applied sciences

MDPI

Article

PerHelper: Helping Developers Make Better Decisions on Permission Uses in Android Apps

Guosheng Xu [1], Shengwei Xu [2], Chuan Gao [3,*], Bo Wang [3,*] and Guoai Xu [1]

[1] School of Cyberspace Security, Being University of Posts and Telecommunications, Beijing 100876, China; guoshengxu@bupt.edu.cn (G.X.); xga@bupt.edu.cn (G.X.)
[2] Information Security Research Institute, Beijing Institute of Electronic Science and Technology, Beijing 100070, China
[3] National Computer Network Emergency Response Technical Team, Coordination Center of China, Beijing 100029, China
* Correspondence: gc@cert.org.cn (C.G.); wb@cert.org.cn (B.W.)

Received: 3 August 2019; Accepted: 20 August 2019; Published: 5 September 2019

Abstract: Permission-related issues in Android apps have been widely studied in our research community, while most of the previous studies considered these issues from the perspective of app users. In this paper, we take a different angle to revisit the permission-related issues from the perspective of app developers. First, we perform an empirical study on investigating how we can help developers make better decisions on permission uses during app development. With detailed experimental results, we show that many permission-related issues can be identified and fixed during the application development phase. In order to help developers to identify and fix these issues, we develop PerHelper, an IDEplugin to automatically infer candidate permission sets, which help guide developers to set permissions more effectively and accurately. We integrate permission-related bug detection into PerHelper and demonstrate its applicability and flexibility through case studies on a set of open-source Android apps.

Keywords: permission; mobile app; permission overprivilege; app developer

1. Introduction

Mobile apps have seen widespread adoption in recent years. The number of apps on Google Play surpassed the 2.5 million mark by August 2019 [1,2]. As apps become more pervasive, they usually access sensitive data stored on mobile devices, including users' personal information (i.e., contacts) and data collected via various sensors (i.e., locations) [3,4]. As a result, it is vital to protect these sensitive data from various adversaries.

The Android operating system provides a permission-based security model to restrict whether an app can access various resources on a smartphone, including sensitive data such as contacts and locations. App developers can declare what permissions the app requires in the Android manifest file, and users will be asked to approve these permissions. However, sometimes, the permission specifications for Android can be quite complicated and confusing to developers because of the large number of permissions and different usages, which will lead to many cases of inappropriate permission requests among Android apps [5,6].

For example, some previous studies [7–12] explored the issue of permission gaps, where there are differences between the set of permissions an app requests and the set of permissions it actually uses. These gaps introduced bad consequences on security and privacy of these Android apps. For example, malware can take advantage of the permission differences to achieve their malicious goals [10].

Currently, most of the existing studies on permission-related issues have been conducted from the perspective of app users. For example, some approaches [13–15] point out security issues due to improper usage of permissions among apps and attempt to detect malware. Others [16,17] take a step further to develop tools or modify the operating system so that users can configure their own permission settings. These approaches try to inform app users of potential security and privacy vulnerabilities on permission-related issues and provide them with methods to avoid such problems. However, most of the existing works somehow neglect the other end of the mobile app ecosystem, i.e., *the app developers*.

We believe that app developers can play an essential role in exploring and fixing these issues. Different from app users, who view permission-related issues as an alert of potential security hazards, app developers view these issues as mistakes they commit during the development phase. As a result, these mistakes can prevent their apps from gaining popularity because they fail to function correctly or inadvertently being classified as malware.

On the other hand, according to existing studies [8], developers commit those mistakes for both intrinsic reasons (i.e., carelessness) and extrinsic reasons (i.e., incomplete Android API documents). This gives us opportunities to tackle the issues during the development phase, which will help developers avoid these mistakes. Solving these issues in advance will not only benefit the developers, but also benefit app users as the apps they use will be free of permission-related issues. In general, this will benefit the whole app ecosystem.

This paper investigates permission-related issues from a developer's perspective. In particular, we propose to integrate automated detection and fixing of permission-related issues in an Integrated Development Environment (IDE)to help developers make better decisions on permission use during app development. In order to achieve this goal, we first give a summary of permission-related issues from the developer's perspective. We find that apart from some widely-recognized issues (i.e., permission overprivilege (an app declares more permission than actually used)), several other issues cannot be ignored among Android apps. For example, permission underprivilege also happens to many apps, which means that they actually use more permissions than they requested. We also find that there are permission-related bugs that might cause the app to crash or function incorrectly.

Based on the summary, we develop *PerHelper*, a prototype tool designed as a plugin of Android Studio, to help developers make better decisions on permission uses. Built on static analysis techniques, PerHelper is able to infer candidate permission sets, which include required and optional permission sets for each app based on its various source files and libraries. The candidate permission sets can help developers decide which permissions should be included to solve the permission gap issues. Meanwhile, PerHelper is also able to detect and fix the permission-related bugs mentioned above. We conduct case studies on PerHelper with about 100 open-source Android projects to demonstrate its effectiveness and efficiency.

Overall, this paper makes the following main research contributions:

- We present a detailed summary on permission-related issues from the developer's perspective. The study reveals the prevalence and importance of various permission-related issues and how they can be fixed in the development phase.
- To the best of our knowledge, we are the first to detect the issue of *permission underprivilege* in Android apps and discuss its possible causes.
- We develop a comprehensive permission analysis, checking, and bug-fixing tool as a plug-in to Android Studio and evaluate it with real-world case studies on open-source Android apps.

2. Background and Related Work

In this section, we first give a brief introduction of the Android permission model, discuss the current challenges of research on the permission model, and then analyze the benefits brought by looking at the problems from the developers' perspective.

2.1. The Android Permission Model

Android uses a permission-based security model to control accesses to system resources. Although the permission enforcement mechanism has evolved from time to time, in general, permissions are required when interacting with system APIs, databases, and the message-passing system when sensitive data on the mobile devices are involved in the process. In order to gain access to these sensitive data, developers must declare appropriate permissions in the Android Manifest file [18]. For example, if an app needs to read the contact information of a smartphone, the developer will need to declare the READ_CONTACT permission (the prefix of standard Android permission name *"android.permission." is omitted throughout this paper*). In most cases, developers will refer to the official Android API documentation provided by Google [19] or use debugging information from IDEs to determine permission declarations, as was shown in a comprehensive survey on app developers [20].

Several studies have evaluated the effectiveness of the Android permission model. Felt et al. [21] and Kelley et al. [22] both conducted surveys on Android users and found that most users do not pay attention to these requests, and even fewer users actually fully comprehend them. On the other hand, studies have shown that even developers are not fully knowledgeable about permissions [5,6]. A main reason is due to the incomplete or incorrect documentation, where Android API documentation does not mention that a permission is required or the wrong permission is documented.

