissions. Besides, when the app is started next time, the app would check whether it has certain dangerous permissions. If not, the app would request for permissions again.*

Definition 1 describes the improper behavior of stubbornly requesting permissions researched by this paper. We argue that the permission requesting prompt should provide users a choice of whether to share privacy with the app, rather than urging users to grant the permissions.

Based on the above definition, our detection for stubborn permission requests includes two steps. As the stubborn permission requests begin with checking permissions, our first step locates code fragments of checking whether an app has dangerous permissions, which is used as the entry points of our further analyzing. The second step figures out whether the code fragments of entry

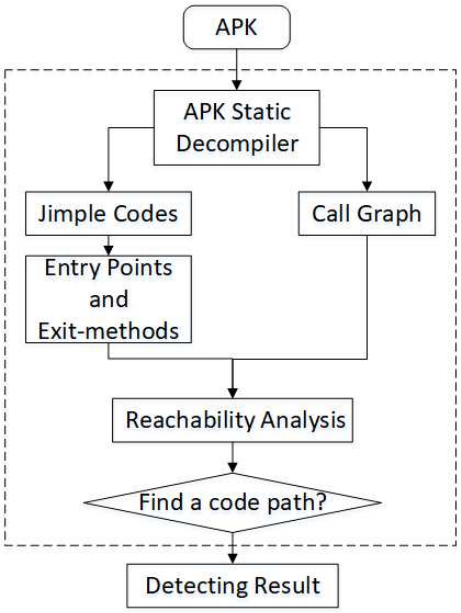


Figure 2: Framework of our approach

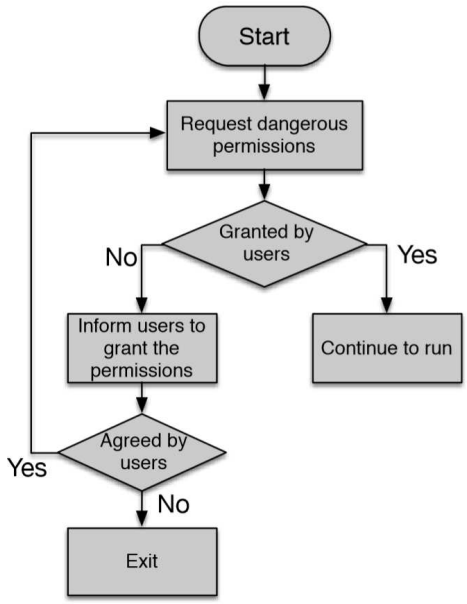


Figure 3: Flow of a stubborn permission request

points would finally lead to the exit of the app. If there exist code paths between the code fragments of checking dangerous permissions and finishing activities, which means that in certain conditions (*e.g.*, the user denies the permission requests), the app would exit after checking whether it has the dangerous permissions, and the stubborn permission requesting behavior is found.

3.2 Design

In this subsection, we first introduce how we choose the entry points for our further analyzing. Then we describe how we figure out whether the entry points would lead to the exit of an app. Figure 2 shows the methodology of our approach. Particularly, we use Soot [18] as the underlying analysis infrastructure. Soot translates the bytecode of an app to Jimple representation, a statement based intermediate representation, and it can generate an accurate call graph of the app. A call graph is a control flow graph which represents calling relationships between methods in an app. Then our approach locates the entry points and exit-methods (APIs which cause apps to exit). Finally, we apply a reachability analysis to figure out whether there exists a code path between the entry points and exit-methods. If found, we report the improper behavior of stubborn permission requests. Similar to prior static approaches [1,20], our analysis is based on the Jimple representation and the call graph. We do not adopt dynamic analysis approach because dynamic analysis faces the problem of low testing coverage, which causes insufficient analysis of the app.

3.2.1 Entry Points and Exit-methods

As our approach focuses on detecting stubborn permission requests, we first figure out the processes of stubborn permission requests. Figure 3 shows the normal flow of a stubborn permission request. The app asks for dangerous permissions first, then the user makes a choice of denying it or allowing it. Afterwards, the app could get the requesting result from method `onRequestPermissionsResult()`. If the requested permission is not granted, the app would inform the user to grant the permission by a prompted dialog or through the system setting. Finally, if the user still refuses to grant the requested dangerous permissions, the app would exit. Overall, whether the app is going to exit depends on the results of method `onRequestPermissionsResult()`.

