Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings

Rui Li*†, Wenrui Diao*†(✉), Zhou Li‡, Jianqi Du*†, and Shanqing Guo*†
*School of Cyber Science and Technology, Shandong University
leiry@mail.sdu.edu.cn, diaowenrui@sdu.edu.cn, dujianqi@mail.sdu.edu.cn, guoshanqing@sdu.edu.cn
†Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
‡University of California, Irvine, zhou.li@uci.edu

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00070

***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 permissions 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 2,384 effective cases with 30 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*, and *inconsistent permission definition*. Exploiting any of these shortcomings, a malicious app can obtain `dangerous` system permissions without user consent and further access unauthorized platform resources. On top of these observations, we propose some general design guidelines to secure custom permissions. Our findings have been acknowledged by the Android security team and rated as High severity.

I. 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 [49], [25], [31], permission usage [33], [46], [32], and malware detection [35], [48], [23]. 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* [7], 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. [41] 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 III). 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 light-weight fuzzing tool called CUPERFUZZER to discover custom permission related privilege escalation vulnerabilities. Different from the previous approaches of permission system modeling [36], [26], 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 Pixel 2 phones equipped with Android 9 / 10, finally, CUPERFUZZER discovered 2,384 successful exploit cases after executing 40,195 fuzzing tests.

These effective cases were further converted to 30 critical paths, say the least necessary operations triggering a privilege

DOI 10.1109/SP40001.2021.00070

escalation issue. Combined with the analysis on the source code of Android OS, we identified four 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 a custom permission to a `dangerous` system permission.
- **DS#4**: *Inconsistent permission definition*: causing breaking the integrity of custom permission definitions.

A malicious app can exploit any of the above design shortcomings to **obtain `dangerous` system permissions without user consent**. As showcases, we present four 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 reported issues have been confirmed with positive severity rating [19], as shown below.

- **DS#1**: rated as **High severity**, assigned `CVE-2021-0307`.
- **DS#2**: rated as **High severity**, assigned `CVE-2020-0418`.
- **DS#3**: rated as **High severity**, assigned `CVE-2021-0306`.
- **DS#4**: rated as **High severity**, assigned `CVE-2021-0317`.

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

Contributions. The main contributions of this paper are:

- *Tool Design and Implementation.* We designed and implemented an automatic black-box fuzzing tool, CUPERFUZZER, to discover custom permission related privilege escalation vulnerabilities in Android.
- *Real-world Experiments.* We deployed CUPERFUZZER under the real-world settings and conducted massive fuzzing analysis. In the end, it discovered 2,384 privilege escalation cases with 30 critical paths.
- *New Design Shortcomings.* We identified four severe design shortcomings lying in the Android permission framework. Malicious apps can exploit these flaws to obtain `dangerous` system permissions without user consent.
- *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 guidelines to secure Android custom permissions.

Roadmap. The rest of this paper is organized as follows. Section II provides the necessary background of Android custom permissions. Section III gives a motivation case and threat model used in this paper. Section IV introduces the detailed design of CUPERFUZZER, and Section V presents the experiment results. The design flaws of custom permissions are analyzed in Section VI. In Section VII, we propose the mitigation solutions and general design guidelines. In Section VIII, we discuss some limitations of our work. Section IX reviews related work, and Section X concludes this paper.

[1]In the following sections, we use **DS#1**, **DS#2**, **DS#3**, and **DS#4** for short.

II. 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.

A. Android Permission Mechanism

In Android, sensitive APIs and system resources are protected by the permission mechanism. To access them, apps must declare 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` [11] and `PermissionController` [13].

Permissions are mainly divided into three protection levels: `normal`, `signature`, and `dangerous`. The system grants apps `normal` and `signature` permissions at the install time. The difference is that `signature` permissions can only be used by the apps signed by the same certificate as the app that defines the permission [16]. On the other hand, users can choose to grant or deny `dangerous` permissions at runtime. Therefore, `dangerous` permissions are also called *runtime permissions*, and accordingly, `normal` and `signature` permissions are called *install-time permissions*. Install-time permissions cannot be revoked by users once they are granted, on the contrary, runtime permissions can be revoked at any time.

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 [15].

From an internal view, to an app, the processes of grant and revocation of 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. Besides, the granting status of permissions are also recorded by `runtime-permissions.xml`[2] (runtime permissions) and `packages.xml`[3] (install-time permissions) for persistent storage.