2.2. Analyzing Permission Usage in Android Apps

Kirin [23] was the first to use Android permissions to identify dangerous functionalities. It produces a set of permission usage rules and checks whether the requested permissions of an app break the security rules. Barrera et al. [24] performed permission analysis among a variety of categories of apps in the Android market by mapping an app to a category based on its requested permissions. Permlyzer [25] leverages the combination of runtime analysis and static examination to analyze fine-grained information of the permission uses. It can identify where a permission is actually used and how the permission is used by analyzing the context of the API calls that trigger the permission check.

STOWAWAY [8], COPES [10], and PScout [9] have addressed the permission gaps in Android apps. They first build a permission map that identifies what permissions are needed for sensitive data access (i.e., API calls) and then use static analysis on the apps to identify the permissions they may invoke at running time. Felt et al. [8] used STOWAWAY to study more than 900 apps. They found that about one-third of the apps were overprivileged. Wu et al. [11] used the permission map provided by PScout to study apps on 10 different devices, the result showing that more than 85% of apps were overprivileged. In this paper, we use the permission map provided by PScout.

2.3. Improving the Android Permission Model

Many works have attempted to improve the Android permission model. These works can be divided into three categories:

- *Modifying the current Android permission model.* For example, Reddy et al. [26] advocated to replace the *app-centric* permission model with the original *resource-centric* model, which provides more flexibility. Other approaches like Apex [27] propose a new policy enforcement framework for Android that allows a user to selectively grant permissions to apps.
- *Rewriting the apps* to perform fine-grained permission access control. For example, Jeon et al. [28] inserted additional methods before the API methods that require a permission. Those added methods will check if users grant permission or not, and if not, the permission-related API methods will not be invoked. Similarly, Liu et al. [29] used the same principle to perform fine-grained access control on third-party libraries.
- *Information flow-based fine-grained policy enforcement.* TaintDroid [30] is a representative work that tracked the sensitive data (taint source) flows during app execution at runtime. A number

of studies extended the system to track information flows. TISSA [31], MockDroid [32], and AppFence [33] propose to replace sensitive information with fake data. CleanOS [34] proposes to modify TaintDroid to enable secure deletion of information from application memory. Kynoid [35] extends TaintDroid with user-defined security policies such as restrictions on destination IP addresses to which data are released. Wang et al. [3,36] extended TaintDroid to enforce fine-grained access control based on the purpose of permission use in Android apps.
- *Crowdsourcing-based methods.* Some approaches [17,37,38] attempt to utilize crowdsourcing to recommend suitable permission settings for an app for different kinds of users. These methods are able to free the users from having to understand the mechanics behind permissions.

As we can see, most existing methods deal with the issue from the user's perspective, without considering how developers can help during the process. Typically, these methods either need to modify and repackage the app or modify the Android framework or the operating system.

The existing approaches face several important challenges:

- Many approaches need to instrument and repackage apps if they want to modify their behaviors. However, repacking apps will not always succeed. Repackaging apps will change the signature of the original app, and if the app checks its signature for integrity, it will refuse to run. Actually, many popular apps (e.g., WeChat) have already applied runtime signature checks, and doing so will be more and more popular in the future.
- Modifying the Android platform, on the other hand, is also not practical and is potentially dangerous. Introducing new permission models or frameworks in practice must have a user either install a whole new system or run them on a rooted device, which will lead to security vulnerability [39,40].
- Typically, mobile apps are the carriers of permission-related issues. However, no matter how hard we have tried to fix the issues after the apps are released, it is impossible to enforce a complete control on all instances of a particular app. We can only solve these issues completely at its very beginning or in the development phase of an app.

2.4. Android Permission from Developers' Perspective

According to the analysis above, we argue that it is more efficient and practical to solve permission-related issues from the perspective of app developers. What concerns a developer most is whether the app he/she develops functions well and meets the requirement of the targeted users. However, this does not necessarily mean that developers have no motivation to solve permission-related issues in their apps. Actually, some permission-related issues will cause the app to behave in an undesirable way (i.e., app crash or security breach). These are detrimental to the popularity of the app. Thus, any permission-related issues that may influence the functionality and robustness of an app are also important concerns for developers.

Generally, fixing permission-related issues at the development phase will have the following advantages:

- To analyze released apps, one has to decompile the app to bytecode to perform further analysis. However, at the development phase, the analysis can be conducted on source code directly, which potentially has more information than bytecode. At least, source code at the development phase will not be obfuscated, which has become a common obstacle in analyzing Android bytecode.
- Compared to app users, app developers have a much better knowledge of the functionalities and behaviors of the apps they are developing. App developers without malicious intentions will not purposely write codes that behave maliciously or contain security vulnerabilities. With the help of an automated tool, the developers can express their intentions correctly in the code snippets.

It is also true that some permission control tools are also efficient enough to assist app users. However, users know little about Android permission specifications. Therefore, it is possible that users unjustly regard a normal permission usage as inappropriate. Developers, on the other hand, have better understanding of the Android permission model and are less prone to making such mistakes.

- It is always better to solve a problem at the earliest stage, instead of postponing it to later. If all permission-related issues are solved at the development phase, all users will enjoy the functionality of the app without any concerns about permission uses. Otherwise, every user of the app will have to experience the bad effect of those issues.

There are already several studies that explored research questions relevant to mobile app developers. For example, Balebako et al. [20] explored how app developers make decisions about privacy and security. They examined whether any privacy and security behaviors were related to the characteristics of the app development companies. They interviewed 19 and surveyed 228 developers about their privacy and security behaviors. Their findings suggested that smaller companies are less likely to demonstrate positive privacy and security behaviors. Besides, developers are not aware of the data collected by third-party libraries. Wang et al. [41–43] analyzed the privacy behaviors of app developers and found that more than 70% of apps with severe privacy risks are created by 1% of developers.

The work of Bello-Ogunu et al. [44] and Vidas et al. [6] were the closest to us. They both developed plugins for the Eclipse IDE to guide developers on the set of required permissions when creating Android applications. In particular, they detected if permission overprivilege existed in the app and recommended appropriate permission settings. In comparison, our work not only deals with permission gaps, but also detects *permission-related bugs* in apps. Our paper is the first work to check the permission underprivilege and unprotected API issues in the form of an IDE plugin, and we separate the permissions used in third-party libraries during analysis. Secondly, we not only design an efficient tool for developers, but also present a comprehensive study on permission-related issues from the developer's perspective.

3. A Taxonomy of Permission-Related Issues

Android has thousands API methods that require permissions [9]. It is impossible for developers to remember all these API methods. Usually, developers would refer to the official Android API documentation provided by Google or debugging information in the IDE, which costs considerable time [20]. To make the problem even worse, studies show that about 16,000–28,000 undocumented APIs require permissions for different versions of Android [9]. Thus, it is challenging for developers to declare a precise set of permissions that they use in their apps. Developers who are not familiar with Android permission specifications are likely to make permission-related mistakes. Even for developers with advanced expertise, they would still make mistakes due to carelessness or incorrect API documentation. Developers may then need further efforts to recheck their permission declaration to avoid mistakes. Some mistakes (i.e., typos) are too subtle to identify, and this becomes another hard and time-consuming task.