Studying the processes of stubborn permission requests, we find that method `onRequestPermissionsResult()` could be considered as the beginning of analyzing stubborn behaviors. Besides, we find that some app do not get the response of the permission request in the recommended method `onRequestPermissionsResult()`. There are other ways of indicating whether users grant the requested permission or not. First, an app could call `checkSelfPermission()` to check permissions and perform stubborn permission request if it is not granted. Hence, method `checkSelfPermission()` and `checkCallingorSelfPermission()` are also considered as entry points by our approach. Second, an app could indicate whether it has certain permissions. For example, an app could invoke an Android API (*e.g.*, `getLongitude()`) which requires a dangerous permission and utilize Java exception handling to infer whether the app has the permission. Table 1 lists some sensitive APIs and their required corresponding permissions. Hence,

Table 1: Sensitive APIs and the corresponding permissions

Method	Permission
com.android.internal.telephony.cdma.CDMAPhone: java.lang.String getDeviceId()	READ_PHONE_STATE
android.location.Location:double getLongitude()	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
com.android.nfc.NfcService:void onSeApduReceived(byte[])	NFC
com.android.server.sip.SipService:void open(android.net.sip.SipProfile)	USE_SIP
android.media.AudioManager: void setRingerMode(int)	RECORD_AUDIO
android.telephony.SmsManager:void sendTextMessage(...)	SEND_SMS

the try-catch blocks which invoke sensitive APIs are also considered as entry points.

Our approach locates all the entry point in an app, by searching the method names of entry point in the decompiled Jimple codes. Particularly, method `onRequestPermissionsResult()` is a callback method which would be invoked by Android system and it is overridden by the app in order to handle the permission request response. As a result, this method is found by retrieving all the third party classes of the app. For the try-catch blocks, we search all the try blocks in the Jimple codes for sensitive APIs (list from PScout [2]). If found, we record the corresponding catch blocks as entry points for further analyzing.

Particularly, we denote *exit-methods* as APIs that would finish the activities or cause the app to exit. Table 2 lists the exit-methods monitored by our analysis. Some of the methods directly finish the activities or terminate the app process. For example, method `finish()` and `finishActivity()` finish an Android activity. The `System.exit()` method quits the current program by terminating the running Java virtual machine. Method `killBackgroundProcesses()` kills all background processes associated with the given package. Method `killProcess()` kills the process with the given PID. `finishAndRemoveTask()` finishes all activities in this task and removes it from the recent tasks list. Another way of stopping users from utilizing the functionalities of an app is to hide the activities of the app to background. `moveTaskToBack()` moves the task containing this activity to the back of the activity stack.

Table 2: The exit-methods

Class	API
android.app.Activity	finish
android.support.v4.app.ActivityCompat	finishAffinity
java.lang.System	exit
android.app.ActivityManager	killBackgroundProcesses
android.os.Process	killProcess
android.app.Activity	moveTaskToBack
android.app.ActivityManager.AppTask	finishAndRemoveTask

Algorithm 1 Identifying stubborn permission requests

1: **Input:** the entry point methods *entryPoints*, and the APK file *app.apk*
2: **Output:** whether the app has the stubborn permission request behavior
3: Begin
4: *callgraph* ←generateCHACallgraphWithSoot(*app.apk*)
5: **for all** $ep \in entryPoints$ **do**
6: **if** reachabilityAnalysis($ep, callgraph$) **then**
7: **return true**
8: **end if**
9: **end for**
10: **return false**
11: End

3.2.2 Finding Stubborn Permission Requests

To figure out whether the entry points would lead to the exit of an app under certain circumstance, we traverse the call graphs which are rooted from the entry points. Here, we introduce our reachability analysis, which finds whether an entry point method would directly or indirectly invoke methods that finish activities or cause the app to exit.

