Android Custom Permissions Demystified: A Comprehensive Security Evaluation

Rui Li, Wenrui Diao, Zhou Li, *Senior Member, IEEE*, Shishuai Yang, Shuang Li, and Shanqing Guo

Abstract—Permission is the fundamental security mechanism for protecting user data and privacy on Android. Given its importance, security researchers have studied the design and usage of permissions from various aspects. However, most of the previous research focused on the security issues of *system permissions*. Overlooked by many researchers, an app can use *custom permissions* to share its resources and capabilities with other apps. However, the security implications of using custom permissions have not been fully understood. In this paper, we systematically evaluate the design and implementation of Android custom permissions. Notably, we built an automatic fuzzing tool, called `CuPerFuzzer+`, to detect custom permission related vulnerabilities existing in the Android OS. `CuPerFuzzer+` treats the operations of the permission mechanism as a black-box and executes massive targeted test cases to trigger privilege escalation. In the experiments, `CuPerFuzzer+` discovered 5,932 effective cases with 47 critical paths successfully. Through investigating these vulnerable cases and analyzing the source code of Android OS, we further identified a series of severe design shortcomings lying in the Android permission framework, including *dangling custom permission*, *inconsistent permission-group mapping*, *custom permission elevating*, *inconsistent permission definition*, *dormant permission group*, and *inconsistent permission type*. Exploiting these shortcomings, a malicious app can access unauthorized platform resources. On top of these observations, we propose three general design guidelines to secure custom permissions. Our findings have been acknowledged by the Android security team and assigned `CVE-2020-0418`, `CVE-2021-0306`, `CVE-2021-0307`, and `CVE-2021-0317`.

Index Terms—Android security, custom permission, automatic analysis

1 Introduction

As the most popular mobile platform, Android provides rich APIs and features to support third-party apps developments. For security concerns, Android also designs a series of mechanisms to prevent malicious behaviors. Among these mechanisms, permission is the fundamental one of Android OS: any app must request specific permissions to access the corresponding sensitive user data and system resources.

On account of the importance of the permission mechanism, its design and usage have been studied by lots of previous research from many aspects, such as permission models [1], [2], [3], permission usage [4], [5], [6], and malware detection [7], [8], [9]. Along with the continuous upgrade of Android OS, the underlying architecture of the permission mechanism becomes more and more complicated. Its current design and implementation are seemingly complete enough. However, overlooked by most of the previous research, Android allows apps to define their own permissions, say *custom permissions* [10], and use them to regulate the sharing of their resources and capabilities with other apps. Since custom permission is not related to system capabilities by design, its range of action is supposed to be confined by the app defining it. Therefore, in theory, dangerous operations cannot be executed through custom permissions, which may be the reason that custom permissions are overlooked by the security community.

To the best of our knowledge, the study of Tuncay *et al.* [11] is the only work focusing on the security of custom permissions. They manually discovered two privilege escalation attacks that exploit the permission upgrade and naming convention flaws, respectively. Currently, according to the Android Security Bulletins, their discovered vulnerabilities have been fixed. Unfortunately, we find that, though both attacks have been blocked, custom permission based attacks can still be achieved with alternative execution paths bypassing the fix (more details are given in Section 3). This preliminary investigation motivates us to explore whether the design of Android custom permissions still has other flaws and how to find these flaws automatically.

Our Work. In this work, we systematically evaluate the design and implementation of Android custom permissions. Notably, we explored the design philosophy of custom permissions and measured their usage status based on a large-scale APK dataset. We also built an automatic lightweight fuzzing tool called `CuPerFuzzer+` to discover

- *Rui Li, Wenrui Diao, Shishuai Yang, Shuang Li, and Shanqing Guo are with the School of Cyber Science and Technology, Shandong University, Qingdao, Shandong 266237, China, and also with the Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, Shandong 266237, China. E-mail: {leiry, shishuai, lishuang1024}@mail.sdu.edu.cn, {diaowenrui, guoshanqing}@sdu.edu.cn.*
- *Zhou Li is with the University of California, Irvine, CA 92697 USA. E-mail: zhou.li@uci.edu.*

Manuscript received 5 July 2021; revised 3 October 2021; accepted 4 October 2021. Date of publication 14 October 2021; date of current version 14 November 2022.
This work was partially supported by the National Natural Science Foundation of China under Grants 61902148 and 91546203, Major Scientific and Technological Innovation Projects of Shandong Province, China under Grants 2018CXGC0708 and 2019JZZY010132, and the Qilu Young Scholar Program of Shandong University.
(Corresponding author: Wenrui Diao.)
Recommended for acceptance by J. Sun.
Digital Object Identifier no. 10.1109/TSE.2021.3119980

custom permission related privilege escalation vulnerabilities. Different from the previous approaches of permission system modeling [12], [13], CuPerFuzzer+ treats the operations of the Android permission mechanism as a black-box and dynamically generates massive test cases for fuzzing. In other words, it does not rely on prior knowledge of the internal permission mechanism and avoids missing inconspicuous system components. After solving a series of technical challenges, CuPerFuzzer+ achieves fully automated seed generation, test case construction, parallel execution, and result verification. Running on four Google Pixel 2 phones equipped with Android 9/10, finally, CuPerFuzzer+ discovered 5,932 successful exploit cases after executing 119,418 fuzzing tests.

These effective cases were further converted to 47 critical paths, say the necessary operations for triggering a privilege escalation issue. Combined with the analysis of the source code of Android OS, we identified six severe design shortcomings[1] in the Android permission framework.

- *DS#1*: *Dangling custom permission*: causing granting apps nonexistent custom permissions.
- *DS#2*: *Inconsistent permission-group mapping*: causing obtaining incorrect permission-group members list.
- *DS#3*: *Custom permission elevating*: causing elevating custom permissions to `dangerous` system permissions.
- *DS#4*: *Inconsistent permission definition*: causing breaking the integrity of custom permission definitions.
- *DS#5*: *Dormant permission group*: causing displaying fake group information at runtime.
- *DS#6*: *Inconsistent permission type*: causing downgrading `signature` custom permissions to `dangerous` custom permissions.

A malicious app can exploit these design shortcomings to *obtain `dangerous` system permissions without user consent* or *get unauthorized access to the resource protected by the `signature` permission*. Besides, we discovered a location permission-specific issue – the grouping of the location permission is alterable. Therefore, the adversary can exploit a custom permission to break the isolation between the foreground and background location permissions. As showcases, we present six concrete attacks to demonstrate their fatal consequences. Attack demos are available at https://sites.google.com/view/custom-permission.

Responsible Disclosure. We reported our findings to the Android security team, and all discovered design shortcomings have been confirmed with positive severity ratings [14], as shown below:

- *DS#1*: *High severity*, assigned `CVE-2021-0307`.
- *DS#2*: *High severity*, assigned `CVE-2020-0418`.
- *DS#3*: *High severity*, assigned `CVE-2021-0306`.
- *DS#4*: *High severity*, assigned `CVE-2021-0317`.
- *DS#5*: *Moderate severity* with `AndroidID-176828496`.
- *DS#6*: *Low severity* with `AndroidID-155649020`.
- *Location permission-specific issue*: *Moderate severity* with `AndroidID-186531661`.

To mitigate the current security risks, we propose some immediate improvements and discuss general design guidelines to secure custom permissions in Android.

Contributions. The main contributions of this paper are:

- *Tool Design and Implementation.* We designed and implemented an automatic black-box fuzzing tool, CuPerFuzzer+[2], to discover custom permission related privilege escalation vulnerabilities in Android.
- *Real-world Experiments.* We deployed CuPerFuzzer + under real-world settings and conducted massive fuzzing analysis. In the end, it discovered 5,932 privilege escalation cases with 47 critical paths.
- *New Design Shortcomings.* We identified six severe design shortcomings lying in the Android permission framework. Malicious apps can exploit these flaws to get unauthorized access to platform resources.
- *Systematic Study.* We explored the design philosophy of custom permissions and measured their usage in the wild. After digging into the essence of discovered design flaws, we discussed the general design guidelines to secure Android custom permissions.

This work is an extension of our conference version appearing in the Proceedings of IEEE S&P 2021 [15]. In this extension, we improved the automatic tool by adding an extra permission-related operation, device reboot, to enhance the execution path coverage and adjusting the method of extracting critical paths to locate causes more accurately and quickly. Therefore, we updated the tool name from CuPerFuzzer to CuPerFuzzer+. In addition, we conducted new experiments and discovered two new design shortcomings and a new location permission-specific issue lying in the Android permission framework. These new findings also have been confirmed by the Android security team. For the new shortcomings, we gave their detailed analysis and the corresponding mitigation solutions. To better explain the newly discovered shortcomings, we complemented the necessary background about the runtime permission granting. Based on the new insights, we proposed a new general design guideline. Also, we updated our APK dataset and conducted a more accurate usage purpose analysis of the custom permissions, combining with the NLP techniques.

Roadmap. The rest of this paper is organized as follows. Section 2 provides the necessary background of Android custom permissions. Section 3 gives a motivation case and the threat model used in this paper. Section 4 introduces the detailed design of CuPerFuzzer+, and Section 5 presents the experiment results. The design flaws of custom permissions are analyzed in Section 6. In Section 7, we propose mitigation solutions and general design guidelines. In Section 8, we discuss some limitations of our work. Section 9 reviews related work, and Section 10 concludes this paper. Additional materials are provided in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3119980.

1. In the following sections, we use *DS#1* $\sim$ *DS#6* for short.

2. The source code of CuPerFuzzer+ is available at https://github.com/little-leiry/CuPerFuzzer.

(a) (b)

Fig. 1. Grant or deny a runtime custom permission to the app `test` in Android 10. (a) The custom permission is not in any group. (b) The custom permission is in the `PHONE` group.

2 Android Custom Permissions

In this section, we provide the necessary background of Android custom permissions and further discuss their usage in the wild based on a large-scale measurement.

2.1 Android Permission Mechanism

In Android, sensitive APIs and system resources are protected by the permission mechanism. To access them, apps must request the corresponding permissions in their manifest files and ask users to authorize. In Android 10 (API level 29), the permission control functionalities are mainly implemented in `PackageManager` [16] and `PermissionController` [17].

Permissions are divided into three main protection levels: `normal`, `signature`, and `dangerous`. The system grants apps `normal` and `signature` permissions at the install time, so these two kinds of permissions are also called *install-time permissions*. The difference is that a `normal` permission is granted to an app automatically, while a `signature` permission can only be used by the app signed by the same certificate as the app defining it [18]. On the other hand, users can choose to grant or deny `dangerous` permissions at runtime, so `dangerous` permissions are also called *runtime permissions*. When the app requests a runtime permission, the system shows a permission prompt to the user, as shown in Fig. 1. The contextual information can help the user understand why the app needs this permission and make a better decision about granting or denying it [19].

All `dangerous` permissions belong to permission groups. For example, both `READ_SMS` and `RECEIVE_SMS` belong to the `SMS` group. Also, `dangerous` permissions are granted on a group basis. If an app requests `dangerous` permissions belonging to the same permission group, once the user grants one, the others will be granted automatically without user confirmation. Note that any permission can be assigned to a permission group regardless of protection level [20].

From an internal view, to an app, the processes of granting and revoking a permission are essentially changing the corresponding granting status parameter, `mGranted` (boolean variable), maintained by `PermissionController` (runtime permissions) and `PermissionManagerService` (install-time permissions). `mGranted` is set as `True` to grant a permission and `False` to revoke a permission. The granting status of permissions is also recorded in `runtime-permissions.xml`[3] (runtime permissions) and `packages.xml`[4] (install-time permissions) for persistent storage.