In this section, we summarize several permission-related issues that are important concerns for Android developers.

3.1. Concepts and Definitions

We first define the following concepts related to Android permissions, which will be used throughout this paper.

- A ***declared permission set (DPS)*** contains all the permissions declared in the Android Manifest file of an app by the developer.
- A ***used permission set (UPS)*** contains all the permissions required by all the APIs that may be invoked during the execution of a particular Android app. We can divide this set into two subsets, the set that contains the used permissions for the code written by developers (UPS_{dev}) and the set

that contains the used permissions for the third-party libraries included in the app (UPS$_{lib}$). It is not difficult to know that:

$$UPS = UPS_{dev} \cup UPS_{lib}$$

Please note that there might be overlaps between these two sets as one permission might be used both in the developer's code and library code.

The perfect scenario would be when DPSequals UPS, which ensures not only that all the data accesses protected by permissions work well, but also that there are no unused permission declarations that might introduce various security threats. On the contrary, If DPS does not equal UPS, there exists a permission gap between the declared and the used permissions.

There are three types of permissions issues we focus on in this paper.

- ***Permission overprivilege*** occurs in an app when $|DPS - UPS| > 0$. This means the app declares more permissions than it actually uses. An app with permission overprivilege violates *the principle of least privilege* (POLP) [10], which may introduce potential security vulnerabilities.
- ***Permission underprivilege*** occurs in an app when $|UPS - DPS| > 0$. This means the app uses more permissions than it declares. As a matter of fact, permission underprivilege happens much more often than we expect. We will discuss this later.
- An ***unprotected API*** occurs when developers forget to add an exception handler to some API methods, which may throw exceptions. Figure 1 is an example code snippet in which developers failed to protect a privileged API, so that the app would crash if the API were called.

```
1   private void acquireWakeLock() {
2       PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
3       int wakeFlags;
4       if (mPedometerSettings.wakeAggressively()) {
5           wakeFlags = PowerManager.SCREEN_DIM_WAKE_LOCK | PowerManager.ACQUIRE_CAUSES_WAKEUP;
6       }
7       else if (mPedometerSettings.keepScreenOn()) {
8           wakeFlags = PowerManager.SCREEN_DIM_WAKE_LOCK;
9       }
10      else {
11          wakeFlags = PowerManager.PARTIAL_WAKE_LOCK;
12      }
13      wakeLock = pm.newWakeLock(wakeFlags, TAG);
14      wakeLock.acquire();
15  }
```

Figure 1. Example Android code with the API level under 23. The API method *acquire*() requires the *WAKE_LOCK* permission (Line 14). The developer is supposed to at least catch the security exception in this function in case the permission is not granted at runtime due to some reasons.

3.2. The Permission Gap Problem

As proposed above, the problem of the permission gap can be generally classified into two sub-problems, permission overprivilege and permission underprivilege.

3.2.1. Permission Overprivilege

The issue has been explored by several previous studies on permission gaps [6,8–12,45,46]. Recent studies showed that this is a quite prevalent problem. For example, in a recent study [12] that leveraged a combination of static analysis and dynamic analysis, people found that 83.4% of apps suffered from permission overprivilege.

Declaring more permissions than needed seems to have no influence on developers, since it does no harm the functionality of an app. However, Bartel et al. [10] showed that malware can leverage the unused permissions to achieve their malicious goals. Meanwhile, several approaches [15,39,40,47–49] have explored methods to detect malware or ranking apps according to the permission usage of each app to recommend apps to users. Especially, over-declaring sensitive permissions like *ACCESS_FINE_LOCATION* (a kind of Android permission used for collecting precise locations of app users) has a high probability of being recognized as malware or getting low rankings. In both situations, an app is likely to lose its popularity even though its functionality remains unaffected.

3.2.2. Permission Underprivilege

In contrast to permission overprivilege, the issue of *permission underprivilege* has not been raised very often. Generally, related functionalities in the app would be affected if certain permissions are missing. Furthermore, these apps may throw *SecurityException* when it invokes an API method that requires a certain permission, but the developer has forgotten to declare it. This would lead to the app crashing, which is the last thing a developer wants to see.

In order to explore this issue, we analyzed 958 popular apps from 10 app categories on Google Play. For each app, we first calculated the *DPS* for an app from the Android Manifest file. Then, we used methods in [12] to estimate the *UPS* for each app. Finally, we compared the differences between the two sets to determine whether permission underprivilege exists.

The statistics on underprivileged apps are shown in Table 1. It is somewhat surprising to discover that more than 90% of apps are underprivileged; in particular, more than 98% of apps in the *ENTERTAINMENT* category are underprivileged. On average, an underprivileged app uses nearly four permissions that are not declared.

Table 1. Permission underprivilege analysis results.

Category	**# of Underpriv.Apps**	**% of Apps**	**Average # of under Declared Permissions**
WALLPAPER	90	84.1%	3.23
WIDGETS	98	95.1%	4.60
BOOKS AND REFERENCE	98	93.3%	3.43
BUSINESS	91	86.7%	3.48
COMICS	97	89.8%	3.06
EDUCATION	94	89.5%	5.62
ENTERTAINMENT	105	98.1%	3.84
FINANCE	91	86.7%	3.53
LIFESTYLE	101	96.2%	4.33
TOOLS	93	88.6%	4.70
TOTAL	958	90.8%	3.98

When we looked into this issue, we found that there were generally two reasons behind it: ignoring the permissions required by third-party libraries and using the wrong permissions.

Third-party libraries. Developers include third-party libraries in their apps to enrich the functionality or meet other purposes [36]. Third-party libraries have occupied a large portion of code in Android apps [50]. Third-party libraries of Android also follow the Android permission model, so developers are required to include permissions used by these libraries in the Android Manifest file as well. However, when developers declare permissions in the corresponding manifest file, they typically consider only their custom-written code, instead of the requirements of other libraries. Thus, it is possible that developers miss some permissions requested by third-party libraries, which would lead to permission underprivilege, whose implications for developers have been discussed above.

Another phenomenon, first introduced in [51], is that some ad libraries may try to dynamically check certain permissions and make use of these permissions to invoke certain API methods if they are available to the apps. For example, the app "cn.leave.sdclean" is a cleanness tool for the SD card. It does not declare any location permissions, but we find that there are some location-related API invocations in the libraries it contains, such as *"baidu/mobstats"*, *"google/ads"*, *"inmobi"*, and *"kyview"*.

According to a recent study on ad libraries, 27% of apps among 63,105 apps have at least one permission used by the libraries, but unused by the apps' main logic [29]. This indicates that developers would need to declare extra permissions for third-party libraries after completing the declaration of their own code, which can be annoying and may cause mistakes.