Our reachability analysis is implemented by finding method invocation chains from the entry points to exit-methods in an app. The method invocation chain indicates that checking the permission request response would finally lead to the exit of an app. In particular, the invocation chain is a path in the call graph which starts with an entry point and ends with an exit-method. Hence, if there is a method invocation chain from an entry point method to an exit-method, we report that the app contains stubborn permission requests.

Algorithm 1 shows how our approach finds stubborn permission requests. It accepts the entry points and the apk file as inputs. The output of this algorithm is a judgment about whether this app contains stubborn permission requests. Generally, Algorithm 1 recursively searches the methods that could be triggered from the entry point

Algorithm 2 reachabilityAnalysis

```
1: Input: a method ep, a call graph callgraph, the exitMethods
2: Output: whether method ep has the stubborn permission request behavior
3: Begin
4: subgraph ← callgraph.rootedWith(ep)
5: children ← subgraph.getChildren(ep)
6: for all child ∈ children do
7:   if child ∈ third party packages then
8:     if reachabilityAnalysis(child, subgraph) then
9:       return true
10:    end if
11:  else
12:    if child ∈ exitMethods then
13:      return true
14:    end if
15:  end if
16: end for
17: return false
18: End
```

methods.

In detail, Algorithm 1 first decompiles the apk file into Jimple codes, then it utilizes Soot to generate a call graph using the class hierarchy analysis (CHA) (*i.e.*, line 4), which conservatively estimates possible receivers of dynamically-dispatched messages. As a result, for example, a *virtual method* could be invoked, which is due to the use of polymorphism through which it is possible for a subtype to override methods defined in its super-types. The actual target of the virtual call is determined at run time. The generated call graph would list the possible target method in it.

Then we apply the reachability analysis (Algorithm 2) for each entry point (*i.e.*, line 5-9 in Algorithm 1). In the reachability analysis, Algorithm 2 first gets a subgraph rooted with the method *ep* from the the call graph of the app. Then, it travels the subgraph to check each method. If a method in the call graph belongs to the third party libraries (*i.e.*, defined by the app), this algorithm would recursively search for exit-methods inside the method (*i.e.*, line 6 to line 16 in Algorithm 2). If one of the exit-methods is found in the sub-callgraphs rooted with an entry point method, the algorithm would report a recognition of stubborn permission request.

4 Evaluation

4.1 Experimental Setup

All experiments were conducted on a 4-processor 16GB-RAM machine, and all the apps analyzed in this section were collected from four third party Android app markets in China (*i.e.*, wandoujia, anzhi, baidu-shouji, tecentyingyongbao) and Google Play. The collected apps were among the top popular apps in each category sorted by the app markets.

4.2 Real-world Apps Study

We first analyzed 104 apps downloaded from the four third party Android app markets to figure out how common is the stubborn permission request in real-world apps of China. The selected apps were the most popular apps (sorted by the downloads) of all the apps in the market, and all of them were listed by the four app markets. Note that some apps may have different release versions of apps targeted for different kinds of mobile devices. Besides, some of the apps may have customized versions cooperated with the app market for the purpose of advertising, but main functionalities of these apps are the same. In our experiments, we found that different versions of an app have the same behaviors of permission requesting. Hence, we treated the apps with same name as the same app. As a result, we found that 25 (*i.e.*, 24%) tested China apps contained stubborn permission request behaviors.

We also investigated corresponding apps from Google Play. Google Play serves as the official app store for the Android operating system. Only 37 out of the 104 Chinese apps are listed in Google Play. Other apps do not provide international services. Among the 37 apps, 4 (*i.e.*, 11%) apps contain stubborn permission requests. We can conclude that apps in Google Play may also contain stubborn permission requests, but apps in Google Play are with lower ratio of containing stubborn permission requests than apps in third party markets of China. Based on the above experimental results, we suggest that app markets pay more attention to stubborn permission requests in apps, especially third party Android app markets.