3. Location: `/data/system/users/0/runtime-permissions.xml`

4. Location: `/data/system/packages.xml`

2.2 Custom Permissions

In essence, system permissions (also called platform permissions) are the permissions defined by system apps located in system folders (`/system/`), such as `framework-res.apk` (package name: `android`), to protect specific system resources. For instance, an app must have the `CALL_PHONE` permission to make a phone call. For third-party apps, they can define their own permissions as well, called *custom permissions*, to share their resources and capabilities with other apps [10].

As shown in Listing 1, a custom permission `com.test.cp` is defined in an app's manifest file using the `permission` element. The app must specify the permission name and protection level (default to `normal` if not specified). If the name is the same as a system permission or an existing custom permission, this custom permission definition will be ignored by the system.

Listing 1. Define and Request a Custom Permission

```
<!-- Define a custom permission -->
<permission
android:name="com.test.cp"
android:protectionLevel="normal"
android:description="@string/per_des"
android:permissionGroup="android.permission-group.
    PHONE"/>
<!-- Request a custom permission -->
<uses-permission android:name="com.test.cp"/>
<!-- Description of the custom permission -->
<string name="per_des">get the message push service
    </string>
```

App developers can use the `android:description` attribute to explain the purpose of the custom permission. For the `dangerous` (runtime) custom permission, the permission description will be presented in the runtime permission dialog, as shown in Fig. 1a. App developers can also assign a permission group to the custom permission optionally. The group can be a custom group defined by third-party apps or a system group defined by system apps (e.g., the `PHONE` group in Listing 2). If the `dangerous` (runtime) permission belongs to a permission group, the system will show the group's functionality to the user, as Fig. 1b shows.

Listing 2. The Definition of the `PHONE` Group in the Manifest File of the System App `Android`

```
<!-- Used for permissions that are associated with tele-
     phony features -->
<permission-group android:name="android.permission-
    group.PHONE"
android:icon="@drawable/perm_group_phone_calls"
android:label="@string/permgrouplab_phone"
android:description="@string/permgroupdesc_phone"
android:request="@string/permgrouprequest_phone"
android:priority="500" />
<!-- Description of the PHONE group -->
<string name="permgroupdesc_phone">make and man-
    age phone calls</string>
```

To use the custom permission, the app needs to request it through the `uses-permission` element in its manifest file.

TABLE 1
Protection Levels of Custom Permissions

Protection Level	Amount	Percentage
`normal`	19,411	13.61%
`dangerous`	644	0.45%
`signature`†	122,525	85.93%
`instant`‡	8	0.01%

†: *Include mixed levels:* `signature|privileged` *and* `signatureOrSystem`.
‡: *Only for instant apps [22].*

Design Philosophy. In most usage scenarios, Android does not intend to distinguish between system and custom permissions. The general permission management policies apply to both types of permissions, including protection levels, runtime permission control, and group management. This design unifies and simplifies the control of permissions.

The fundamental difference is that, system permissions are defined by the system (system apps), and custom permissions are defined by third-party apps. Actually, if the system needs to judge whether a permission is a system one, it will check whether its source package is a system app [21]. Also, system apps are pre-installed and cannot be modified or removed by users. Accordingly, their defined permissions are stable, including names, protection levels, grouping, and protected system components. Therefore, system permissions are treated as constant features of Android OS. On the other hand, users can install, uninstall, and update arbitrary third-party apps, making the usage of custom permissions more flexible. That is, it brings the possibilities of adding, removing, and updating permission definitions, though these permission-related operations are not only designed for custom permissions.

Since system permissions are used to protect essential platform resources, Android indeed designs some mechanisms to ensure custom permissions will not affect the scope of system permissions. For instance, system permissions cannot be occupied by third-party apps, say changing the permission owner. This guarantee is achieved through three conditions: (1) Android does not allow an app to define a permission with the same name as an existing permission. (2) If multiple apps define a permission with the same name, the app installed first is the owner of this permission. (3) System apps are installed before any third-party apps and first define a set of permissions to protect specific platform resources. It can be seen that Android does not distinguish the permission type in this course, reflecting the design philosophy of custom permissions, to a certain extent.

2.3 Usage Status in the Wild

To understand the status quo of using custom permissions, we conducted a large-scale measurement based on 178,493 APK files crawled from 16 popular app markets in February 2021[5], including APKPure, Anzhi, Baidu, Xiaomi, Huawei, Lenovo, 9Apps, 2265, DownloadPCAPK, F-Droid, Gfan, LapTopPCAPK, LePlay, Leyou, PC6, and Uptodown. Specifically, we focus on the following two research questions:

1) *How many apps use custom permissions?*
2) *What are the purposes of using custom permissions?*

To answer these questions, we developed a light-weight tool based on Androguard [24] to obtain custom permission related information in apps for further processing.

TABLE 2
Permission Groups of Custom Permissions

Group Type	Amount	Percentage
System Group	1,757	58.14%
Custom Group	1,265	41.86%

To *Question-1*, our results show that 66,639 apps (around 37.3%) declare a total of 142,588 custom permissions. We can find that the use of custom permissions is not unusual. On the aspect of protection levels, about 86% of these permissions are `signature`, as summarized in Table 1. The reason for such a high percentage may be that a series of apps only want to share some sensitive resources with the apps developed by the same company (signed by the same certificate). On the other hand, `normal` permissions account for 13.61%, and `dangerous` permissions account for only 0.45%. The use of `instant` permissions is extremely rare ($< 0.01\%$), which are only used for instant apps [22].

Besides, 3,022 custom permissions (around 2.1% of the total) are assigned to permission groups, see Table 2. Among them, system permission groups are used more frequently than custom permission groups. Using a system group can simplify the permission UI shown to the user, which is recommended by Google [10].

To *Question-2*, for each custom permission, we crawled its name, label, description, and the information of the components it protects for analysis. Based on the extracted data, we designed an approach utilizing the natural language processing (NLP) techniques to analyze the purposes of using custom permissions [25], [26]. As illustrated in Fig. 2, the main steps are as follows:

1) *Data Pre-processing*. First, we clean the collected data to filter the invalid information that is not helpful for the purpose analysis, such as package names, general words (e.g., the "permission" word in the permission name), prepositions, and punctuation.
2) *Word Segmentation*. Next, we segment the pre-processed data by word for the subsequent vectorization. Note that our dataset contains Chinese and English. English is composed of single words and does not need word segmentation. For Chinese, we use

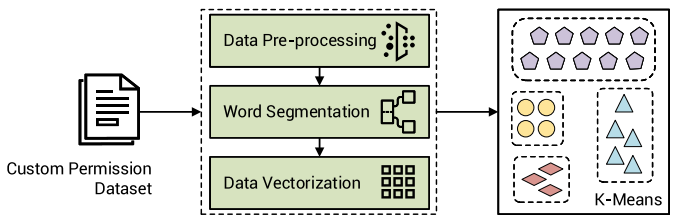


Fig. 2. Analysis flow of custom permission usage purposes.

5. We developed a crawler based on Scrapy (a framework for extracting the data from websites [23]) to traverse the pages of app market websites to obtain the download links of APK files.

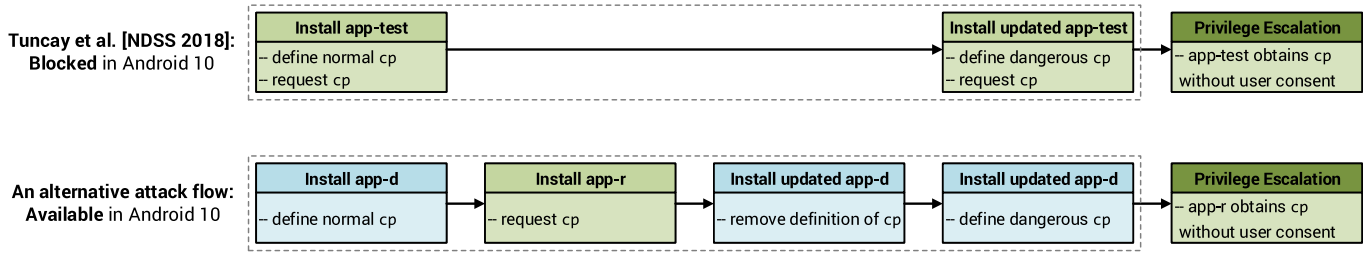


Fig. 3. An alternative attack flow achieving privilege escalation.

`jieba` [27], a Python Chinese word segmentation module, to process.

3) *Data Vectorization.* Then, we vectorize the data based on the word segmentation results utilizing the `Bag-of-Words` model. In order to achieve a better clustering effect, we convert the obtained vectors into a `TF-IDF` matrix [28].
4) *Data Clustering.* Finally, we use the `K-Means` algorithm to cluster the data. This algorithm converges quickly and is quite suitable for clustering large amounts of data. The initial value of `K` (the number of clusters) is determined by a rough classification[6], and we gradually adjust it to obtain the optimal clustering results.

We manually analyzed each obtained custom permission cluster's usage purpose based on the top 3 words appearing most frequently in it. The final classification results are plotted in Fig. 4, containing the following 8 categories:

- *C1: Use message push services.* More than 80% of apps in our dataset declare such kinds of permissions to using the corresponding message push services, such as `C2D_MESSAGE` offered by Google, `PROCESS_PUSH_MSG` offered by Huawei, `MIPUSH_RECEIVE` offered by Xiaomi, and `JPUSH_MESSAGE` offered by the JPush platform [29].
- *C2: Use location and map services.* For example, 590 apps in our dataset declare the `BAIDU_LOCATION_SERVICE` permission to obtain the location service provided by Baidu.
- *C3: Use instant message services.* For example, 2,366 apps in our dataset declare the `RECEIVE_MSG` permission to obtain the instant message service provided by the Ronglian cloud communication platform [30].
- *C4: Integrate third-party open-source frameworks.* For example, 981 apps in our dataset declare the `andPermission.bridge` permission and integrate `AndPermission`, a third-party open-source framework for permission management [31].
- *C5: Restrict the access to apps' shared data.* For example, the app Blue Mail (`me.bluemail.mail`) defines the `READ_MESSAGES`[7] permission to control the access to the mails it manages.
- *C6: Restrict the access to apps' download managers.* For example, only the apps with the `ACCESS_DOWNLOAD_MANAGER`[8] permission can access the download manager (and the corresponding downloaded data) of the app APUS Browser (`com.apusapps.browser`).
- *C7: Control the communication between apps.* For example, only the apps with the `INTERNAL_BROADCAST`[9] permission can send a broadcast to the broadcast receivers of the app Cisco Webex Meetings (`com.cisco.webex.meetings`).
- *C8: Others.* There are three cases in this category: (1) The number of some similar custom permissions is too tiny to be clustered. (2) Due to insufficient related information, the usage purposes of some custom permission clusters cannot be determined. (3) The declarations of 6,552 custom permissions are invalid because these custom permissions have the same names as the system permissions.

3 Motivation and Threat Model

In this section, we discuss the motivation case of our work and give the threat model.

3.1 Motivation Case

The security of Android custom permissions was not thoroughly studied in previous research. The reason may be that the corresponding security threats were regarded as limited, irrelevant to sensitive system resources and user data. As the only literature focusing on custom permissions, the study of Tuncay *et al.* [11] found that custom permissions were insufficiently isolated, and there is no enforcing name convention for custom permissions in Android. They also presented two privilege escalation attacks to access unauthorized resources.

As shown in upper Fig. 3, one attack case is that the adversary creates an app `app-test` that defines and requests a `normal` custom permission `cp`, and the user installs this app. Then, the protection level of `cp` is changed to `dangerous`, and the user installs this updated `app-test` again. Finally, `app-test` obtains `dangerous cp` without user consent, that is, privilege escalation. This attack can be further extended to obtain `dangerous` system permissions.

6. Specifically, we first randomly select a permission name and keep its keyword by filtering invalid information, such as the package name. Then, we count the number of permission names containing this keyword. If the number is larger than 1000, we classify these custom permissions into a category.