Wrong permission usage. Another reason behind the permission underprivilege is that some permissions are not declared because of the carelessness of or mistakes by developers. For example, the developer typed the wrong permission or deprecated permissions, which was also raised by [8].

A misused permission may cause both permission overprivilege and underprivilege at the same time. In the worst case, the app with the wrong permissions may be recognized as a potential threat to users' privacy and also crashes when users are running it, which harms the popularity of the app.

To investigate this issue in detail, we analyzed 7982 popular apps from 10 categories in the Google Play market. For each app, we first extracted the DPS from the Android Manifest files. We then checked each permission in the DPS to see if it was misspelled or had been deprecated in the SDK version the app was using. Overall, we found 472 apps with the wrong permissions, which represents about 6% of the apps. Although this was a small portion, it is still surprising that the developers of these apps failed to correct these trivial mistakes.

Table 2 shows the top 10 most frequently misused/mistyped permissions. Some of them are just one or two letters different from the correct one (e.g., $ACCESS\_FIND\_LOCATION$), so it is difficult to detect the mistakes without an automated tool. We also searched some of the mistyped permissions on Google and found that they also appeared in some technology forums or open-source code. It is quite possible that developers just copy these wrong permissions without double checking.

Table 2. The top 10 wrong permissions used in apps.

Wrong Permissions	# of Apps	‰ of Apps
WRITE	64	8.02‰
REORDER_TASKS	54	6.77‰
CAMER	51	6.39‰
ACCESS_GPS	43	5.39‰
PERMISSION_NAME	38	4.76‰
ACCES_MOCK_LOCATION	37	4.64‰
ACCESS_LOCATION	31	3.88‰
ACCESS_FIND_LOCATION	26	3.26‰
LOCATION	20	2.51‰
SET_ORIENTATION	12	1.50‰

Besides misspellings, there are also some deprecated permissions, such as $ACCESS\_GPS$, which is replaced by $ACCESS\_FINE\_LOCATION$. This can also be attributed to the carelessness of developers, when they forget to update the permissions or copied out-of-date code.

3.3. The Problem of Unprotected APIs

In previous Android API levels (before 23), the permissions of an app were granted at anytime as long as they were accepted by users at install time. However, many approaches [26–28,52–57] have been proposed to offer more fine-grained permission control. One typical functionality of these mechanisms is to allow users to control the permissions of each app after installation. The removal of permissions proposes challenges to those apps with *unprotected APIs*, which would cause crashing.

The new Android system, which was released in summer 2015, proposed a brand-new permission system [58]. Users grant permissions to apps while the app is running, not when they install the app. There were also some changes to the way developers write the code. Every time a developer uses an API that requires a "dangerous" permission (permissions that grant access to sensitive resources, e.g., contacts), he/she needs to use a combination of APIs *ContextCompat.checkSelfPermission()* and *ActivityCompat.requestPermissions()* to make sure the permission has been granted by users. Otherwise, the program would also throw *SecurityException* and lead to the app crashing.

Kennedy et al. [59] studied the effects of removing permissions from Android apps using the old permission model and identified 39 out of 662 (5.9%) that crashed directly because of the removal. We did not obtain the data for the apps using the new permission model. This problem seems not so common, but on the one hand, it can lead to quite unpleasant results (app crashing); on the other hand, it is certain that the frequency at which permissions are denied at runtime would increase greatly. Thus, the issue of unprotected APIs should raise an important concern for developers in the future.

3.4. Summary

We summarized permission-related issues raised by pervious work and also conducted program analysis to identify new issues. From the developers' perspective, on the one hand, these issues typically lead to unpleasant results and should be tackled as early as possible in the development phase; on the other hand, the issues are difficult and time-consuming to solve only by developers. Thus, it is necessary to use automated methods to help developers fix those issues in the developing process. This motivated us to propose *PerHelper*, an IDE plugin to fix permission-related issues for developers automatically, effectively, and accurately.

4. PerHelper Design and Implementation

In this section, we present the detailed design and implementation of PerHelper, an IDE plug-in to help developers understand and fix permission-related issues.

4.1. Overview

The goal of PerHelper is to **design a plugin for Android Studio and IntelliJ IDEA (the base of Android Studio) to help Android app developers make better decisions about permission settings**. In particular, PerHelper has two main functions. The first one is to help the developer to decide the exact permission sets he/she should use, based on the analysis of the permission declared and used in each app. The second task is to detect permission-related bugs such as unprotected API calls and wrong permissions. Thus, PerHelper is designed to be invoked by app developers during the app development phase. For example, developers could perform a permission check to generate the precise permission sets before app release or they could invoke PerHelper to detect permission-related bugs once they have declared a permission set manually.

In order to help developers understand the best way to declare permissions in the Manifest file, we first introduce two permission set concepts as follows.

- A ***Required Permission Set (RPS)*** contains all the permissions that must be included for the code written by the developers (custom code) to function correctly. Permissions in this set must be declared to guarantee the overall functionality of the app. Actually, *RPS* is the same as *UPS_dev* defined earlier, but here, we introduce it from the perspective of developers.
- An ***Optional Permission Set (OPS)*** is defined as the set of permissions used *only* by third-party libraries. Obviously, we have:

$$OPS = UPS_{lib} - UPS_{dev}$$

As shown earlier, permissions in this set are likely to be ignored by developers. The reason why we regard this permission set as "optional" is that previous studies [29,60] showed that some third-party libraries (i.e., ad libraries) might want to retrieve all the details on a user, including location, contact list, viewing history, etc. Typically, they request many sensitive permissions. Rational developers would decide which permissions should be provided to these libraries, instead of giving the libraries full capability to access all sensitive information of a mobile user. Developers should make their decisions based on the tradeoff between giving library more access and protecting user privacy.

One might question whether ignoring some permissions used by third-party libraries would cause the app to crash, as discussed in Section 3. To tackle this, we will first use our tool to detect and fix the code to avoid potential crashes caused by those permissions, whose details will be shown later. More good news is that according to Kennedy et al. [59], third-party libraries are often able to handle the situation gracefully when the permissions they requested are missed. In practice, we also find few cases when removing permissions in OPS leads to app crashing.

After performing analysis on permissions based on the app source code, PerHelper calculates the RPS and OPS sets and makes suggestions to developers on permission declaration:

- All permissions in RPS should be declared; the developer would be warned if any permissions in RPS was missing in the Manifest file.
- Only permissions in RPS are required to be declared; the developer would also be warned if any permissions out of RPS have been declared in the Manifest file.
- All permissions in OPS can be declared at the developer's discretion, based on his/her understanding of the library functionality and privacy concerns of the potential users of his/her app.

4.2. PerHelper Implementation

An overview of the PerHelper architecture is shown in Figure 2. In particular, PerHelper includes the following main steps: (1) data preprocessing and collection, (2) calculating required and optional permission sets, (3) checking permission correctness, and (4) detecting unprotected APIs. In the following, we will discuss each of these steps in detail.