Table 3: Results of market apps study (P: precision, R: recall)

Category	analyzed	stubborn	detected	P	R
Movie &Music	19	2	2	100%	100%
News	20	7	7	100%	100%
Social	18	4	3	100%	75%
Tools	19	6	6	100%	100%
Sport	19	5	4	100%	80%
Shopping	20	7	6	100%	86%
Weather	20	6	6	100%	100%

Furthermore, in order to investigate stubborn permission requests in different categories of apps, we analyzed 140 apps from 7 different categories, each of which contains 20 apps, and each app was analyzed for maximum 30 minutes. As apps providing different functionalities require different permissions, the occurrence frequency of stubborn permission requests may be different. For example, apps which provide "map" function naturally requires `GPS` permission, and the stubborn permis-

sion requests are more likely to be found in these apps. Table 3 lists the results of our investigation. Of the 140 apps, 5 (3.5%) apps were not analyzable, which is caused by exceeding the RAM limit or the 30 minutes timeout, and Soot exceptions while transforming bytecode to Jimple representation.

The results show that stubborn permission requests occur differently in each app category. Particularly, the stubborn permission request occurs more among apps in the categories of *news*, *tools*, *shopping* and *weather* than others, which indicates that app providers in these categories are more likely to collect users' privacy. Apps in these categories provide functionalities frequently used by users, which increases the possibility of leaking privacy. As a result, we suggest app providers of these apps regulate the behaviors in the apps of requesting dangerous permissions and utilizing users' private data.

Table 4: Results of popular apps from different app markets

App Name	Wandoujia	Anzhi	Baidu-shouji	Tecent-yingy-ongbao	Google Play
UC browser	◯	◯	◯	◯	×
WeChat	◯	◯	◯	◯	◯
Taobao	◯	◯	◯	◯	◯
Zhihu	×	×	×	×	×
Meituan	◯	◯	◯	◯	◯
DiDi	◯	◯	◯	◯	◯
Ctrip	×	×	×	×	×
zhifubao	◯	◯	◯	◯	◯
Pinduoduo	×	×	×	×	–
QQ Music	◯	◯	◯	◯	–
Toutiao	×	×	×	×	–
Facebook	–	–	–	–	×
Reddit	–	–	–	–	×
Quora	×	×	×	×	×

◯ means that the app contains stubborn permission requests.
× means the app does not contain stubborn permission requests.
– means that the app is not listed on the corresponding app market.

Next, we present some detection results of popular apps from different app markets. Table 4 lists our detection results. The improper behavior of stubbornly permission requests is not rare in the most popular apps. We observe different results regarding to different app markets. For example, some apps from Chinese app markets contain stubborn permission requests, but the releases of them on Google Play does not contain stubborn permission requests (*e.g.*, *UC browser*). Some apps contain stubborn permission requests, and these apps are only listed on Chinese app markets (*e.g.*, *QQ Music*). We also observe different behaviors that do not contain stubborn permission requests. For example, *Zhihu* does not request permission when it is started to run, *Ctrip* requests permissions but it does not exit even if users refuse the request.

4.3 Precision and Recall

To get the precision and recall of our analysis results, we manually installed the tested apps and checked whether these apps have stubborn permission requests. For apps which require users to sign up the app before enjoying the functionalities, we created accounts to test these apps. We ran each app for 5 minutes, manually triggering all the UI elements of each app's activities, until we found a stubborn permission request. Finally, the ground truth of whether the tested apps have stubborn permission requests was collected.

We consider two evaluation metrics, the precision and recall.

Precision: The fraction of permission requests correctly identified as stubborn among those reported by our static approach.

Recall: The fraction of permission requests correctly identified as stubborn among those manually tested, *i.e.*, the ground truth.

Given the ground truth information and the detection results, there are four possible outcomes: True positive(TP), true negative (TN), false positive (FP) and false negative (FN). TP means that an app contains improper behavior of stubbornly requesting sensitive permissions with respect to the ground truth and it is detected by our approach. TN means that an app does not contain improper behavior of stubbornly requesting sensitive permissions with respect to the ground truth and our approach does not find stubborn permission requests in the app. FP means that an app does not contain improper behavior of stubbornly requesting sensitive permissions with respect to the ground truth but our approach reports that the app contains stubborn permission behaviors. FN means that an app contains improper behavior of stubbornly requesting sensitive permissions with respect to the ground truth but our approach does not find stubborn permission requests in the app. Finally, the precision and recall are computed by the following formulas:

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

The precisions and the recalls of the analysis results for apps from different categories are listed in Table 3 (the last two columns). The third column lists the manually analyzed results. For all the app categories, we got 100% detection precision, which means that all the reported apps in our detection results have stubborn permission requests. The recalls of our detection are also high, which indicates the effectiveness of our work. There exist apps which actually have stubborn permission requests, but our static analysis approach didn't recognize them. Two reasons account for this situation. First, there are other covert ways of checking whether an app has certain dangerous permission. For example, some malicious app may use Java reflection to obfuscate method calls. It

is difficult for static analysis based approaches to handle reflection [26]. In this situation, the method calls of entry point methods may be missed by our static approach for that our approach does not support handling reflection currently. Second, we find that there exist apps which do not exit after the permission requests are denied by users. Instead, these apps continuously request for permissions. Even if the standard dialog of requesting permissions would not be shown after users choose "Don't ask again", these app would present their own dialog (*e.g.*, as shown in Figure 1b and Figure 1c). Users could finish the app by clicking the default Android "home" button. We leave these as future work.

We believe that our approach is effective in detecting improper behaviors of stubbornly requesting permissions in Android applications. Although manual detection can achieve high precision and recall, it is not applicable for massive app audition. As our static analysis approach can achieve similar detection results compared with manual detection and it does not require human assist, we believe it can help app markets or analysts to automatically detect stubborn permission requests in Android apps.

4.4 Findings

During our analysis on the market apps, we have some findings. The `PHONE` and `STORAGE` are the two most frequently requested permissions and they are also the top permissions occurring in the stubborn permission requests. We studied the corresponding apps that stubbornly request the two permissions to figure out why these apps insist in getting the permissions.

We find that the some of the Android permission groups are not properly designed , which actually contributes to the concern about privacy leakage of stubborn permission requests. For example, the `PHONE` permission group contains `READ_PHONE_STATE`, `CALL_PHONE`, `READ_CALL_LOG`, `WRITE_CALL_LOG`, *etc.* Some apps need to read the IMEI (International Mobile Equipment Identity) of a device for identifying the unique phone. For instances, in some voting systems, it is required that each device only has one vote. Mobile app developers need to understand who are using their apps, and the IMEI is often used to distinguish different users [22]. Hence, these apps request for the corresponding permission: `READ_PHONE_STATE`. However, the system informs users by the permission group `PHONE`. Users may be worried about granting the app this group of permissions for that if the app requests for other permissions in this group latter, the system would directly grant it the permissions without informing users again. The permissions in the permissions group are organized by Android permissions mechanism with the same sensitive resource, but as the situation in this case, the `READ_PHONE_STATE` is more frequently used by apps than other permissions in its group. We believe that the frequency of the permissions used by apps should also be taken into consideration. Based on our study, we suggest that this permission could be taken out from the permission group `PHONE` in order to reduce the privacy concern.

We also find that different versions of an app may have different behaviors of permission requests. For example, *Weibo* and *Weibo international* are different versions of the client app of *Sina Weibo*. The *Weibo* is an interface for Chinese market, and the *Weibo international* aims to be adapted by other cultures. However, *Weibo* contains stubborn permission requests, while *Weibo international* does not have ones. Different markets may have different data protection rules, which may account for this situation that two apps of the same company have different behaviors of stubborn permission requests. We believe that users' privacy should be put it in the first place, and app providers should follow the same data protection rules to develop their apps.

We observe that all the stubborn permission requests occur at the initialization of an app. The purpose of the stubborn permission is to urge users to grant the requested permissions. Hence, stubborn permission requests at the initialization of an app put pressure on users: if they do not grant the permissions, they would not enjoy normal app functionalities. As a result, it increases the possibility for the app to get the requested permissions. Besides, we observe some apps, which contain stubborn permission requests, do not check whether they have the permissions at runtime in the codes. Hence, stubbornly requesting permissions at the initialization of an app guarantees that the app always has the requested permissions at runtime, which reduces the workload of the app developers. Based on these observations, we suggest the analyzers and users pay more attention to the apps which stubbornly request permissions at initialization.