7. Full name: `me.bluemail.mail.permission.READ_MESSAGES`

8. Full name: `com.apusapps.permission.ACCESS_DOWNLOAD_MANAGER`

9. Full name: `com.cisco.webex.permission.INTERNAL_BROADCAST`

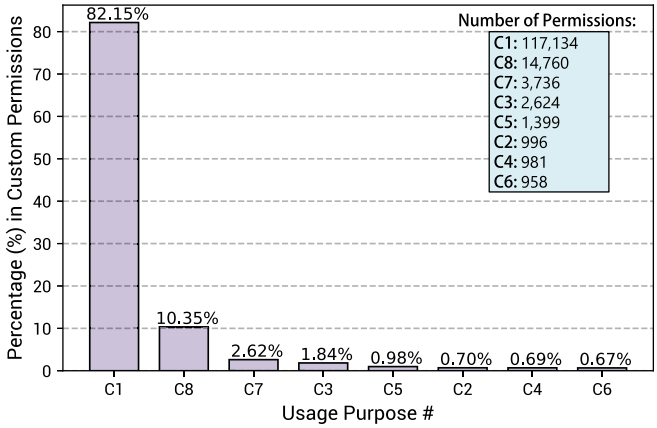


Fig. 4. Statistics on custom permission usage purposes.

Our Findings. According to the Android Security Bulletins and the corresponding source code change logs [32], the above attack has been fixed in Android 10. Google's fix prevents the permission protection level changing operation – from `normal` or `signature` to `dangerous`.

However, we find that, though this attack indeed has been blocked by Google, another app execution path still can achieve the same consequence, which bypasses the fix. As illustrated in lower Fig. 3, the adversary creates two apps, `app-d` and `app-r`. `app-d` defines a `normal` custom permission `cp`, and `app-r` requests `cp`. Also, there are two updated versions of `app-d`, say `app-d-1` and `app-d-2`. To be specific, `app-d-1` removes the definition of `cp`, and `app-d-2` re-defines `cp` with changing the protection level to `dangerous`. The user executes the following sequence: install `app-d`, install `app-r`, install `app-d-1`, and install `app-d-2`. Finally, `app-r` obtains `cp` and achieves the privilege escalation.

Our further investigation shows this newly discovered attack derives from a design shortcoming lying in the Android permission framework, that is, *DS#1 – dangling custom permission* (see Section 6.1).

Insight. This preliminary exploration motivates us to think about how to check the security of the complicated custom permission mechanism effectively. The previously reported two attack cases [11] may be only the tip of the iceberg, and an automatic analysis tool is needed. Besides, our ultimate target is supposed to be identifying design shortcomings lying in the permission framework, not just discovering successful attack cases that can achieve privilege escalation.

3.2 Automatic Analysis

On the high level, there exist two ways to conduct automatic analysis for custom permissions: static analysis (e.g., analyzing the source code of Android OS to find design flaws) and dynamic analysis (e.g., executing multitudinous test cases to trigger unexpected behaviors). In the end, we decided to adopt the strategy of dynamic analysis for two main reasons: (1) The internal implementation of the permission mechanism is quite complicated. (2) Static analysis usually requires prior knowledge to construct targeted models for matching [12], [13], [33].

Also, inspired by the motivation case, the analysis process could be abstracted as finding specific app execution sequences that can trigger privilege escalation issues. The internal operations of the permission mechanism could be treated as a black-box accordingly. Following this high-level idea, we designed an automatic fuzzing tool – `CuPerFuzzer+` (see Section 4).

3.3 Threat Model

In our study, we consider a general local exploit scenario. That is, the adversary can distribute malicious apps to app markets. The user may download and install some malicious apps on her Android phone. Note that this user understands the security risks of sensitive permissions and is cautious about granting permissions to apps. To conduct malicious actions, malicious apps try to exploit the flaws of custom permissions to access unauthorized platform resources, such as obtaining `dangerous` system permissions without user consent.

4 Design of CuPerFuzzer+

In this section, we introduce the detailed design of our automatic analysis tool – `CuPerFuzzer+`. It treats the internal operations of the Android permission framework as a black-box and tries to trigger privilege escalation issues by executing massive test cases. As discussed in Section 3.2, each test case is essentially an *app execution sequence composed of various test apps and permission-related operations*. Besides, we also consider how to check the execution results and identify critical paths to facilitate locating the causes efficiently.

As illustrated in Fig. 5, on a high level, `CuPerFuzzer+` contains five main steps, as follows:

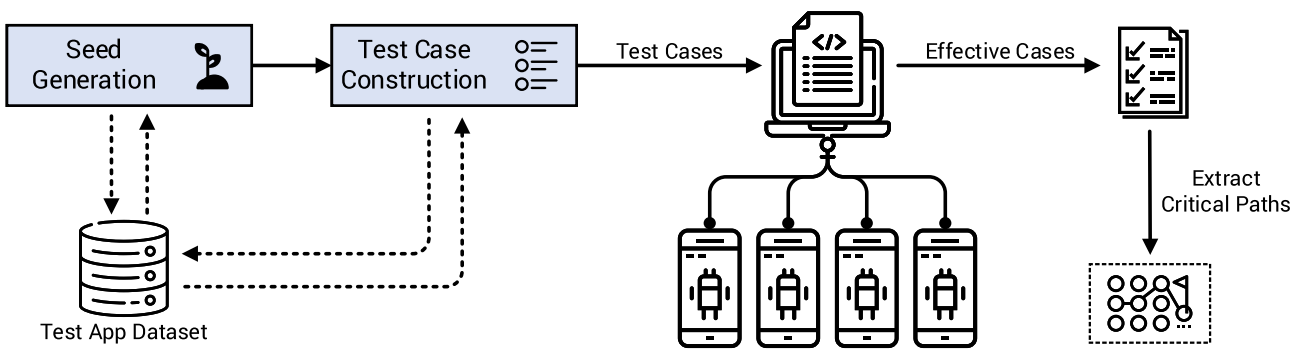


Fig. 5. Overview of `CuPerFuzzer+`.

1) *Seed Generation.* As the first step, CuPerFuzzer+ needs to generate a test app as the seed to activate the subsequent fuzzing process.
2) *Test Case Construction.* Next, CuPerFuzzer+ dynamically constructs plenty of complete cases for testing.
3) *Test Case Execution.* Then, CuPerFuzzer+ executes test cases in a controlled environment in parallel.
4) *Effective Case Checking.* After executing a test case, CuPerFuzzer+ checks whether a privilege escalation issue has been triggered.
5) *Critical Path Extraction.* Finally, among all discovered effective cases, CuPerFuzzer+ filters duplicated cases and identifies critical paths.

4.1 Seed Generation

As mentioned in our threat model (see Section 3.3), we want to discover local privilege escalation cases. Therefore, a successful attack will be achieved by malicious apps installed on the phone, and our fuzzing test will start with installing a test app, say the seed app.

Seed Variables. This seed app defines and requests a custom permission. Also, it requests all `dangerous` and `signature` system permissions[10]. Three attributes of this custom permission definition are variable, including:

- *Permission name*: based on a pre-defined list but cannot be the same as a system permission.
- *Protection level*: `normal`, `dangerous`, or `signature`.
- *Group*: a certain system group or not set.

Note that we prepare a pre-defined permission name list instead of random generating because some unusual names may trigger unexpected behaviors, such as containing special characters and starting with the general system permission prefix `android.permission`. Therefore, they need to be constructed ingeniously.

Seed Generation Modes. The key components of the seed app can be split into two apps, say one app defining the custom permission and the other app requesting permissions. Also, they are signed by different certificates. Therefore, we have two seed generation modes: *single-app mode* and *dual-app mode*. Different modes will further affect the subsequent step of test case construction.

Seed Generation. As a result, when generating a new seed, CuPerFuzzer+ needs to determine the seed generation mode and custom permission definition. In practice, to avoid the time cost of real-time app construction, CuPerFuzzer+ could construct plenty of test apps and store them in a dataset in advance. When running tests, CuPerFuzzer+ randomly selects an app from the prepared dataset as the seed and quickly activates the fuzzing process.

4.2 Test Case Construction

Next, CuPerFuzzer+ constructs a complete test case. As illustrated in Fig. 6, it is an app execution sequence consisting of multiple test apps and operations that may affect the granting of requested permissions.

10. Note that it does not mean that these permissions have been granted to the seed app. In fact, the granting of `dangerous` system permissions needs the user's consent, and the `signature` ones cannot be granted to the apps signed by different certificates from system apps.

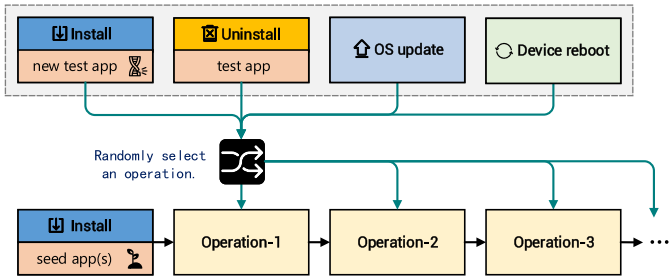


Fig. 6. Test case (app execution sequence) construction.

Operation Selection. After reviewing the Android technical documents and source code [21], we confirm five operations meeting the requirement: app installation, app uninstallation, app update, OS update, and device reboot. All of them can trigger the system to refresh the granting status of existing permissions.

- When installing a new app, new custom permission definitions may be added to the system.
- When uninstalling an app, existing custom permission definitions may be removed.
- When updating an app, existing custom permission definitions may be updated or removed.
- During major OS updates, new system permission definitions may be added to the system, and existing system permission definitions may be updated or removed.
- During the device reboot, the system will recover the granting status of requested permissions in apps before the device shutdown.

Note that considering that updating an app is installing (different versions of) this app multiple times, we do not need to indicate the app update operation in test cases.

Test App Mutation. In the app installation operation, the test app to install is the mutated version of the previously installed test app, say the same package name and app signature. It changes some attributes (group and protection level) of the previously defined custom permission or removes this permission definition directly. For example, it changes the protection level from `normal` to `dangerous` and puts the custom permission into the `PHONE` group. The permission name cannot be changed. Otherwise, it will define a new permission. Also, in the dual-app mode, the app defining the custom permission is treated as the test app because we do not change the permission requests in the whole execution sequence.

Test Case Construction. When CuPerFuzzer+ constructs a test case, it randomly selects an undetermined number of operations from the {*app installation*, *app uninstallation*, *OS update*, *device reboot*} and combines them with the seed app to generate a specific app execution sequence.

Also, to generate a meaningful test case, we set the following restrictions:

- The first operation must be seed app installation because the fuzzing execution environment (physical phone) will be reset before every test.
- Before executing an app uninstallation operation, there must exist a test app for uninstalling. Uninstalling a non-existing app is meaningless.

- In the single-app mode, the test app must exist on the phone after executing the last operation. In other words, the permission requests must exist at the end of case execution.
- Device reboot operations cannot be continuous. There is no change in the system between two continuous device reboot operations.
- The OS update operation only can be executed once. Our test focuses on the latest version of Android OS and thus only considers updating the OS from the previous version to the current version.

Besides, we can control the fuzzing testing scale by limiting the number of test cases deriving from one seed app.

4.3 Test Case Execution

In this step, CuPerFuzzer+ dynamically executes the operations in test cases in order. All operations are conducted on physical devices equipped with AOSP Android OS, e.g., Google Pixel series phones. The reasons for not using Android emulators (virtual devices) include:

- Emulators do not support the OS update operation.
- There may exist undocumented modifications in emulators' images to adapt to the underlying hardware.

Parallel Case Execution. To facilitate the automatic execution of test cases on physical devices, a computer is used as the controller to send test cases and monitor the execution status. The communication between them is supported by adb (Android Debug Bridge), a versatile command-line tool [34]. Also, CuPerFuzzer+ supports parallel execution by increasing the number of test devices. It can assign test cases to different devices to achieve the load balance during the testing.