4.2.1. Data Preprocessing

PerHelper performs analysis on multiple sources, including Java source code, native source code, and JARfiles to analyze permission usage in order to generate permission sets RPS and OPS.

We take advantage of the parser integrated in IntelliJ IDEA to deal with Java codes. The IDE has a parser framework called PSI (Program Structure Interface), which represents the whole project as a tree of PSI elements, such as *.java* files and method-invoking expressions.

For C/C++ code, due to the flexibility of these languages and the vagueness of the NDKdocumentation, we decided to roughly parse them with regular expressions and offer a conservative permission set. We summarized a list of sensitive APIs in native code and performed a broad matching of such APIs in the corresponding C/C++ code. Note that, as we only performed analysis on the source code, we thus did not handle native external libraries (.so), and we also could not handle reflection calls.

Lastly, we wrote a Java bytecode parser to extract all APIs used in JAR files for permission analysis.

4.2.2. Calculating Permission Sets

In order to determine the permission sets RPS and OPS for each app, PerHelper checks data from all three sources.

We first iterated through all Java method call expressions using IntelliJ IDEA's PSI utility and searched for each method name in an API-to-permission mapping generated with PScout [9]. When a method is found in the mapping, PerHelper checks the package name of the class containing that expression. If it is the name of a third-party library compiled from LibRadar [61], all permissions required by the API would be added to OPS. Otherwise, new permissions would be added to RPS, while the API name and the file containing that invoking expression would be recorded in case the developer wants to know why those permissions are required.

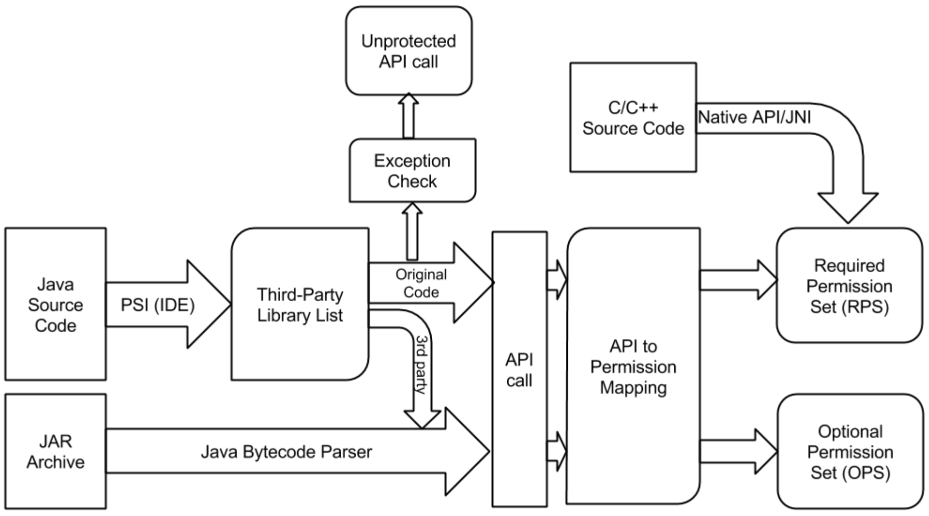


Figure 2. Overall architecture of PerHelper. PSI, Program Structure Interface.

JAR files were treated similarly. We assumed that developers would only use JAR files to import third-party libraries; thus, all permissions used by API calls in the JAR files would be added to OPS.

If the project contains C/C++ source files, PerHelper assumes that they are all essential to the project. Android has very little documentation about native APIs; none of them have clearly clarified which permission is needed for each function. Therefore, we cannot provide an accurate mapping even though we have manually reviewed the comments in all stable NDK headers. Besides native APIs, the JNIfunction call, which is somewhat similar to Android Intents, is also a tricky mechanism. To make sure the app works correctly, PerHelper provides any permissions that appear useful for native codes. If the developer is sure that some permissions are unnecessary, he or she has the freedom to ignore the suggestions.

4.2.3. Permission Correctness Checking

When RPS and OPS are completed, PerHelper compares them with permissions requested in AndroidManifest.xml and presents their differences to the developer. Specifically, if an applied permission has less than two different letters with a required or optional permission, then PerHelper will present a warning indicating that it may be mistyped.

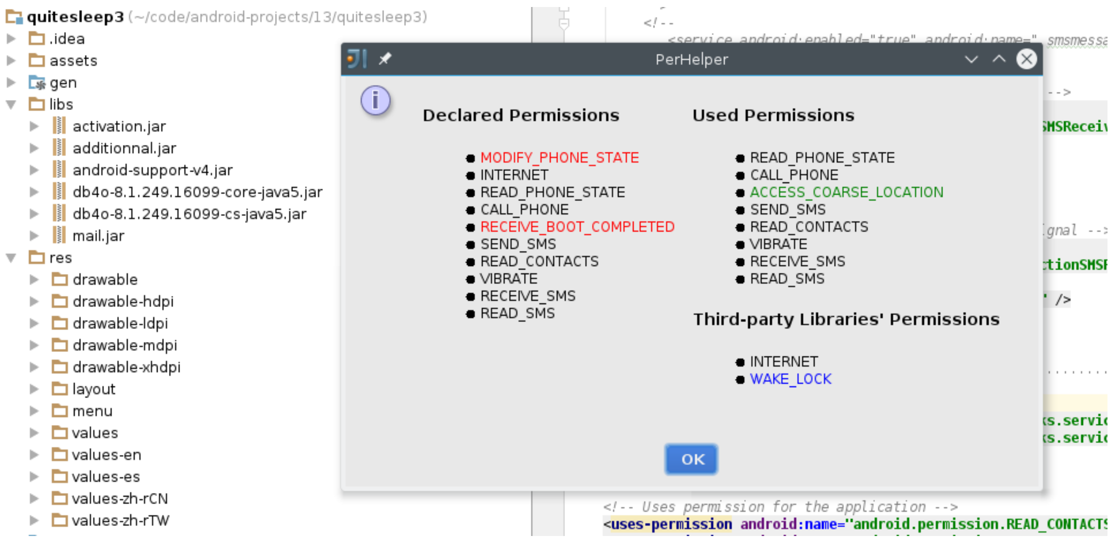
4.2.4. Detecting Unprotected APIs

Another important feature of PerHelper is to detect permission-related bugs, such as potentially uncaught SecurityException raised by API calls, which may crash the app. The IDE itself will give a warning for some of the APIs that can throw SecurityException, but for some reason, it fails to identify all of them. For instance, we have verified that $LocationManager.requestLocationUpdates()$ will crash the application if the permission is missing; however, the analyzer returns an empty thrown list for this method.

Therefore, we have manually reviewed the Android API reference and made a list of APIs that has a description on security exception in its documentation. If an API call either has SecurityException in its thrown list or the API name appears in our list, then the expression should be protected by a try-catch block. PerHelper will give a warning for every API call that does not meet this requirement.