5 Related Work

Prior work demonstrates that install-time prompts of requesting permissions fail to protect users' privacy because users do not comprehend these permission requests or pay attention to them [12, 17]. Users often do not understand which permission correspond to which functionalities in apps before they are familiar with the apps. As a result, users are prone to grant the permissions. Apex [23] and πbox [19] provide users with the ability to grant permissions to the app at runtime. This feature is now integrated in Android since the version of Marshmallow. In our work, the stubborn permission request occurs at the initialization of Android apps, which has the same problem that users are not familiar with apps. Moreover, stubbornly requesting permission urges users to grant the permissions.

Researches have designed systems to recommend permissions for app developers to properly request permissions [3, 16]. These researches are based on mining technique or collaborative filtering technique. Other researchers have developed systems to to predict permission decisions at runtime based on contextual information and

machine learning methods [24]. By requiring users to report privacy preferences, clustering algorithms have been used to define user privacy profiles even in the face of diverse preferences [21].

Wijesekera *et al.* [28] build a classifier to make privacy decisions on the user's behalf by detecting when context has changed and, when necessary, inferring privacy preferences based on the user's past decisions and behavior. It automatically grants appropriate resource requests without further user intervention, denies inappropriate requests, and only prompts the user when the system is uncertain of the user's preferences.

There is a large body of work researching the improper use of permissions in Android permissions. Wei *et al.* [27] find that some Android Applications do not follow the principle of least privilege, intentionally or unintentionally requesting permissions which are not related to the declared app functions. Fauzia *et al.* investigate the combined effects of permissions and intent filters to distinguish between the malware and benign apps [15]. Qian *et al.* [25] use static analysis to determine whether an app has potential risks, and then embed monitoring Smali code for sensitive APIs.

As a result, their approach could reveal the malicious behaviors of applications leaking users' private data. Zhao *et al.* [30] extract the API packages, risky API functions and permission information and then use convolutional neural network to identify Android malwares. Pegasus [6] focuses on detecting the malicious behavior that can be characterized by the temporal order in which an app uses APIs and permissions. It can automatically detect sensitive operations being performed without the user's consent. Our work detects a kind of improper behavior that stubbornly requests dangerous permissions.

To enhance the Android permission mechanism, MockDroid [4] allows users to mock an application's access to a resource. It offers users with binary options that either revoking access to particular resources or providing full access to the app. However, MockDroid only works for explicitly requested resources. To deal with innocuous sensors, IpShield [5] performs monitoring of every sensor accessed by an app and allows users to configure privacy rules which consist of binary privacy actions on individual sensors. Blue Seal [13] extends the Android permission mechanism with semantic information based on information flows, which allows users to examine and grant information flows within or across multiple applications. It can remove unnecessary permissions from over privileged apps and synthesize flow permissions for the app. FineDroid [29] associates each permission request with its application context and provides a fine-grained permission control. FineDroid also features a policy framework to flexibly regulate context-sensitive permission rules. SmarPer [24] relies on contextual information and machine learning methods to predict permission decisions at runtime.

6 Conclusion

This paper presents a static analysis approach which targets for detecting stubborn permission requests: if users do not grant the required dangerous permissions, the app would not provide any functionalities. This stubborn behavior threatens users' privacy for that if users yield to it, the app would get full access to sensitive data and the users are not aware of how the app would use the sensitive data. By statically analyzing the decompiled codes, we identify the stubborn permission requests. Our experimental results indicate that our approach is effective in detecting the stubborn requests. Our work could be utilized by app markets so that users can be informed if an app contains stubborn permission requests.

Acknowledgments

The research is supported by National Natural Science Foundation of China under Grant No.61572453, No.61202404, No.61520106007, No.61170233, No.61232018, No.61572454, Natural Science in Colleges and Universities in Anhui Province under Grant No.KJ2015A257, and Anhui Provincial Natural Science Foundation under Grant No.1508085SQF215. The authors gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 49, no. 6, pp. 259–269, 2014.

[2] K. W. Y. Au, Y. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 217–228, 2012.

[3] L. Bao, D. Lo, X. Xia, and S. Li, "What permissions should this android app request?," in *International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 36–41, 2016.