As mentioned in Section 4.2, there are four kinds of operations in a test case. Among them, the app installation, app uninstallation, and device reboot operations can be executed through the adb install, adb uninstall, and adb reboot commands directly. To perform the OS update operation, we combine the capabilities of adb and fastboot to automate this process, that is, rebooting the device into the fastboot mode and flashing a new OS image (without wiping data) [35].

Environment Reset. After completing a test case execution, the test environment will be reset to the factory default status. It should be noted that, in general, the user needs to manually authorize to allow the computer to interact with a device through adb. However, once the device is reset (through a factory reset or OS downgrade), the previous authorization status will be erased, breaking the adb communication. To solve this issue, we can modify the source code of Android OS and compile a special version of the target OS image for test devices, which skips the authorization step and keeps adb always open. More specifically, in the build.prop file of the image, if ro.adb.secure is set to 0, the device will trust the connected computer by default without user authorization. Also, the image can be built with the userdebug type option to support the always-open adb debugging [36].

4.4 Effective Case Checking

For each completed test case, CuPerFuzzer+ needs to check whether it is an effective case achieving privilege escalation. An effective case can be determined by checking the granting status of the requested permissions in the test app (or the app requesting permissions in the dual-app mode). Expressly, we set the following two rules:

- *Rule 1*: The test app (or the app requesting permissions in the dual-app mode) has been granted a dangerous permission without user consent.
- *Rule 2*: The test app (or the app requesting permissions in the dual-app mode) has been granted a signature permission, but this test app and the app defining this permission are signed by different certificates.

Note that, in the whole process of test case execution, CuPerFuzzer+ does not grant any dangerous permission to the test app (or the app requesting permissions in the dual-app mode) through simulating user interactions.

To automate this checking, CuPerFuzzer+ uses adb to obtain the permission granting list of the test app (or the app requesting permissions in the dual-app mode) and extracts the granted permissions. If there is any granted dangerous or signature permission matching the above rules, this test case is effective and will be recorded for further analysis.

4.5 Critical Path Extraction

After obtaining all effective test cases, CuPerFuzzer+ extracts the critical paths to assist the cause identifications. A critical path is defined as the least necessary operations to trigger a privilege escalation issue. An effective test case contains multiple operations, and some operations are not related to the final privilege escalation. Also, many similar effective cases may be discovered in the test and contain the same critical path. CuPerFuzzer+ extracts the critical paths through the following steps:

1) *Classify Test Cases.* Discovered effective cases are classified into different categories according to their execution results. The test cases in the same category should lead to the same permission granting status.
2) *Find Critical Path.* In each category, CuPerFuzzer+ finds the test cases with the least operations, called *candidate cases*. To a randomly selected candidate case, CuPerFuzzer+ deletes its first operation and executes this pruned case. Also, this case should be meaningful, meeting the restrictions mentioned in Section 4.2. If the same execution result occurs, CuPerFuzzer+ adds this pruned case to this category and repeats this step. If it is different, CuPerFuzzer+ deletes the second operation, and so on until the last operation. If the execution results of all pruned cases are different from the candidate case's execution result, the operation sequence of this candidate case is a critical path.
3) *Delete Duplicate Cases.* In the same category, if an effective case's operation sequence *relatively contains*[11] the extracted critical path, this case will be

11. The effective case's operation sequence should contain the same operations as the critical path, and the operations' relative positions are the same. For example, the operation sequence {Installation → Device-reboot → Uninstallation → Installation} relatively contains the critical path {Installation → Uninstallation → Installation}.

deleted. Note that, in this matching process, we do not require that the apps used in the installation operations are the same.

CuPerFuzzer+ repeats Step 2 and Step 3 until all effective cases have been deleted.

Finally, based on the extracted critical paths, we try to find out the root causes of the discovered effective cases by analyzing the source code of Android OS.

5 Implementation and Experiment Results

In this section, we present the prototype implementation of CuPerFuzzer+ and summarize the experiment results.

5.1 Prototype Implementation

We implemented a full-feature prototype of CuPerFuzzer + with around 2,900 lines of Python code. Besides, in order to make our framework fully automated, we integrated several tools into it. For example, as mentioned in Section 4.3, adb and fastboot are used for device control and case execution.

For test app generation, Apktool [37] and jarsigner [38] are integrated. Since our fuzzer needs lots of test apps, it is impractical to generate them manually. In our implementation, we first use Android Studio to build a signed APK file that declares a custom permission. Then, CuPerFuzzer+ decodes this APK file using Apktool to obtain its manifest file. When generating a new app declaring a new custom permission, CuPerFuzzer+ replaces the old permission definition with the new one (in the manifest) and then repackages the decoded resources back to an APK file using Apktool as well. Finally, CuPerFuzzer+ uses jarsigner to sign the APK file, and a new signed APK file is built. Therefore, the whole process can be completed automatically.

5.2 Experiment Setup

Hardware Setup. In our experiments, we deployed a computer (Windows 10, 16G RAM, Intel Core i7) as the controller and four Google Pixel 2 phones as the case execution devices. The controller can assign test cases to different phones for parallel execution.

Android OS. Our experiments focused on the custom permission security issues on the latest version of Android OS, which is Android 10. Following the approach described in Section 4.3, we built two versions of Android OS images for Pixel 2 based on the source code of AOSP Android 9 (PQ3A.190801.002) and 10 (QQ3A.200705.002). Note that we only modified the adb connection and screen locking related parameters. CuPerFuzzer+ executes a test case containing the OS update operation on the devices equipped with Android 9 and flashes the Android 10 image (without wiping data) to achieve the OS update. Other test cases are executed on the devices equipped with Android 10.

Test Case Optimization. Since the amount of generated test cases can be infinite in theory and dynamic execution is time-consuming, we set some optimization measures to control the experiment scale and improve the vulnerability discovery efficiency.

Operations. If a test case contains many operations, it is too complex to be exploited in practice. Therefore, we empirically limited that a test case only can have up to five operations (without counting the operation of seed app installation).

Seed apps. When generating a seed app, the name of the defined custom permission is a variable and cannot be the same as a system permission. In order to follow this rule, we extracted all declared system permissions[12] from Android 9 and Android 10. The results show that there are 88 system permissions (3 for normal, 3 for dangerous, and 82 for signature) existing in Android 10 but not in Android 9, as listed in Appendix B, available in the online supplemental material. In Android 9, the permission with any of these 88 system permission names will be treated as a custom permission. Therefore, we randomly selected one permission name from the new dangerous system permissions and the new signature system permissions, respectively, and constructed the following pre-defined permission name list for seed apps to handle this special situation.

- android.permission.ACTIVITY_RECOGNITION (new dangerous system permission in Android 10)
- android.permission.MANAGE_APPOPS (new signature system permission in Android 10)
- com.test.cp (a general custom permission name)

Note that the seed apps with the first two permission names are only used to construct the test cases containing the OS update operation. The last permission name is used to construct all kinds of test cases. Also, we do not use the name of a new normal system permission added in Android 10 because normal permissions will be granted automatically.

Since we have 2 seed generation modes, 3 available permission names, 3 protection levels, and 12 system permission groups[13] (as listed in Appendix C, available in the online supplemental material), the combinations of custom permission attributes could be calculated as Combinations = $2\times3\times3\times13$ (12 groups and no group) = 234. Therefore, 234 kinds of seed apps can be selected for our experiments in total.

Test case execution. A seed app can generate lots of test cases. To balance the coverage of test cases from different seeds, we used the following case execution method. CuPerFuzzer+ randomly selects a seed app and executes a test case generated from it. This process is repeated until all test cases have been executed or the controller interrupts the testing.

5.3 Result Summary

In our experiments, CuPerFuzzer+ executed 119,418 test cases on four Pixel 2 phones in 1256.2 hours (around 52.3 days) until we stopped it.

Efficiency. The average execution time is 151.5s per test case in the experiments. In Table 3, we list the average time

12. Obtained by running adb shell pm list permissions -f -g.
13. Obtained by running adb shell pm list permission-groups.

TABLE 3
Average Execution Time of Operations

Operation Type	Operation	Time Cost (s)
Case execution	App installation	0.9
	App uninstallation	0.3
	OS update	97.4
	Device reboot	30.1
Environment reset	Factory reset	58.3
	OS downgrade	123.9

cost of operation execution and environment reset in an ideal environment (i.e., the test devices are well operated, and every operation is executed successfully). The time cost of an OS update or downgrade operation includes the cost of image flashing and device reboot.

Results. Finally, CuPerFuzzer+ discovered 5,932 effective test cases triggering privilege escalation issues. All effective cases were matched by the first checking rule defined in Section 4.4, say *obtaining* `dangerous` *permissions without user consent*. To the second rule (*obtaining* `signature`*permissions*), through analyzing the source code of Android OS, we found that there is a checking process before granting a `signature` permission, which cannot be bypassed. This checking ensures that the app requesting a `signature` permission is signed by the same certificate as the app defining this permission.

As listed in Table 4, CuPerFuzzer+ further extracted 42 critical paths (PathNo.$1-41$) from these discovered effective cases. After manually checking and confirming, we identified 4 other critical paths (PathNo.$42-47$). In this table, we can find that if the critical path is very simple, many cases may contain this path. For example, up to 3,843 effective cases are derived from PathNo.4, a two-operation path (`Installation` $\rightarrow$ `OS-update`). Below we show some interesting findings:

- The test cases based on PathNo.1 can obtain the `ACCESS_BACKGROUND_LOCATION` permission. While, in the similar PathNo.$2,3$, this permission is missing.
- As mentioned in Section 3.1, the permission protection level changing operation has been blocked by Google. However, in PathNo.$5-15,29-39$, an additional OS update or device reboot operation reactivates such a privilege escalation attack.
- In PathNo.$16,28,40$, the `UNDEFINED` group is an undocumented system permission group but can be listed by `adb shell pm`. It triggers 30 `dangerous` system permissions (in different groups) to be obtained.
- In PathNo.$44-47$, the custom permission becomes a `signature` permission after executing the last operation. However, this permission is granted to the app as a `dangerous` one.

We manually analyzed the extracted 47 critical paths and reviewed the corresponding source code of Android OS. Finally, we identified six fatal design shortcomings lying in the Android permission framework, as labeled in the last column of Table 4. In the following sections, we will discuss these shortcomings and their improvements in detail.

Remarks. To support the additional control that users have over an app's access to location information, Android 10 introduces the `ACCESS_BACKGROUND_LOCATION` permission [39]. The location permissions have been split into two categories: *foreground location permission* (the `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` permissions) and *background location permission* (the `ACCESS_BACKGROUND_LOCATION` permission). If the app running on Android 9 requests both kinds of location permissions at the same time, and it has got the foreground location permission, it will be granted the background location permission automatically after the OS being updated to Android 10. Therefore, the test cases based on PathNo.1 can obtain all location permissions. On the other hand, if the app runs on Android 10, even if this app has got the granted foreground location permission, it still needs the user's confirmation of the background location permission granting request. Thus, the `ACCESS_BACKGROUND_LOCATION` permission is missing in Path No. $2,3$.

That is to say, in Android 10, there is isolation between the foreground and background location permissions when requesting them simultaneously. However, after analyzing the source code of Android OS, we found that this isolation can be broken due to the grouping of the location permission being alterable, as detailed in Appendix A, available in the online supplemental material. This location permission-specific issue has been confirmed by the Android security team with rating Moderate severity (`AndroidID-186531661`). Since this issue only involves the location permission, we will not discuss it as a design shortcoming.

6 Design Shortcomings and Attacks

In this section, we analyze the discovered design shortcomings in-depth and demonstrate the corresponding exploit cases. Following the responsible disclosure policy, we reported our findings to the Android security team, and all of them have been confirmed. The corresponding fixes will be released in the upcoming Android Security Bulletins. Also, attack demos can be found at https://sites.google.com/view/custom-permission.