5. Evaluation

We implemented PerHelper as a plug-in in the Android Studio environment. Figure 3 presents a screenshot showing the result of the permission analysis of PerHelper. Three permission sets are shown in the figure, where declared permissions represent all permissions declared by the app, used permissions represent all permissions used by the code except libraries, and library permissions are listed separately. Permissions declared but not used (shown in red) and permissions used but not declared (shown in green) are highlighted, suggesting that the developer should take appropriate actions to fix the issues. Permissions from third-party libraries are optional, so the developer can choose whether to include them. The permissions in blue suggest the new permissions introduced by the third-party libraries, which were not invoked in the custom code written by the app developers.


```
quitesleep3 (~/code/android-projects/13/quitesleep3)
  .idea
  assets
  gen
  libs
    activation.jar
    additionnal.jar
    android-support-v4.jar
    db4o-8.1.249.16099-core-java5.jar
    db4o-8.1.249.16099-cs-java5.jar
    mail.jar
  res
    drawable
    drawable-hdpi
    drawable-ldpi
    drawable-mdpi
    drawable-xhdpi
    layout
    menu
    values
    values-en
    values-es
    values-zh-rCN
    values-zh-rTW
```

PerHelper

Declared Permissions

- MODIFY_PHONE_STATE
- INTERNET
- READ_PHONE_STATE
- CALL_PHONE
- RECEIVE_BOOT_COMPLETED
- SEND_SMS
- READ_CONTACTS
- VIBRATE
- RECEIVE_SMS
- READ_SMS

Used Permissions

- READ_PHONE_STATE
- CALL_PHONE
- ACCESS_COARSE_LOCATION
- SEND_SMS
- READ_CONTACTS
- VIBRATE
- RECEIVE_SMS
- READ_SMS

Third-party Libraries' Permissions

- INTERNET
- WAKE_LOCK

OK

```
<!--
<service android:enabled="true" android:name=" smsmessa
-->
SMSReceiv
gnal -->
tionSMSP
' />
s.servic
s.servic
<!-- Uses permission for the application -->
<uses-permission android:name="android.permission.READ_CONTACTS
```

Figure 3. A screenshot showing permission set analysis results for the app “Android SMSPopup”.

5.1. Case Studies

Because PerHelper is designed to analyze source code for developers, we downloaded 100 open-source Android apps from GitHub (note that the 100 apps were randomly choose from the F-Droid project: https://f-droid.org/) to evaluate PerHelper. In particular, We loaded each app into Android Studio (three apps failed this step), using PerHelper to check whether it had any permission-related issues discussed above. The overall result on the remaining 97 apps is shown in Table 3.

About 23% of the apps were permission overprivileged, and 36% were permission underprivileged. A small fraction of the apps (about 3%) used the wrong permissions, all of which were deprecated permissions. About 22% of the apps failed to protect certain API calls, which may result in crashing when the APIs are called. Table 4 shows 5 examples of them. On the other hand, only eight apps included third-party libraries, and no libraries requested extra permissions.

It is somewhat surprising to find that the results on open-source apps were different from the results we obtained from Google Play apps. Firstly, permission overprivilege and underprivilege were not as common. A main cause for this difference is that open-source projects do not declare many permissions (2.7 permissions on average, compared to 7.5 permissions declared by Google Play apps). Thus it was less likely to find permission-related issues in these open-source apps.

Table 3. Permission-related issues detected in open source projects.

Issue	# of Apps	# of Apps with the Issue	% of Apps
Permission overprivilege	97	22	22.7%
Permission underprivilege	97	34	35.8%
Extra third-party permissions	8	7	87.5%
Wrong permissions	97	3	3.09%
Unprotected APIs	97	21	21.6%

Table 4. Examples of open-source apps with unprotected API methods.

Name	Unprotected API
RingsExtended	MediaPlayer#setDataSource
Android-SMSpopup	ActivityManager#getRunningTasks
Android-scripting	ActivityManager#gerRunningServices
Antriplog	LocationManager.requestLocationUpdates
Radar	LocationManager.requestLocationUpdates

5.2. Performance Overhead

We also evaluated the execution performance of PerHelper. We selected six open-source apps with different sizes and measured the time cost of PerHelper to generate permission sets and detect unprotected API methods. When an app was just loaded into the IDE (cold start), it cost more time for PerHelper to run compared to when the app had been used in the IDE for a while (warm start). The reason was that the IDE would not load all the code when the app was first loaded. However, when the app was edited and viewed several times, the IDE would cache some snippets of code to gain better performances, which also made it faster for PerHelper to parse the code.

The result is shown in Figure 4. The processing time of PerHelper was roughly proportional to the size of the app it attempted to analyze. For the cold start case, the time cost to complete all its tasks for an app with about 50K lines of code was about 6 s. As for the warm start case (which is a more common situation PerHelper), the performance was much better, with the cost for an app with about 10K lines of code requiring less than 1 s and the cost for an app with about 50K lines of code needing only less than 2 s. Compared to the time used to look up API documentation or search the Internet, PerHelper introduced acceptable overhead for developers.

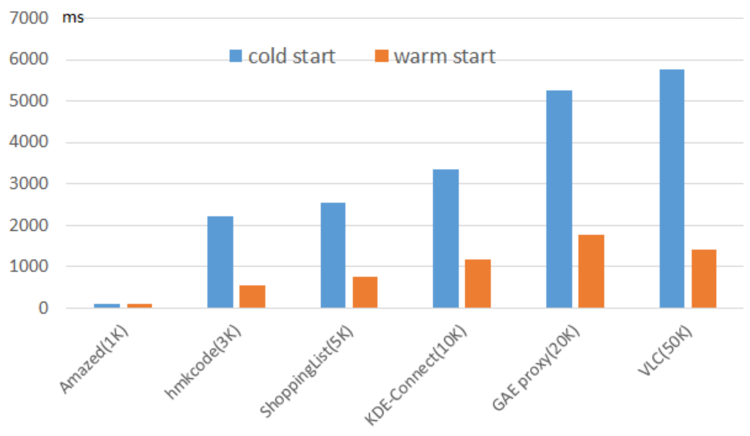


Figure 4. Execution time of PerHelper. The vertical axis denotes the time in milliseconds used to generate permission sets and detect unprotected API methods. The horizontal axis denotes the app name and lines of code.

6. Discussions

Handling native code. The current mechanism to detect permissions in native code is not accurate enough. For JNI functions, a complete control flow analysis is needed to deal with complex situations. For native APIs, an extensive study on the source code of native libraries will improve the accuracy of the permission map.

Intents and Broadcast Receivers. PerHelper does not have the ability to deal with Android Intents and Broadcast Receivers. If an app makes use of a privileged intent, PerHelper will miss a required permission.