B. 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 `CALL_PHONE` permission to make a phone call. For third-party apps, they can define their

[2]Location: `/data/system/users/0/runtime-permissions.xml`
[3]Location: `/data/system/packages.xml`

own permissions as well, called *custom permissions*, to share their resources and capabilities with other apps.

```
<!-- Define a custom permission -->
<permission
android:name="com.test.cp"
android:protectionLevel="normal"
android:permissionGroup="android.permission-group.PHONE"/>
< !-- Request a custom permission -->
<uses-permission android:name="com.test.cp"/>
```

Listing 1: Define and request a custom permission.

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.

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 (such as the `PHONE` group in the above example). In order to use the custom permission, the app needs to request it through the *uses-permission* element in its manifest file [7].

Design Philosophy. In most usage scenarios, Android does not intend to distinguish 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 [12]. 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) The permission owner is recorded as the app that defines this permission first. (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.

TABLE I: Protection levels of custom permissions.

Protection Level	Amount	Percentage
`normal`	26,330	32.09%
`dangerous`	1,986	2.42%
`signature`†	53,724	65.48%
`instant`‡	12	0.01%

†: Include mixed levels: `signature|privileged` and `signatureOrSystem`.
‡: Only for instant apps [9].

TABLE II: Permission groups of custom permissions.

Group Type	Amount	Percentage
System Group	4,526	83.64%
Custom Group	885	16.36%

C. Usage Status

To understand the status quo of using custom permissions, we conducted a large-scale measurement based on 208,987 APK files crawled from third-party app markets and randomly selected from the AndroZoo dataset [22], mainly ranging from 2017-2019. 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 script to scan the manifest files of apps. Through parsing custom permission related attributes, we obtained the first-hand statistics data for further processing.

To **Question-1**, our results show that 52,601 apps (around 25.2%) declare a total of 82,052 custom permissions. We could find the use of custom permissions is not unusual. On the aspect of protection levels, more than 65% of these permissions are `signature`, as listed in Table I. The reason for such a high percentage may be that a series of apps are developed by the same company (signed by the same certificate) and need to share some resources only restricted to themselves. On the other hand, `normal` permissions account for 32.09%, and `dangerous` permissions account for only 2.42%.

Besides, 5,411 custom permissions (around 6.6% of the total) are assigned to permission groups, see Table II. Among them, system permission groups are used more frequently than custom permission groups (4,526 vs. 885). Using a system group can simplify the permission UI shown to the user, which is recommended by Google [7].

To **Question-2**, we crawled the custom permission names and their permission descriptions for analysis. Combined with a number of manual case studies, here we summarize the purposes of using custom permissions.

- *Use services provided by third-parties.* For example, up to 16,259 apps in our dataset declare the `JPUSH_MESSAGE`

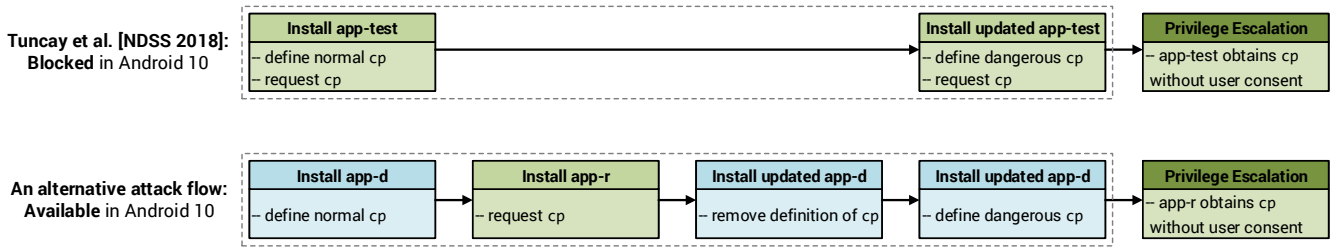

Fig. 1: An alternative attack flow achieving privilege escalation.

permission to obtain the message push service offered by the JPush platform [10].

- *Restrict the accessing to apps' shared data.* For example, `com.qidian.QDReaderMM` defines the `READ_DB`[4] permission to control the accessing to its database of e-books.
- *Control the communication between apps.* For example, only the apps with the `BROADCAST_RECEIVER`[5] permission can send a broadcast to the broadcast receiver of `com.tencent.portfolio` which defines this permission.