6.1 DS#1: Dangling Custom Permission

As illustrated in Fig. 7, when an app is uninstalled or updated, `PackageManagerService` (PMS for short) will refresh the registration and granting status of all permissions. During this process, if a `dangerous` (runtime) custom permission's definition is removed, the system will also revoke its grants from apps. However, we find that:

> *DS#1: If the removed custom permission is an install-time permission, the corresponding permission granting status of apps will be kept, causing dangling permission.*

It means that, under this situation, an app has been granted a `normal` or `signature` custom permission, but there is no definition of this permission in the system. Therefore, if another app re-defines this permission with different attributes, it may trigger privilege escalation.

TABLE 4
Extracted Critical Paths in the Experiments

No.	Effective Cases	Seed Mode	Critical Path[†]	Privilege Escalation (Granted Permissions)	Flaw
1	24	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, LOCATION] → OS-update	com.test.cp (dangerous) ACCESS_BACKGROUND_ LOCATION ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION	DS#4
2	81	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, LOCATION] → Device-reboot	com.test.cp (dangerous) ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION	DS#4
3	29	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, LOCATION]	com.test.cp (dangerous) ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION	DS#1
4	3,843	single-app dual-app	Installation [ACTIVITY_RECOGNITION, normal, NULL] → OS-update	ACTIVITY_RECOGNITION	DS#3
5	29	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → OS-update	com.test.cp (dangerous)	DS#4
6-15[‡]	212	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, {Group}] → OS-update	com.test.cp (dangerous) dangerous system permissions in {Group}	DS#4
16	24	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, UNDEFINED] → OS-update	com.test.cp (dangerous) READ_CONTACTS ... (30 dangerous system permissions in total)	DS#2
17	156	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, NULL]	com.test.cp (dangerous)	DS#1
18-27[‡]	348	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, {Group}]	com.test.cp (dangerous) dangerous system permissions in {Group}	DS#1
28	35	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, UNDEFINED]	com.test.cp (dangerous) READ_CONTACTS ... (30 dangerous system permissions in total)	DS#2
29	197	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → Device-reboot	com.test.cp (dangerous)	DS#4
30-39[‡]	705	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, {Group}] → Device-reboot	com.test.cp (dangerous) dangerous system permissions in {Group}	DS#4
40	72	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, UNDEFINED] → Device-reboot	com.test.cp (dangerous) READ_CONTACTS ... (30 dangerous system permissions in total)	DS#2
41	9	dual-app	Installation [com.test.cp, normal, NULL] → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL]	com.test.cp (dangerous)	DS#1
42-43	6	dual-app	Installation [com.test.cp, normal, NULL] → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, UNDEFINED / ACTIVITY_RECOGNITION] → OS-update	com.test.cp (dangerous) dangerous system permissions in the UNDEFINED / ACTIVITY_ RECOGNITION group	DS#5
44	92	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, NULL] → Installation [com.test.cp, signature, NULL]	com.test.cp (dangerous)	DS#6
45	66	dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → Device-reboot → Installation [com.test.cp, signature, NULL]	com.test.cp (dangerous)	DS#6
46	2	dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → OS-update → Installation [com.test.cp, signature, NULL]	com.test.cp (dangerous)	DS#6
47	2	dual-app	Installation [com.test.cp, normal, NULL] → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL] → Installation [com.test.cp, signature, NULL]	com.test.cp (dangerous)	DS#6

[†]: *In the app* `Installation` *operation, the custom permission defined by the installed test app is put into the square brackets (* `[]`*), which is represented as* `[permission name, protection level, permission group]`*.* `NULL` *represents the corresponding attribute is not set.*
[‡]: *They are similar critical paths, and the only difference is the used system group.*

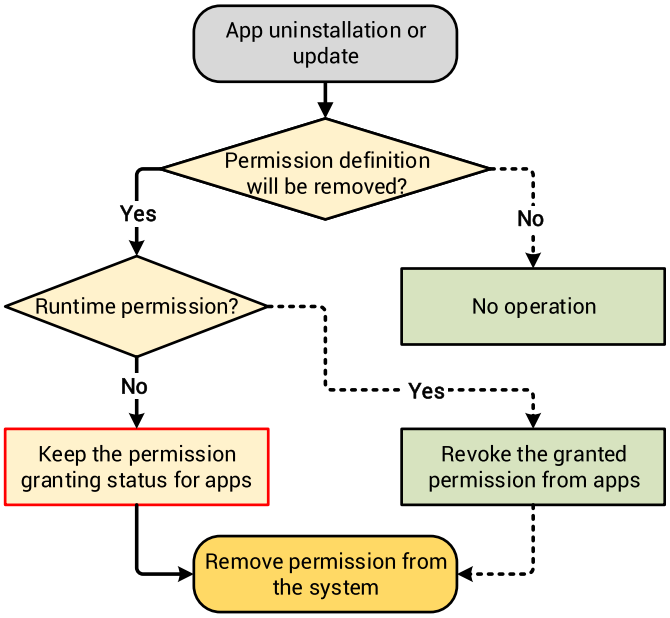


Fig. 7. Dangling custom permission.

Attack Case. The adversary creates and distributes two apps to app markets, `app-ds1-d` and `app-ds1-r` (their signing certificates can be the same or not). `app-ds1-d` defines a `normal` custom permission `com.test.cp`, and `app-ds1-r` requests `com.test.cp` and the `CALL_PHONE` permission (`dangerous` system permission belonging to the `PHONE` group). The adversary also prepares an updated version of `app-ds1-d` that declares the following permission:

Listing 3. Updated Custom Permission

```
< permission
android:name="com.test.cp"
android:protectionLevel="dangerous"
android:permissionGroup="android.permission-group.
    PHONE" > < /permission >
```

The user installs `app-ds1-d` and `app-ds1-r` on her phone. At this moment, `app-ds1-r` has been granted `normal` `com.test.cp`. Then, she is induced to execute the following operations: uninstall `app-ds1-d` and install the updated `app-ds1-d`. For example, a reasonable scenario is that `app-ds1-d` frequently crashes deliberately. Then it reminds the user to delete the current version and install a new version. When the user installs the updated `app-ds1-d`, PMS scans the package and adds the updated `com.test.cp` to the system. After that, PMS iterates over the existing apps to adjust the granting status of their requested permissions. Since `com.test.cp` has become a `dangerous` permission, `com.test.cp` will be re-granted to `app-ds1-r` as a `dangerous` permission. Further, the granting of `dangerous` permissions is group-based. Since both `CALL_PHONE` and `com.test.cp` are in the `PHONE` group, `app-ds1-r` obtains the `CALL_PHONE` permission without user consent.

Discussion. Through changing the `PHONE` group to other permission groups, the malicious app can obtain arbitrary `dangerous` system permissions.

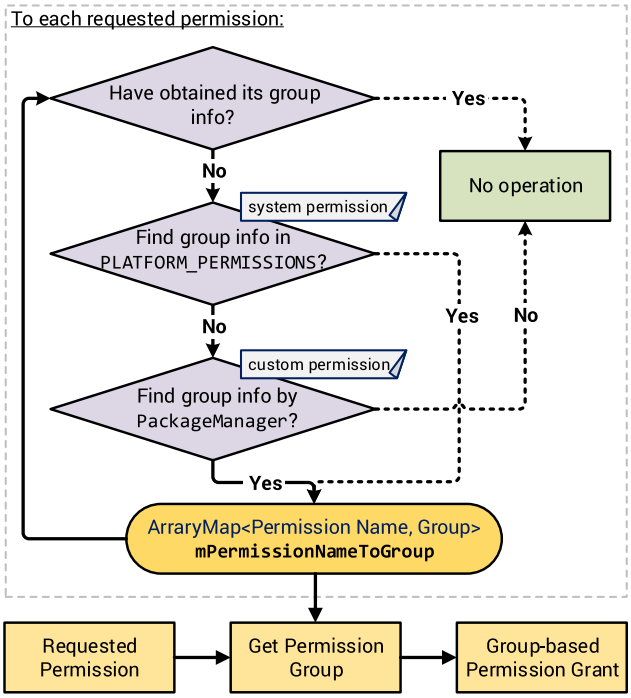


Fig. 8. Inconsistent permission-group mapping.

The root cause of the attack case described in Section 3.1 is also *DS#1*. It creates a dangling custom permission during the first app update. However, it cannot be extended to obtain system permissions through the group-based permission granting. The reason is that, when handling runtime permissions, their association with the permission groups cannot be changed (cannot remove a permission from a group and assign it to another group) [19].

Impact. *DS#1* and its exploits (as two individual attack cases in two reports) have been confirmed by Google. Both reports were rated as *High severity* (`AndroidID-155648771` and `AndroidID-165615162`), and a CVE ID has been assigned: `CVE-2021-0307`.

6.2 DS#2: Inconsistent Permission-Group Mapping

In Android, the grant of `dangerous` permissions is group-based. If an app has been granted a `dangerous` permission, it can obtain all the other permissions belonging to the same group without user interactions. Therefore, the correct < permission, group > mapping relationship is quite critical in this process.

As illustrated in Fig. 8, when Android OS processes a `dangerous` permission granting request, it will query the group (members) information of the requested permission through `ArrayMap mPermissionNameToGroup` [40]. Based on the obtained <permission, group > mapping information, the system can determine whether this permission can be granted to the app automatically, that is, whether one permission of the group has been granted to the app previously.

To facilitate this operation, the system needs to construct `mPermissionNameToGroup` in advance. To each requested permission, if its mapping relationship can be found in `mPermissionNameToGroup`, no operation is

needed. Otherwise, mPermissionNameToGroup will be updated with adding new data. However, we find that:

DS#2: System and custom permissions rely on different sources to obtain the < permission, group > mapping relationship, which may exist inconsistent definitions.

The system tries to obtain the group information of the requested permission by querying PLATFORM_PERMISSIONS and PackageManager. Since PLATFORM_PERMISSIONS is a hard-coded < system permission, system group > mapping array defined in PermissionController [41], custom permissions cannot be found in this mapping array. That is to say, if the requested permission is a custom permission, the system will invoke PackageManager to get the group information. Note that, PackageManager relies on the app's AndroidManifest.xml to construct such mapping data. Therefore, once there exist inconsistent definitions between PLATFORM_PERMISSIONS and AndroidManifest.xml, privilege escalation may occur.

We find that, in Android 10, there indeed exist such inconsistent definitions. Specifically, in the AndroidManifest.xml of the app android (the core manifest file of the system [42]), all dangerous system permissions are put into a special permission group, named android.permission-group.UNDEFINED. The adversary can exploit such inconsistency and the group-based permission granting to obtain all dangerous system permissions (apart from the ACCESS_BACKGROUND_LOCATION permission).

Attack Case. The adversary creates an app app-ds2 that requests the WRITE_EXTERNAL_STORAGE permission, a common permission for saving app data. The user installs app-ds2 and grants WRITE_EXTERNAL_STORAGE to app-ds2.

Then, the adversary creates an updated version of app-ds2, which defines and requests a dangerous custom permission com.test.cp. Also, the updated app-ds2 requests all dangerous system permissions (apart from the ACCESS_BACKGROUND_LOCATION permission), as shown below:

Listing 4. Updated Version of app-ds2

```
< permission
android:name="com.test.cp"
android:protectionLevel="dangerous"
android:permissionGroup="android.permission-group.
    UNDEFINED" / >
< uses-permission      android:name="android.permission.
    WRITE_EXTERNAL_STORAGE" / >
< uses-permission      android:name="android.permission.
    SEND_SMS" / >
< uses-permission      android:name="android.permission.
    CAMERA" / >
... < ! − −Omit lots of permission requests−− >
< uses-permission      android:name="android.permission.
     BODY_SENSORS" / >
< uses-permission android:name="com.test.cp" / >
```

Next, the user installs this updated version of app-ds2, and the system automatically grants it all dangerous system permissions (apart from the ACCESS_BACKGROUND_LOCATION permission) without user permitting.