Permission-API mapping. PerHelper uses a static API-permission map for all API versions, while the mapping between APIs and permissions is actually mutable in different versions of Android systems. If an app is designed to be compatible with an older version of Android, the permission set provided by PerHelper may fail to reach high accuracy.

The difference between commercial apps and open-source apps. We evaluated the performance of PerHelper using open-source projects, which significantly differ from commercial apps in Google Play. Therefore, to accurately measure its usability, it is better to involve some real developers at companies to offer feedback.

7. Concluding Remarks

This paper revisited and dissected Android permissions from the perspective of app developers. We conducted a comprehensive empirical study to identify the permission-related issues that are most relevant and important to developers. Using detailed analysis of the experimental data, we showed that it is more desirable to solve the issues during the app development phase. We also built PerHelper, an IDE plug-in to guide developers to declare permissions automatically and correctly and identify permission-related mistakes. PerHelper was demonstrated to be effective and practical through case studies on a set of open-source Android apps.

Author Contributions: G.X. (Guosheng Xu) conceived of and organized the research work; G.X. (Guosheng Xu), S.X., and C.G. conducted the experiments and analyzed the data; G.X. (Guosheng Xu) and S.X. wrote the paper; C.G., B.W., and G.X. (Guoai Xu) checked and verified the paper. All authors reviewed the paper.

Funding: This work was funded by the National Key Research and Development Program of China (No. 2017YFB0801903), the National Key Program for Basic Research of China (Grant No. 2017-JCJQ-ZD-043), and the National Secrecy Scientific Research Program of China (No. BMKY2018802-1).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Number of Android Applications. Available online: https://www.appbrain.com/stats (accessed on 2 August 2019).
2. Wang, H.; Li, H.; Guo, Y. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In Proceedings of the World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 1988–1999.
3. Wang, H.; Hong, J.; Guo, Y. Using text mining to infer the purpose of permission use in mobile apps. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, Osaka, Japan, 7–11 September 2015; pp. 1107–1118.
4. Kumar, R.; Zhang, X.; Khan, R.U.; Sharif, A. Research on data mining of permission-induced risk for android IoT devices. *Appl. Sci.* **2019**, *9*, 277. [CrossRef]
5. Stevens, R.; Ganz, J.; Devanbu, P.; Chen, H.; Filkov, V. Asking for (and About) Permissions Used by Android Apps. In Proceedings of the the 10th Working Conference on Mining Software Repositories (MSR'13), San Francisco, CA, USA, 18–19 May 2013.
6. Vidas, T.; Christin, N.; Cranor, L. Curbing Android permission creep. In Proceedings of the 20th International Conference on World Wide Web, Hyderabad, India, 28 March–1 April 2011.

7. Wu, S.; Liu, J. Overprivileged Permission Detection for Android Applications. In Proceedings of the ICC 2019—2019 IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019; pp. 1–6.
8. Felt, A.P.; Chin, E.; Hanna, S.; Song, D.; Wagner, D. Android Permissions Demystified. In Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11), Chicago, IL, USA, 17–21 October 2011.
9. Au, K.W.Y.; Zhou, Y.F.; Huang, Z.; Lie, D. PScout: Analyzing the Android Permission Specification. In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12), Raleigh, NC, USA, 16–18 October 2012.
10. Bartel, A.; Klein, J.; Le Traon, Y.; Monperrus, M. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12), Essen, Germany, 3–7 September 2012.
11. Wu, L.; Grace, M.; Zhou, Y.; Wu, C.; Jiang, X. The Impact of Vendor Customizations on Android Security. In Proceedings of the the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS '13), Berlin, Germany, 4–8 November 2013.
12. Wang , H.; Guo, Y.; Tang, Z.; Bai, G.; Chen, X. Reevaluating android permission gaps with static and dynamic analysis. In Proceedings of the 2015 IEEE Global Communications Conference (GLOBECOM), San Diego, CA, USA, 6–10 December 2015; pp. 1–6.
13. Yang, K.; Zhuge, J.; Wang, Y.; Zhou, L.; Duan, H. IntentFuzzer: Detecting Capability Leaks of Android Applications. In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, Kyoto, Japan, 3–6 June 2014.
14. Gibler, C.; Crussell, J.; Erickson, J.; Chen, H. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12), Vienna, Austria, 13–15 June 2012.
15. Grace, M.; Zhou, Y.; Zhang, Q.; Zou, S.; Jiang, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12), Low Wood Bay, UK, 25–29 June 2012.
16. Liu, B.; Lin, J.; Sadeh, N. Reconciling Mobile App Privacy and Usability on Smartphones: Could User Privacy Profiles Help? In Proceedings of the 23rd International Conference on World Wide Web, Seoul, Korea, 7–11 April 2014.
17. Lin, J.; Liu, B.; Sadeh, N.; Hong, J.I. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In Proceedings of the Symposium On Usable Privacy and Security (SOUPS 2014), Menlo Park, CA, USA, 9–11 July 2014.
18. System Permissions. Available online: http://developer.android.com/guide/topics/security/permissions.html (accessed on 2 August 2019).
19. Android API References. Available online: http://developer.android.com/reference/packages.html (accessed on 2 August 2019).
20. Balebako, R.; Marsh, A.; Lin, J.; Hong, J.; Cranor, L. The Privacy and Security Behaviors of Smartphone App Developers. In Proceedings of the Workshop on Usable Security (USEC), San Diego, CA, USA, 23 February 2014.
21. Felt, A.P.; Ha, E.; Egelman, S.; Haney, A.; Chin, E.; Wagner, D. Android permissions: User attention, comprehension, and behavior. In Proceedings of the Eighth Symposium on Usable Privacy and Security, Washington, DC, USA, 11–13 July 2012.
22. Kelley, P.G.; Consolvo, S.; Cranor, L.F.; Jung, J.; Sadeh, N.; Wetherall, D. A conundrum of permissions: Installing applications on an Android smartphone. In *Financial Cryptography and Data Security*; Springer: Berlin/Heidelberg, Germany, 2012.
23. Enck, W.; Ongtang, M.; McDaniel, P. On Lightweight Mobile Phone Application Certification. In Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), Chicago, IL, USA, 9–13 November 2009.
24. Barrera, D.; Kayacik, H.G.; van Oorschot, P.C.; Somayaji, A. A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10), Chicago, IL, USA, 4–8 October 2010.