III. Motivation and Threat Model

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

A. 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. [41] 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 Figure 1, 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 definition 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.

Our Findings. According to the Android Security Bulletins and the corresponding source code change logs [5], the above attack has been fixed on 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 Figure 1, 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 VI-A).

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 [41] may be only the tip of the iceberg, and an automatic analysis tool is needed. Besides, our ultimate target should be identifying design shortcomings lying in the permission framework, not just discovering successful attack cases.

B. 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.

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.

C. 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

[4]Full name: `com.qidian.QDReader.permission.READ_DB`
[5]Full name: `com.tencent.portfolio.permission.BROADCAST_RECEIVER`

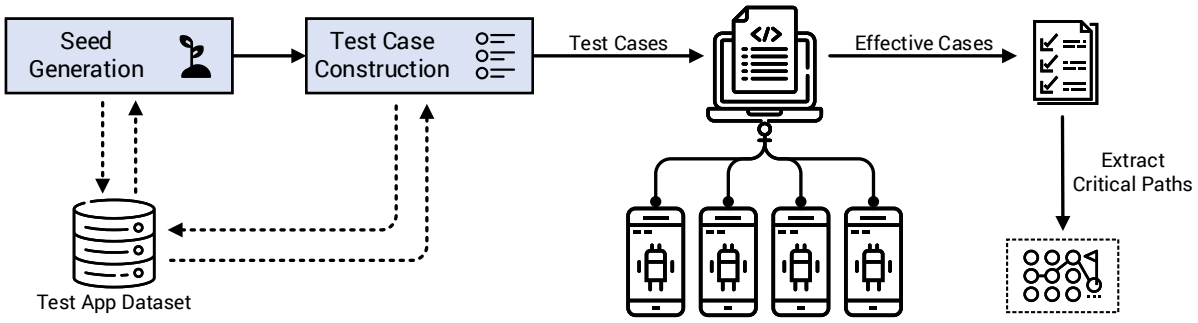

Fig. 2: Overview of CUPERFUZZER.

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.

IV. 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 III-B, each test case is essentially *an 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 Figure 2, on a high level, CUPERFUZZER contains five main steps, as following.

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

A. Seed Generation

As mentioned in our threat model (see Section III-C), 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[6]. 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 generation 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. Besides, based on our workflow, the number of seed apps is usually small, not enough for randomness.

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, say *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.

B. Test Case Construction

Next, CUPERFUZZER constructs a complete test case. As illustrated in Figure 3, it is an execution sequence consisting of multiple test apps and operations that may affect the granting of requested permissions.

Operation Selection. After reviewing the Android technical documents and source code [12], we confirm four operations

[6]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.

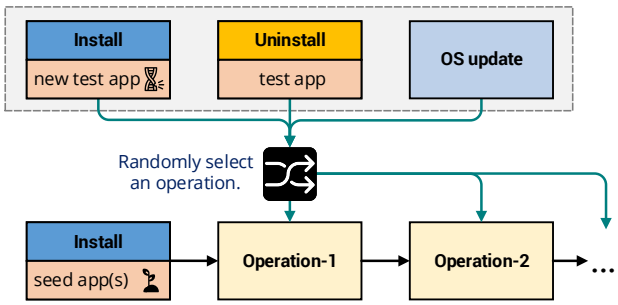

Fig. 3: Construct a test case (an execution sequence).

meeting the requirements: app installation, app uninstallation, app update, and OS update. 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 permissions may be added to the system, and existing system permissions may be removed.

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.

Mutating Test App. 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 operation sequence.

Test Case Construction. When CUPERFUZZER constructs a test case, it randomly selects uncertain amount of operations from the {*app installation*, *app uninstallation*, *OS update*} to generate an 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 last.
- The OS update operation only can be executed once. Our test focuses on the lasted 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 seed apps and test cases deriving from one seed app.

C. 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.
- The images of emulators may exist undocumented modifications to adapt to the underlying hardware.

Parallel Case Execution. To facilitate the 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. 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 before, there are three kinds of operations in a test case. Among them, the app installation (uninstallation) operation can be executed through the `adb install` (`uninstall`) command directly. To 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).

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 [6].