As mentioned before (see Fig. 8), to each requested permission, the system will add its group member information to mPermissionNameToGroup. To system permissions (Line 6-10), the < permission, group > mapping looks like:

Listing 5. Mapping mPermissionNameToGroup

```
< WRITE_EXTERNAL_STORAGE, STORAGE >
< SEND_SMS, SMS >
< CAMERA, CAMERA >
...
< BODY_SENSORS, SENSORS >
```

When reaching the custom permission (Line 11), since it belongs to the UNDEFINED group, and this group contains all dangerous system permissions. The < permission, group > mapping is refreshed as:

Listing 6. Updated Mapping mPermissionNameToGroup

```
< WRITE_EXTERNAL_STORAGE, UNDEFINED >
< SEND_SMS, UNDEFINED >
< CAMERA, UNDEFINED >
...
< BODY_SENSORS, UNDEFINED >
```

Therefore, under this situation, if one dangerous permission (WRITE_EXTERNAL_STORAGE) has been granted, the other dangerous permissions will be granted without user permitting because they belong to the same permission group, that is, android.permission-group.UNDEFINED.

Discussion. Obviously, a hard-coded < system permission, system group > mapping table is more secure. However, Android allows app developers to put custom permissions into system groups, which forces the system to manage dynamic group information in the mix of different types of permissions.

According to the commit logs [43], [44], in the source code of Android OS, the UNDEFINED group was introduced as a dummy group to prevent apps from obtaining the grouping information (through PackageManager). The OS developers commented, "*the grouping was never meant to be authoritative, but this was not documented*."

Impact DS#2 and its exploit have been confirmed by Google with rating *High severity* (AndroidID-153879813), and a CVE ID has been assigned: CVE-2020-0418.

6.3 DS#3: Custom Permission Elevating

As illustrated in Fig. 9, during the Android OS initialization (device booting), PackageManagerService (PMS for short) will be constructed, which is used for managing all package-related operations, such as installation and uninstallation. Then, PMS reads packages.xml and runtime-permissions.xml to get the stored permission declaration information and grant states.

After that, PMS scans APKs located in system folders and then adds the parsed permissions to an internal structure. Note that if the current owner of a parsed permission is not

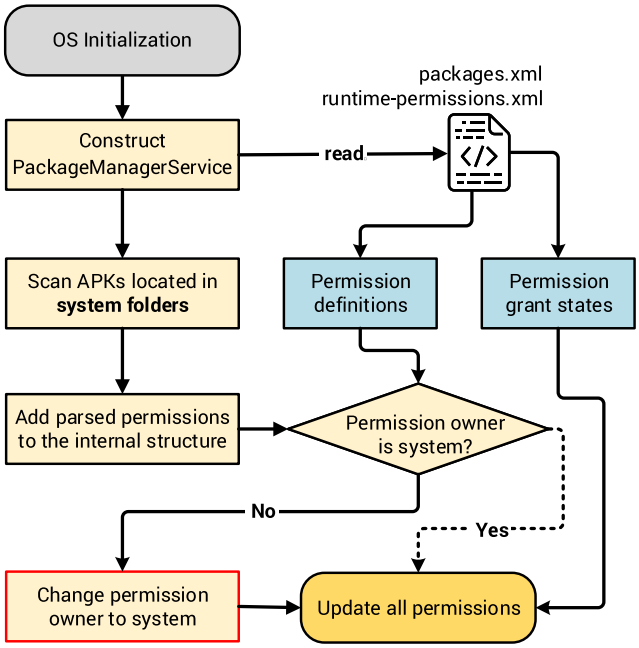


Fig. 9. Custom permission elevating.

the system, this permission will be overridden, changing the owner to the system. However, we find that:

> *DS#3: When Android OS overrides a custom permission, the granting status of this permission is not revoked, further resulting in permission elevating.*

That is to say, if an app has been granted a custom permission with the same name as a system permission, this granted custom permission will be elevated to system permission after permission overriding.

Attack Case. In general, an app cannot define a custom permission with the same name as an existing permission. However, if we consider the OS upgrading operation, this scenario will become possible. For instance, on an Android 9 device, the adversary creates an app `app-ds3`, which defines and requests a custom permission `ACTIVITY_RECOGNITION`, as follows:

Listing 7. Define and Request `ACTIVITY_RECOGNITION`

```
< permission
android:name="android.permission.
    ACTIVITY_RECOGNITION"
android:protectionLevel="normal" / >
< uses-permission      android:name="android.permission.
    ACTIVITY_RECOGNITION" / >
```

Note that the `ACTIVITY_RECOGNITION` permission is a new `dangerous` *system permission* introduced in Android 10. However, on devices running Android 9, `ACTIVITY_RECOGNITION` is only treated as a `normal` *custom permission*.

After the user installs `app-ds3`, she performs OTA OS update, and later the device reboots with running Android 10. After finishing OS initialization, `app-ds3` has been granted the `ACTIVITY_RECOGNITION` permission (`dangerous` system permission) automatically without user consent.

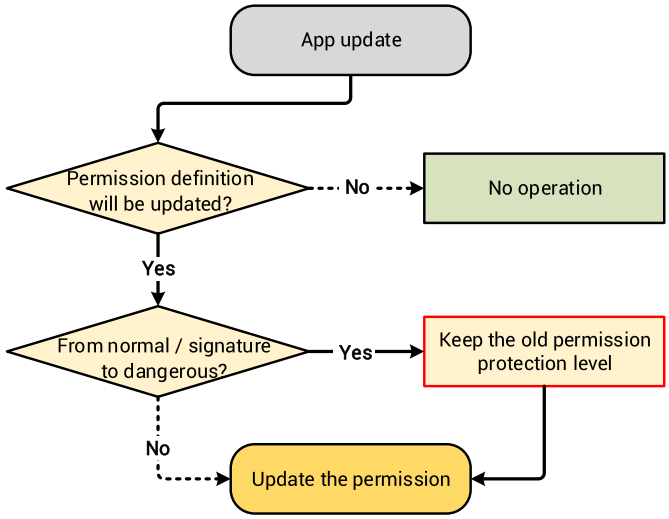


Fig. 10. Inconsistent permission definition.

Discussion. Our further investigation shows that *DS#3* was introduced when Google fixed the Pileup flaw discovered by Xing *et al.* [45]. An exploit scenario of Pileup is that, in Android 2.3, a third-party app defines a `normal` custom permission with the same name as a `signature` system permission, which was added in Android 4.0. After OS upgrading, this app becomes the owner of this new system permission, and the protection provided by this permission also becomes ineffective.

Google's fix to the Pileup flaw is that, during the OS initialization, the OS will override all permissions declared by the system, say taking the ownership [46]. However, in this process, the OS still keeps the previous granting status, which results in *DS#3*.

Impact. *DS#3* and its exploit have been confirmed by Google with rating *High severity* (`AndroidID-154505240`), and a CVE ID has been assigned: `CVE-2021-0306`.

6.4 DS#4: Inconsistent Permission Definition

In the process of the app update, as Fig. 10 shows, if a custom permission's protection level is changed from `normal` or `signature` to `dangerous`, the system will keep its old protection level. Such a design is to block the permission upgrade attack (see upper Fig. 3). However, we find that:

> *DS#4: At this moment, the permission definition held by the system is different from the permission definition provided by the owner app, say inconsistent permission definition.*

If there is any logic in refreshing the permission granting status based on the source package in the system, a privilege escalation issue may occur.

During the OS initialization (device booting), PMS also needs to scan APKs located in app folders. Later, the existing custom permissions' protection levels will be updated according to the package information extracted from the scanned APKs. That is, the custom permission definitions recorded by the system will be updated. After the OS refreshes all permission granting status, the corresponding apps will be granted the updated custom permissions.

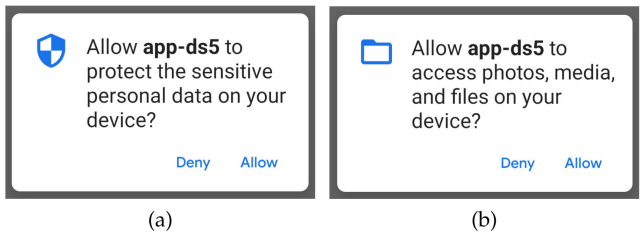


Fig. 11. Induce the user to grant a runtime custom permission to the app `app-ds5` in Android 9. (a) Give the permission a misleading description. (b) Fake the permission as a common one for saving app data.

Attack Case. The adversary creates an app `app-ds4` that defines and requests a `normal` custom permission `com.test.cp`. There is also an updated version of `app-ds4` that changes the protection level of `com.test.cp` to `dangerous` and puts it into the `PHONE` group. It also requests the `CALL_PHONE` permission. The user installs `app-ds4` and then updates it. After that, she reboots her phone. When the OS initialization is complete, `app-ds4` obtains `com.test.cp` (`dangerous` custom permission) automatically. Then it can obtain the `CALL_PHONE` permission without user consent because both `com.test.cp` and `CALL_PHONE` belong to the `PHONE` group.

Discussion. As mentioned in Section 3.1, *DS#4* was introduced when Google fixed the vulnerability discovered by Tuncay *et al.* [11]. Google's fix only considers how to break the attack flow with the minimum code modifications but ignores the consistency issue [32].

Impact. *DS#4* and its exploit have been confirmed by Google with rating *High severity* (`AndroidID-168319670`), and a CVE ID has been assigned: `CVE-2021-0317`.

6.5 DS#5: Dormant Permission Group

As mentioned in Section 2.2, for the runtime permission request, the system will present a permission dialog to the user. The text in the dialog references the permission group associated with the permission. If the group is a system group, the displayed text cannot be modified by a third-party app because the third-party app cannot re-define a system group. However, we find that:

> *DS#5: While a group has not been defined, permissions still can be assigned to this group. However, this group will not be effective until it is defined, leading to dormant permission group.*

It means that if a `dangerous` (runtime) custom permission belongs to a dormant permission group, its management will not be group-based before this group becomes effective. The system will still show the custom permission's description (can be customized by a third-party app) to the user rather than the group's information. Therefore, if the dormant permission group is a system group, a privilege escalation issue may occur.

Attack Case. In general, system groups are defined by system apps that are pre-installed in the system. Thus, system groups cannot be dormant. However, like *DS#3*, if we consider the OS update operation, a dormant system group may exist. For example, on the device running Android 9, the adversary creates an app `app-ds5` that defines a

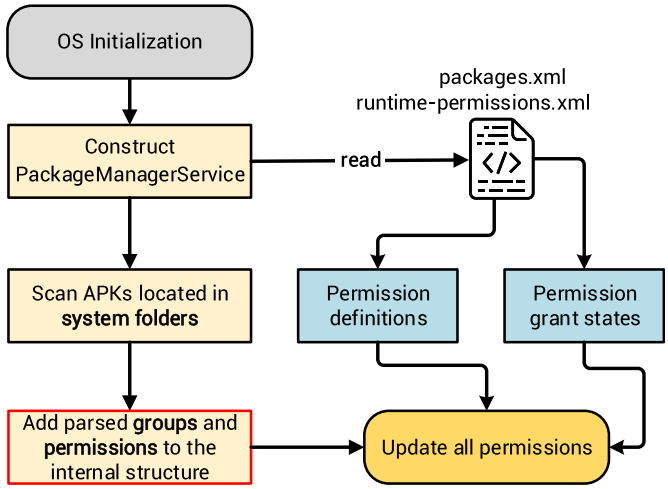


Fig. 12. Dormant permission group.

`dangerous` custom permission `com.test.cp` and puts it into the `ACTIVITY_RECOGNITION`[14] group. Also, `app-ds5` requests this custom permission and the `ACTIVITY_RECOGNITION`[15] permission (belonging to the `ACTIVITY_RECOGNITION` group).