25. Xu, W.; Zhang, F.; Zhu, S. Permlyzer: Analyzing permission usage in Android applications. In Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, USA, 4–7 November 2013.
26. Reddy, N.; Jeon, J.; Vaughan, J.; Millstein, T.; Foster, J. *Application-Centric Security Policies on Unmodified Android*; Technical Report; UCLA Computer Science Department: Los Angeles, CA, USA, 2011.
27. Nauman, M.; Khan, S.; Zhang, X. 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, Beijing, China, 13–16 April 2010.
28. Jeon, J.; Micinski, K.K.; Vaughan, J.A.; Fogel, A.; Reddy, N.; Foster, J.S.; Millstein, T. Dr. Android and mr. Hide: Fine-grained permissions in Android applications. In Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Raleigh, NC, USA, 19 October 2012.
29. Liu, B.; Liu, B.; Jin, H.; Govindanz, R. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys '15), Florence, Italy, 18–22 May 2015.
30. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst. (TOCS)* **2014**, *32*, 5. [CrossRef]
31. Zhou, Y.; Zhang, X.; Jiang, X.; Freeh, V.W. Taming information-stealing smartphone applications (on android). In Proceedings of the International Conference on Trust and Trustworthy Computing, Pittsburgh, PA, USA, 22–24 June 2011; pp. 93–107.
32. Beresford, A.R.; Rice, A.; Skehin, N.; Sohan, R. Mockdroid: Trading privacy for application functionality on smartphones. In Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, Phoenix, AZ, USA, 1–3 March 2011; pp. 49–54.
33. Hornyack, P.; Han, S.; Jung, J.; Schechter, S.; Wetherall, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 17–21 October 2011; pp. 639–652.
34. Tang, Y.; Ames, P.; Bhamidipati, S.; Bijlani, A.; Geambasu, R.; Sarda, N. CleanOS: Limiting mobile data exposure with idle eviction. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), Hollywood, CA, USA, 8–10 October 2012; pp. 77–91.
35. Schreckling, D.; KöStler, J.; Schaff, M. Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Inf. Secur. Tech. Rep.* **2013**, *17*, 71–80. [CrossRef]
36. Wang, H.; Li, Y.; Guo, Y.; Agarwal, Y.; Hong, J.I. Understanding the purpose of permission use in mobile apps. *ACM Trans. Inf. Syst. (TOIS)* **2017**, *35*, 43. [CrossRef]
37. Lin, J.; Amini, S.; Hong, J.I.; Sadeh, N.; Lindqvist, J.; Zhang, J. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp'12), Pittsburgh, PA, USA, 5–8 September 2012.
38. Ismail, Q.; Ahmed, T.; Kapadia, A.; Reiter, M.K. Crowdsourced Exploration of Security Configurations. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Korea, 18–23 April 2015.
39. Zhou, Y.; Jiang, X. Dissecting Android malware: Characterization and evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–23 May 2012.
40. Zhou, Y.; Wang, Z.; Zhou, W.; Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In Proceedings of the NDSS 2012, San Diego, CA, USA, 5–8 February 2012.
41. Wang, H.; Liu, Z.; Guo, Y.; Chen, X.; Zhang, M.; Xu, G.; Hong, J. An explorative study of the mobile app ecosystem from app developers' perspective. In Proceedings of the 26th International Conference on World Wide Web, Perth, Australia, 3–7 April 2017; pp. 163–172.
42. Wang, H.; Wang, X.; Guo, Y. Characterizing the global mobile app developers: A large-scale empirical study. In Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, Montreal, QC, Canada, 25 May 2019; pp. 150–161.
43. Wang, H.; Liu, Z.; Liang, J.; Vallina-Rodriguez, N.; Guo, Y.; Li, L.; Tapiador, J.; Cao, J.; Xu, G. Beyond google play: A large-scale comparative study of chinese android app markets. In Proceedings of the Internet Measurement Conference 2018, Boston, MA, USA, 31 October–2 November 2018; pp. 293–307.

44. Bello-Ogunu, E.; Shehab, M. PERMITME: Integrating Android permissioning support in the IDE. In Proceedings of the 2014 Workshop on Eclipse Technology eXchange, Portland, OR, USA, 21 October 2014.
45. Johnson, R.; Wang, Z.; Gagnon, C.; Stavrou, A. Analysis of Android applications' permissions. In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C), Gaithersburg, MD, USA, 20–22 June 2012.
46. Bartel, A.; Klein, J.; Monperrus, M.; Le Traon, Y. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. *IEEE Trans. Softw. Eng. (TSE)* **2014**, 40, 617–632.
47. Zhang, Y.; Yang, M.; Xu, B.; Yang, Z.; Gu, G.; Ning, P.; Wang, X.S.; Zang, B. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13), Berlin, Germany, 4–8 November 2013.
48. Xu, R.; Saïdi, H.; Anderson, R. Aurasium: Practical Policy Enforcement for Android Applications. In Proceedings of the USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012.
49. Zhu, H.; Xiong, H.; Ge, Y.; Chen, E. Mobile app recommendations with security and privacy awareness. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014.
50. Wang, H.; Ma, Z.; Guo, Y.; Chen, X. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15), Baltimore, MD, USA, 13–17 July 2015.
51. Stevens, R.; Gibler, C.; Crussell, J.; Erickson, J.; Chen, H. Investigating user privacy in Android ad libraries. In Proceedings of the Workshop on Mobile Security Technologies (MoST), San Diego, CA, USA, 28–29 February 2012.
52. Backes, M.; Gerling, S.; Hammer, C.; Maffei, M.; von Styp-Rekowsky, P. AppGuard-enforcing user requirements on Android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*; Springer International Publishing: Basel, Switzerland, 2013.
53. Bartel, A.; Klein, J.; Monperrus, M.; Allix, K.; Le Traon, Y. Improving privacy on Android smartphones through in-vivo bytecode instrumentation. *arXiv* **2012**, arXiv:1208.4536 .
54. Davis, B.; Sanders, B.; Khodaverdian, A.; Chen, H. I-arm-droid: A rewriting framework for in-app reference monitors for Android applications. *Mob. Secur. Technol.* **2012**, *2*, 1–7
55. Davis, B.; Chen, H. RetroSkeleton: Retrofitting Android Apps. In Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys'13), Taipei, Taiwan, 25–28 June 2013.
56. Que, P.; Guo, X.; Wang, Z. Lightweight Optimization of Android Permission Model. In *Proceedings of the 4th International Conference on Computer Engineering and Networks*; Springer: Berlin, Germany, 2015.
57. Wu, L.; Du, X.; Zhang, H. An effective access control scheme for preventing permission leak in Android. In Proceedings of the 2015 International Conference on Computing, Networking and Communications (ICNC), Garden Grove, CA, USA, 16–19 February 2015.
58. Requesting Permissions at Run Time. Available online: http://developer.android.com/training/permissions/requesting.html (accessed on 2 August 2019).
59. Kennedy, K.; Gustafson, E.; Chen, H. Quantifying the effects of removing permissions from Android applications. In Proceedings of the IEEE MoST 2013, San Francisco, CA, USA, 23 May 2013.
60. Book, T.; Pridgen, A.; Wallach, D.S. Longitudinal analysis of Android ad library permissions. *arXiv* **2013**, arXiv:1303.0857.
61. Ma, Z.; Wang, H.; Guo, Y.; Chen, X. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 653–656.