D. Effective Case Checking

To each completed test case, CUPERFUZZER needs to check whether it is an effective case that achieves privilege escalation. An effective case can be determined through 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 the 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 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 exists any granted `dangerous` or `signature` permission matching the above rules, this test case is effective and will be recorded for further analysis.

E. 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) *Test Cases Classification.* 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.

(2) *Find Critical Path.* In each category, we find the test cases with the least operations, called *candidate cases*. To a randomly selected candidate case, we delete its first / last operation (including the operation of seed installation) and execute this case again. If the same execution result occurs, we add this pruned case into this category and repeat this step. If it is different, we obtain a *critical path* (the operation sequence of this candidate case) and record this path.

(3) *Delete Duplicate Cases.* In the same category, if an effective case contains the extracted critical path, this case will be deleted. Note that, in this matching process, we do not require that the installed apps used in the operation of app installation are the same. We repeat Step 2 and Step 3 until all effective cases have been deleted.

Based on the extracted critical paths, we try to determine the root causes of the discovered effective cases by analyzing the source code of Android OS.

V. IMPLEMENTATION AND EXPERIMENT RESULTS

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

A. Prototype Implementation

We implemented a full-feature prototype of CUPERFUZZER with around 653 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 IV-C, `adb` and `fastboot` are used for device control and OS update.

For test app generation, `Apktool` [2] and `jarsigner` [20] 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.

B. Experiment Setup

Hardware Setup. In our experiments, we deployed a laptop (Windows 10, 4G RAM, Intel Core i5) 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 IV-C, 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 system parameters. CUPERFUZZER executes a test case containing OS update operations 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 contain 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[7] from Android 9 and Android 10. The results showed there are 88 system

[7]Obtained through `adb shell pm list permissions -f -g` on Pixel 2.

permissions (3 for `normal`, 3 for `dangerous`, and 82 for `signature`) which exist in Android 10 but not in Android 9, as listed in Appendix A. 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 models, 3 available permission names, 3 kinds of protection levels, and 12 system permission groups[8] (as listed in Appendix B), 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.

Execution sequence. An app execution sequence (test case) will be generated based on a selected seed app. Since there are 3 kinds of protection levels and 13 kinds of groups, the subsequent app installation operation can install 40 (3×13 + no custom permission definition) kinds of mutated test apps. If an execution sequence contains three app installation operations (without including the installation of the seed app), there will be 64,000 ($40\times40\times40$) combinations, which is quite large. On the other hand, we hope CUPERFUZZER can perform different execution sequences as many as possible. Therefore, we set the following restriction: when generating a new test app, only one attribute can be changed differently from the previously installed test app, say protection level or group. Under this restriction, the kinds of mutated test apps become 15, and the combinations of three app installation operations have been reduced to 3,346.

Test Case Execution. After applying the above measures, a seed app can still 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.

C. Result Summary

During our experiments, CUPERFUZZER executed 40,195 test cases on four Pixel 2 phones in 319.3 hours (around 13.3 days) until we stopped it.

[8]Obtained through `adb shell pm list permission-groups` on Pixel 2.

TABLE III: Average execution time of the operation.

Operation Type	Operation	Time Cost (second)
Case execution	App installation	1.1
	App uninstallation	0.5
	OS update	109.8
Environment reset	Factory reset	60.2
	OS downgrade	129.5

Efficiency. On average, CUPERFUZZER costed 114.4s to execute a test case in the experiment, which is slower than the case execution in an ideal situation (without execution errors). The extra time cost came from error handling. Among the 40,195 test cases, 4,788 cases (around 11.9%) cannot be executed successfully for the first time, and CUPERFUZZER skipped them. The main reason is that when multiple devices enter the `fastboot` mode at the same time, reading or writing data may fail with an error message, such as "*status read failed (too many links)*" or "*command write failed (unknown error)*". Under this situation, the device is unable to reboot normally and has to go through environment reset. Note that we do not retest the failed cases because there are many cases waiting for testing, and such a retest step will increase the complexity of execution logic.

In Table III, we list the average time cost of operation execution and environment reset in an ideal situation (i.e., every operation is executed successfully). The time cost of an upgrade or downgrade operation includes the cost of flashing images and device reboot.

Results. Finally, CUPERFUZZER discovered 2,384 effective test cases which triggered privilege escalation issues. All these issues were matched by the first checking rule defined in Section IV-D, say *obtaining `dangerous` permissions without user consent*. To the second rule (*obtaining `signature` permissions*), through analyzing the source code of Android OS, we find 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.