Note that the `ACTIVITY_RECOGNITION` group is a new *system group* introduced in Android 10. Thus, it is a *dormant group* in Android 9.

The user first installs `app-ds5`. Then, she is induced to grant `com.test.cp` to `app-ds5`. Since the system will show the permission's description, the adversary can give the permission a misleading description (as shown in Fig. 11a) to make the user believe that granting this permission is necessary or fake it as another common permission for saving app data (as shown in Fig. 11b). Next, she upgrades the device to Android 10 through OTA OS update. During the OS initialization, the definition of the `ACTIVITY_RECOGNITION` group will be added to the system (as shown in Fig. 12), and this dormant permission group will become effective. Finally, since both `com.test.cp` and `ACTIVITY_RECOGNITION` are in the `ACTIVITY_RECOGNITION` group, `app-ds5` gets the `ACTIVITY_RECOGNITION` permission (`dangerous` system permission) without user consent.

Discussion. The runtime permission model provides excellent transparency for the permission request and makes the user more cautious about granting a sensitive system permission to an app. If there is a mismatch between the system permission and the app's purpose, the user will deny the permission request. However, the transparency fails to apply to a dormant system group, which misleads the user to grant a seemingly necessary or insensitive permission.

Also, as discussed in Section 6.1, the permission groups associated with the runtime permissions cannot be changed. However, the dormant groups are not restricted by this rule (i.e., can remove a permission from a group and assign it to another dormant group).

Impact. *DS#5* and its exploit have been confirmed by Google with rating *Moderate severity* (`AndroidID- 176828496`).

14. Full name: `android.permission-group.ACTIVITY_RECOGNITION`.
15. Full name: `android.permission.ACTIVITY_RECOGNITION`.

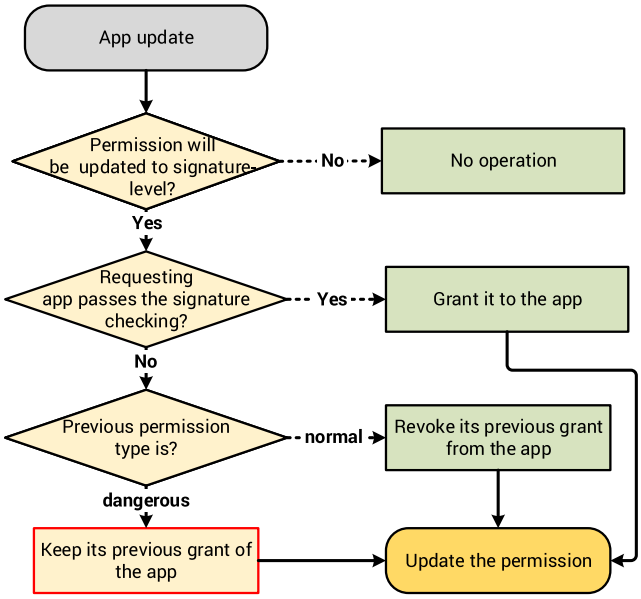


Fig. 13. Inconsistent permission type.

6.6 DS#6: Inconsistent Permission Type

As illustrated in Fig. 13, if a custom permission is changed from `normal` to `signature` during the app update, and the app requesting it does not pass the signature checking (as mentioned in Section 5.3), the updated `signature` permission will not be granted to the app, and further, the system will revoke its previous grant (as a `normal` one) from the app. However, we find that:

> *DS#6: If this* `signature` *custom permission is updated from* `dangerous`*, its previous grant (as a* `dangerous` *one) will not be revoked, causing inconsistent permission type.*

That is to say, under this situation, an app has been granted a `dangerous` custom permission, but in the system, this custom permission is a `signature` permission. What is more, Android only checks the names of permissions during permission enforcement [11]. It cannot distinguish the two permissions with the same name but different types. Thus, the app can hold a `dangerous` custom permission to gain unauthorized access to the resource protected by a `signature` custom permission.

Attack Case. The user has installed an app `app-ds6-victim` on her phone. This app declares a `dangerous` custom permission `com.test.cp` and uses this permission to protect its resources. The adversary prepares an app `app-ds6-attacker` requesting `com.test.cp`. Note that `app-ds6-attacker` and `app-ds6-victim` are signed by different certificates.

The user installs `app-ds6-attacker` and grants it the `com.test.cp` permission. Once `app-ds6-victim` is updated, and `com.test.cp` is changed from `dangerous` to `signature` (e.g., the protected resources become more private and only can be shared by some specific apps), `app-ds6-attacker` can still access the protected resources even if it is not authorized by `app-ds6-victim`.

Discussion. In general, the grant of `signature` permissions is certain, depending on apps' certificates. Whereas Android only relies on the names to distinguish permissions when enforcing the permission control. As a result, the `signature` permissions can be downgraded to `dangerous` ones, and the grant of them becomes uncertain, depending on users' decisions.

Impact. DS#6 and its exploit have been confirmed by Google with rating *Low severity* (`AndroidID-155649020`).

7 Secure Custom Permissions

This section proposes some improvements to mitigate the current security risks and discusses general design guidelines for custom permissions. Also, due to the consideration of backward compatibility, we will not introduce heavy changes to the current permission framework.

7.1 Mitigation

For each design shortcoming, we propose a minimum modification (Google preferred fix), which can immediately prevent the corresponding attacks.

For *DS#1*, the adversary re-defines a dangling custom permission and changes the original permission attributes. The direct fix is that, when the system removes a custom permission, its grants for apps should be revoked.

For *DS#2*, the adversary exploits the inconsistency in `AndroidManifest.xml` and `PLATFORM_PERMISSIONS`. Therefore, the direct fix is to remove the current inconsistent permission-group mapping data (the `UNDEFINED` group).

For *DS#3*, the adversary can elevate a custom permission to a system permission. The direct fix is that, when the system takes ownership of a custom permission, its grants for apps should be revoked.

For *DS#4*, the adversary exploits the inconsistent permission definitions in the system and the owner app. The direct fix is that, during the permission update, its grants for apps should be revoked.

For *DS#5*, the adversary puts a custom permission into a dormant permission group. The direct fix is that, permissions cannot be assigned to an undefined group.

For *DS#6*, the adversary exploits the inconsistency in the permission's definition and its granting status. The direct fix is the same as *DS#4* (i.e., during the permission update, its grants for apps should be revoked) because both of them involve the process of the permission update.

7.2 General Security Guidelines

Though the above solutions can fix the discovered design shortcomings, it is difficult to avoid that custom permission related flaws will be introduced again in the future versions of Android OS. Here we discuss some general design guidelines to secure custom permissions.

The previous research proposed to isolate system permissions from custom permissions, including (1) introducing distinct representations and not allowing custom permissions to share groups with system permissions, and (2) introducing an internal naming convention to prevent naming collisions [11]. Such solutions surely could avoid many security risks. However, they are against the design philosophy of Android permission management (i.e., do not distinguish system and custom permissions, see Section 2.2). Also, these solutions will introduce heavy logic

and code changes to the OS. Most importantly, they do not essentially fix the defects, like eliminating inconsistencies mentioned in *DS#2*, *DS#4*, and *DS#6*. Instead, we propose the following three guidelines without differentiating system and custom permissions, and avoiding logical errors.

> *Guideline#1: Any part of a permission's definition should always exist.*

Any dangling part of a permission's definition can be exploited to achieve some malicious behaviors. *DS#1* causes a dangling permission definition that can be exploited to change the permission granting status. *DS#5* causes a dangling grouping of the permission that can be exploited to trick users into allowing the permission granting request. Also, the existence of each part of a definition is beneficial to ensure the transparency of runtime permission requests.

> *Guideline#2: The definition of a permission held by the system should be consistent with the permission owner's declaration.*

The system obtains the permission definition through parsing the owner app's manifest file. The subsequent permission management should always rely on the definition obtained at this stage. Any inconsistent permission definition (different protection levels or grouping) may trigger permission upgrading. The permission-group mapping is inconsistent in *DS#2*, and the protection level is inconsistent in *DS#4*.

> *Guideline#3: If the definition of a permission is changed, the corresponding grants for apps should be revoked.*

The changes contain permission owner, grouping, and protection level. This guideline prevents the risk of TOCTTOU (time-of-check to time-of-use) issues. That is, the user only confirms the grant of the original permission, not the updated permission. To both *DS#1* and *DS#3*, the permission owner is changed without revoking grants. This guideline also can cover *DS#4*, *DS#6*, and the two attack cases (changed protection level and permission owner) discovered by Tuncay *et al*. [11].

8 Discussion

In this work, we proposed `CuPerFuzzer+` to detect the vulnerabilities in Android custom permissions and elaborated the findings of our experiments. Here we discuss some limitations of our work.

Attacks in Practice. Some attacks described in Section 6 need user interactions more than once. For instance, if an adversary wants to exploit *DS#1*, she needs to prepare two malicious apps and induce a victim user to re-install an app after uninstalling it. Such an attack workflow may be difficult to execute in practice. It is likely that, after the user uninstalls a buggy app, she may not install it again. Therefore, it would be better to conduct a user study to demonstrate the feasibility of the proposed attacks relying on multiple user interactions.

Test Case Generation. `CuPerFuzzer+` needs to generate massive test cases for fuzzing. In our design, `CuPerFuzzer+` constructs a test case randomly, including random seed selection and operation sequence construction. To improve the effectiveness of vulnerability discovery, we could deploy some feedback mechanism to generate more *interesting* test cases. That is, the current case execution result will affect how to generate the next test case. However, a feedback mechanism may result in generating too many similar test cases that are duplicate from the view of critical paths. Thus, it needs to trade off the diversity against the effectiveness of test cases.

9 Related Work

The Android permission mechanism has been studied by plenty of previous work. However, most research focused on system permissions, and rare work noticed the security implications of custom permissions. In this section, we review the related work on Android permissions.

Custom Permissions. The first custom permission related flaw was described in a blog [47]. It noticed the installation order issue of custom permissions, say "first one in wins" strategy. Nevertheless, Google did not accept this issue and mentioned, "*this is the way permissions work*" [48].

Xing *et al*. [45] discovered the Pileup flaw, which achieves privilege escalation through OS upgrading. One attack case is to exploit a custom permission to hijack a system permission. Nevertheless, their research focused on the Android OS updating mechanism rather than the custom permissions. Tuncay *et al*. [11] identified two classes of vulnerabilities in custom permissions that result from mixing system and custom permissions. In order to address these shortcomings, they proposed a new modular design called Cusper. According to our study, such a design is against the design philosophy of Android permission management. More recently, Gamba *et al*. [49] extracted and analyzed the custom permissions, both declared and requested, by preinstalled apps on Android devices. However, they focused on the aspect of service integration and commercial partnerships, not the security implications.

Unlike the above research, in this paper, we systematically study the security implications of Android custom permissions, not just individual bugs. Also, all previous flaws related to custom permissions were discovered manually. Considering the lack of an automatic tool to detect the design flaws lying in the Android permission framework, we developed `CuPerFuzzer+` and utilized it to discover several new vulnerabilities successfully. We also propose feasible fix solutions and design guidelines.

Permission Models. Various previous work studied the design of the permission-based security model. Barrera *et al*. [50] proposed a self-organizing map-based methodology to analyze the permission model of the early version of Android OS. Wei *et al*. [51] studied the evolution of the Android ecosystem (platform and apps) to understand the security implications of the permission model. Fragkaki *et al*. [12] developed a framework for formally analyzing Android-style permission systems. Backes *et al*. [2] studied

the internals of the Android application framework and provided a high-level classification of its protected resources. Based on Android 6.0, Zhauniarovich *et al.* [1] analyzed the design of the permission system, especially the introduction of runtime permissions. More recently, Tuncay *et al.* [52] identified false transparency attacks in the runtime permission model, which achieves the phishing-based privilege escalation on runtime permissions.