[4] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pp. 49–54, 2011.

[5] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "Ipshield: A framework for enforcing context-aware privacy," in *11th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI'14), pp. 143–156, 2014.

[6] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs," in *NDSS*, 2013. (https://www.cs.cornell.edu/~tmagrino/papers/ndss13-pegasus.pdf)

[7] Developer.android.com., *Requesting Permissions at Run Time*. (http://developer.android.com/training/permissions/requesting.html)

[8] Developer.android.com., *Permissions Overview*. (https://developer.android.com/guide/topics/permissions/overview.html#normal-dangerous)

[9] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 627–638, 2011.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the eighth symposium on usable privacy and security*, pp. 3, 2012.

[12] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 1025–1035, 2014.

[13] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," in *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*, pp. 652–657, 2013.

[14] IDC, *Smartphone OS Market Share*, 2018. (http://www.idc.com/promo/smartphone-market-share/os).

[15] F. Idrees and M. Rajarajan, "Investigating the android intents and permissions for malware detection," in *IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'14)*, pp. 354–358, 2014.

[16] M. Y. Karim, H. Kagdi, and M. Di Penta, "Mining android apps to recommend permissions," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, vol. 1, pp. 427–437, 2016.

[17] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A conundrum of permissions: Installing applications on an android smartphone," in *International Conference on Financial Cryptography and Data Security*, pp. 68–79, 2012.

[18] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: A retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS'11)*, vol. 15, pp. 35, 2011.

[19] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "πbox: A platform for privacy-preserving apps," in *NSDI*, pp. 501–514, 2013.

[20] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in android apps," in *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, 2015. ISBN: 978-1-4799-1934-5.

[21] J. Lin, B. Liu, N. Sadeh, and J. I. Hong, *Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings*, 2014. (https://www.usenix.org/system/files/conference/soups2014/soups14-paper-lin.pdf)

[22] W. Liu, Y. Zhang, Z. Li, and H. Duan, "What you see isn't always what you get: A measurement study of usage fraud on android apps," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 23–32, 2016.

[23] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pp. 328–332, 2010.

[24] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. Hubaux, "Smarper: Context-aware and automatic runtime-permissions for mobile devices," in *IEEE Symposium on Security and Privacy (SP'17)*, pp. 1058–1076, 2017.

[25] Q. Qian, J. Cai, M. Xie, and R. Zhang, "Malicious behavior analysis for android applications," *International Journal of Network Security*, vol. 18, no. 1, pp. 182–192, 2016.

[26] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS'16)*, 2016. (https://www.bodden.de/pubs/ssme16harvesting.pdf)

[27] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 31–40, 2012.

[28] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *IEEE Symposium on Security and Privacy (SP'17)*, pp. 1077–1093, 2017.

[29] Y. Zhang, M. Yang, G. Gu, and H. Chen, "Rethinking permission enforcement mechanism on mobile

systems," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 10, pp. 2227–2240, 2016.

[30] Y. Zhao and Q. Qian, "Android malware identification through visual exploration of disassembly files," *International Journal of Network Security*, vol. 20, no. 6, pp. 1061–1073, 2018.

Biography

Jianmeng Huang received the B.S. degree in computer science from University of Science and Technology of China in 2013. He is currently working towards the Ph.D. degree at the Department of Computer Science and Technology, University of Science and Technology of China. His current research interests include information security and mobile computing.

Wenchao Huang received the B.S. and Ph.D degrees in computer science from University of Science and Technology of China in 2006 and 2011, respectively. He is an associate professor in School of Computer Science and Technology, University of Science and Technology of China. His current research interests include information security, trusted computing, formal methods and mobile computing.

Fuyou Miao received his Ph.D of computer science from University of Science and Technology of China in 2003. He is an associate professor in the School of Computer Science and Technology, University of Science and Technology of China. His research interests include applied cryptography, trusted computing and mobile computing.

Yan Xiong received the B.S., M.S., and Ph.D degrees from University of Science and Technology of China in 1983, 1986 and 1990 respectively. He is a professor in School of Computer Science and Technology, University of Science and Technology of China. His main research interests include distributed processing, mobile computing, computer network and information security.