To improve the current permission model, Dawoud *et al.* [3] proposed DroidCap to achieve per-process permission management, which removes Android's UID-based ambient authority. Raval *et al.* [53] proposed Dalf, a framework for extensible permissions plugins that provides both flexibility and isolation. The possibilities of flexible and fine-grained permission management also were studied by ipShield [54], SemaDroid [55], SweetDroid [56], and Dr. Android [57].

Permission Usage. From the aspect of app developers, some researchers focused on studying whether permissions were used correctly in Android apps. Felt *et al.* [4] developed a tool – Stowaway to detect over-privilege in apps, and they found that about one-third are over-privileged. Au *et al.* [58] built PScout to extract the permission specification from the Android OS source code using static analysis, which provided meta-data supports for the permission usage analysis. Xu *et al.* [5] designed and implemented Permlyzer, a framework for automatically analyzing the use of permissions in Android apps. Fang *et al.* [6] analyzed the potential side effects of permission revocation in Android apps.

Usable Security. From the view of user interaction, previous work has shown that most users do not pay attention to permissions during app installation [59]. Bonné *et al.* [60] focused on the usability of runtime permissions, and their study suggests the context provided via runtime permissions appears to be helping users make decisions. The study of Wijesekera *et al.* [61] shows the visibility of the requesting app and the frequency at which requests occur are two significant factors in designing a runtime consent platform. More recently, Shen *et al.* [62] identified several common misunderstandings on the runtime permission model among users due to the limited system-provided information and explored five types of information that are helpful for users' decisions on the runtime permission requesting.

10 Conclusion

In this paper, we systematically study the security implications of Android custom permissions. Specifically, we designed `CuPerFuzzer+`, a black-box fuzzer, to detect custom permission related privilege escalation issues automatically. During the real-world experiments, it discovered 5,932 effective cases with 47 critical paths successfully. Our further analysis showed that these effective cases could be attributed to six fundamental design shortcomings lying in the Android permission framework. We also demonstrated concrete exploit cases of these flaws and proposed general design guidelines to secure Android custom permissions.

Acknowledgments

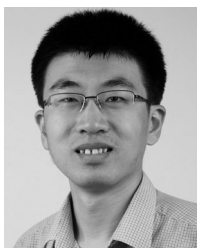
We thank the anonymous reviewers for their insightful comments.

References

[1] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: An updated view on the Android permission system," in *Proc. 19th Int. Symp. Res. Attacks Intrusions Defenses*, 2016, pp. 346–367.
[2] M. Backes, S. Bugiel, E. Derr, P. D. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the Android application framework: Re-visiting android permission specification analysis," in *Proc. 25th USENIX Secur. Symp. USENIX-SEC*, 2016, pp. 1101–1118.
[3] A. Dawoud and S. Bugiel, "DroidCap: OS support for capability-based permissions in Android," in *Proc. 26th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.
[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur*, 2011, pp. 627–638.
[5] W. Xu, F. Zhang, and S. Zhu, "Permlyzer: Analyzing permission usage in Android applications," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng.*, 2013, pp. 400–410.
[6] Z. Fang *et al.* "revDroid: Code analysis of the side effects after dynamic permission revocation of Android apps," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 747–758.
[7] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. 20th USENIX Secur. Symp. USENIX-SEC*, 2011, pp. 331–346.
[8] Y. Zhang *et al.*, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. 20th ACM Conf. Comput. Commun. Secur.*, 2013, pp. 611–622.
[9] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–12.
[10] "Define a custom app permission." Accessed: Jul. 4, 2021. [Online]. Available: https://developer.android.com/guide/topics/permissions/defining
[11] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter, "Resolving the predicament of Android custom permissions," in *Proc. 25th Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[12] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing Android's permission system," in *Proc. 17th Eur. Symp. Res. Comput. Secur*, 2012, pp. 1–18.
[13] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of design flaws in the Android permission protocol through bounded verification," in *Proc. Formal Methods 20th Int. Symp.*, 2015, pp. 73–89.
[14] "Security updates and resources: Severity." Accessed: Jul. 4, 2021. [Online]. Available: https://source.android.com/security/overview/updates-resources#severity
[15] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *Proc. 42nd IEEE Symp. Secur. Privacy*, 2021, pp. 70–86.
[16] "PackageManager." Accessed: Sep. 3, 2020. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/services/core/java/com/android/server/pm/
[17] "PermissionController." Accessed: Sep. 3, 2020. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/
[18] "Permissions overview: Protection levels." Accessed: Sep. 3, 2020. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview#normal-dangerous
[19] "Runtime permissions." Accessed: Jul. 4, 2021. [Online]. Available: https://source.android.com/devices/tech/config/runtime_perms
[20] "Permissions overview: Permission groups." Accessed: Sep. 3, 2020. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview#perm-groups
[21] "PackageManagerService.java." Accessed: Sep. 3, 2020. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java
[22] "Google play instant." Accessed: Jul. 4, 2021. [Online]. Available: https://developer.android.com/topic/google-play-instant
[23] "Scrapy." Accessed: Jul. 4, 2021. [Online]. Available: https://scrapy.org/
[24] "Androguard," 2020. [Online]. Available: https://github.com/androguard

[25] W. Diao *et al.*, "Kindness is a risky business: On the usage of the accessibility APIs in Android," in *Proc. 22nd Int. Symp. Res. Attacks Intrusions Defenses*, 2019, pp. 261–275.
[26] J. Huang *et al.*, "SUPOR: Precise and scalable sensitive user input detection for Android apps," in *Proc. 24th USENIX Secur. Symp. USENIX-SEC*, 2015, pp. 977–992.
[27] S. Junyi, "Jieba," 2020. [Online]. Available: https://github.com/fxsjy/jieba
[28] J. Thanaki, "Feature engineering and NLP algorithms," in *Python Natural Lang. Process.*, ch. 5. Birmingham, U.K.: Packt Publishing, 2017, pp. 102–171.
[29] "JPush." Accessed: Jul. 4, 2021. [Online]. Available: https://docs.jiguang.cn/en/jpush/guideline/intro/
[30] "Rongling cloud communication." Accessed: Jul. 4, 2021. [Online]. Available: https://www.yuntongxun.com/
[31] Z. Yan, "AndPermission," 2020. [Online]. Available: https://github.com/yanzhenjie/AndPermission
[32] S. Ganov, "Bug: 33860747," 2016. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/78efbc95412b8efa9a44d573f5767ae927927d48
[33] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek, "A temporal permission analysis and enforcement framework for Android," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 846–857.
[34] "Android debug bridge (ADB)." Accessed: Jul. 4, 2021. [Online]. Available: https://developer.android.com/studio/command-line/adb
[35] "Factory images for nexus and pixel devices: Flashing instructions." Accessed: Jul. 4, 2021. [Online]. Available: https://developers.google.com/android/images#instructions
[36] "Building Android." Accessed: Jul. 4, 2021. [Online]. Available: https://source.android.com/setup/build/building
[37] "Apktool." Accessed: Jul. 4, 2021. [Online]. Available: https://ibotpeaches.github.io/Apktool/
[38] "Signing JAR files." Accessed: Jul. 4, 2021. [Online]. Available: https://docs.oracle.com/javase/tutorial/deployment/jar/signing.html
[39] "Access to device location in the background requires permission." Accessed: Jul. 4, 2021. [Online]. Available: https://developer.android.google.cn/about/versions/10/privacy/changes#app-access-device-location
[40] "AppPermissions.java." Accessed: Jul. 4, 2021. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/src/com/android/packageinstaller/permission/model/AppPermissions.java
[41] "Utils.java." Accessed: Sep. 3, 2020. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/src/com/android/packageinstaller/permission/utils/Utils.java
[42] "AndroidManifest.xml." Accessed: Sep. 3, 2020. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/core/res/AndroidManifest.xml
[43] P. P. Moltmann, "Remove grouping for platform permissions," 2018. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/17eae45cf9a3948ed268e51bf13528ad82a465f0
[44] P. P. Moltmann, "Give platform permissions a dummy group," 2018. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/2a01ddbb4ea572ec82687dc0d9602eff36cc0886
[45] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your Android, elevating my malware: Privilege escalation through mobile OS updating," in *Proc. 35th IEEE Symp. Secur. Privacy*, 2014, pp. 393–408.
[46] C. Tate, "Bug: 11242510," 2013. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/3aeea1f
[47] M. L. Murphy, "Vulnerabilities with Custom Permissions," 2014. [Online]. Available: https://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html
[48] "Permissions are install-order dependent," 2018. [Online]. Available: https://issuetracker.google.com/issues/36941003
[49] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed Android software," in *Proc. 41st IEEE Symp. Secur. Privacy*, 2020, pp. 1039–1055.
[50] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 73–84.
[51] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the Android ecosystem," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 31–40.
[52] G. S. Tuncay, J. Qian, and C. A. Gunter, "See no evil: Phishing for permissions with false transparency," in *Proc. 29th USENIX Secur. Symp. USENIX-SEC Virt.*, 2020, pp. 415–432.
[53] N. Raval, A. Razeen, A. Machanavajjhala, L. P. Cox, and A. Warfield, "Permissions plugins as Android Apps," in *Proc. 17th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2019, pp. 180–192.
[54] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. B. Srivastava, "ipShield: A framework for enforcing context-aware privacy," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 143–156.
[55] Z. Xu and S. Zhu, "SemaDroid: A privacy-aware sensor management framework for smartphones," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 61–72.
[56] X. Chen, H. Huang, S. Zhu, Q. Li, and Q. Guan, "SweetDroid: Toward a context-sensitive privacy policy enforcement framework for Android OS," in *Proc. Workshop Privacy Electron. Soc.*, 2017, pp. 75–86.
[57] J. Jeon *et al.*, "Dr. Android and Mr. Hide: Fine-grained permissions in Android applications," in *Proc. 2nd Workshop Secur. Privacy Smartphones Mobile Devices*, 2012, pp. 3–14.
[58] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. 19th ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228.
[59] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. 8th Symp. Usable Privacy Secur.*, 2012, pp. 3:1–3:14.
[60] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft, "Exploring decision making with Android's runtime permission dialogs using in-context surveys," in *Proc. 13th Symp. Usable Privacy Secur*, 2017, pp. 195–210.
[61] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Secur. Symp. USENIX-SEC*, 2015, pp. 499–514.
[62] B. Shen *et al.*, "Can systems explain permissions better? Understanding users' misperceptions under smartphone runtime permission model," in *Proc. 30th USENIX Secur. Symp. USENIX-SEC*, 2021, pp. 751–768.

Rui Li received the BEng degree in information security from the Central University of Finance and Economics, Beijing, China, in 2019. She is currently working toward the PhD degree at the School of Cyber Science and Technology, Shandong University, Qingdao, China. Her research interests include system security and mobile security.



Wenrui Diao received the PhD degree in information engineering from the Chinese University of Hong Kong, Hong Kong, in 2017, under the supervision of Prof. Kehuan Zhang. He is currently a professor with the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include mobile security and IoT security. He received the 2019 ACM SIGSAC China Rising Star Award.

Zhou Li (Senior Member, IEEE) received the PhD degree in computer science from Indiana University Bloomington, Bloomington, Indiana. He was a principal research scientist with RSA Labs from 2014-2018. He is currently an assistant professor with EECS Department, University of California, Irvine. He has published more than 40 refereed research articles. His research interests include cyber security, privacy, and machine learning.

Shishuai Yang received the BEng degree in software engineering from Henan University, Kaifeng, China, in 2020. He is currently working toward the MEng degree at the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interest include mobile security.

Shuang Li received the BSc degree in information security from Shandong University, Qingdao, China, in 2021. She is currently working toward the MEng degree at the School of Cyber Science and Technology, Shandong University, Qingdao, China. Her research interests include mobile security and software security.

Shanqing Guo received the PhD degree in computer software from Nanjing University, Nanjing, China, in 2006. He is currently a professor with the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include software security, network security, and AI security.