Also, CUPERFUZZER further extracted 30 critical paths from these discovered effective cases, as listed in Table IV. In this table, we can find that, if the critical path is very simple, many cases may contain this path. For example, up to 1,904 effective cases are derived from Path No.1, a two-operation path (`Installation` → `OS-update`). Below we show some interesting findings.

- As mentioned in Section III-A, the permission protection level changing operation has been blocked by Google. However, in Path No.3, 5-15, an additional OS update operation reactivates such a privilege escalation attack.
- In Path No.16 and Path No.28, 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.

TABLE IV: Discovered critical paths in our experiments.

No.	Effective Cases	Seed Mode	Critical Path†	Privilege Escalation (Granted Permissions)	Flaw
1	1,904	single-app dual-app	Installation [ACTIVITY_RECOGNITION, normal, NULL] → OS-update	ACTIVITY_RECOGNITION	**DS#3**
2	3	dual-app	Installation [com.test.cp, normal, NULL] → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL]	com.test.cp	**DS#1**
3	4	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → OS-update	com.test.cp	**DS#4**
4	92	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, NULL]	com.test.cp	**DS#1**
5-15‡	44	single-app dual-app	Installation [com.test.cp, normal, {Group}] → Installation [com.test.cp, dangerous, {Group}] → OS-update	com.test.cp system permissions in {Group}	**DS#4**
16	4	single-app dual-app	Installation [com.test.cp, normal, UNDEFINED] → Installation [com.test.cp, dangerous, UNDEFINED] → OS-update	com.test.cp READ_CONTACTS ... (30 dangerous system permissions in total)	**DS#2**
17-27‡	304	dual-app	Installation [com.test.cp, normal, {Group}] → Uninstallation → Installation [com.test.cp, dangerous, {Group}]	com.test.cp system permissions in {Group}	**DS#1**
28	27	dual-app	Installation [com.test.cp, normal, UNDEFINED] → Uninstallation → Installation [com.test.cp, dangerous, UNDEFINED]	com.test.cp READ_CONTACTS ... (30 dangerous system permissions in total)	**DS#2**
29	1	dual-app	Installation [com.test.cp, normal, NULL] → OS-update → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL]	com.test.cp	**DS#1**
30	1	dual-app	Installation [com.test.cp, normal, NULL] → OS-update → Uninstallation → Installation [com.test.cp, dangerous, NULL]	com.test.cp	**DS#1**

†: In the app `Installation` operation, the custom permission defined by the installed test app is put in the 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.

We manually analyzed the extracted 30 critical paths and reviewed the corresponding source code of Android OS. Finally, we identified four fatal design shortcomings lying in the Android permission framework, as labeled in the last column of Table IV. In the following sections, we will discuss these shortcomings and corresponding improvements with more details.

VI. 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.

A. DS#1: Dangling Custom Permission

As illustrated in Figure 4, 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 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 with 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.

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). The adversary also prepares an updated version of `app-ds1-d` which declares the following permission.

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

Listing 2: Updated custom permission.

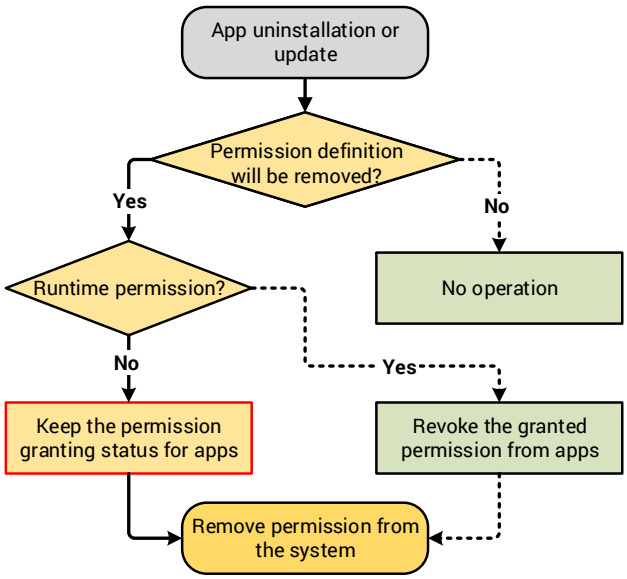

Fig. 4: Dangling custom permission.

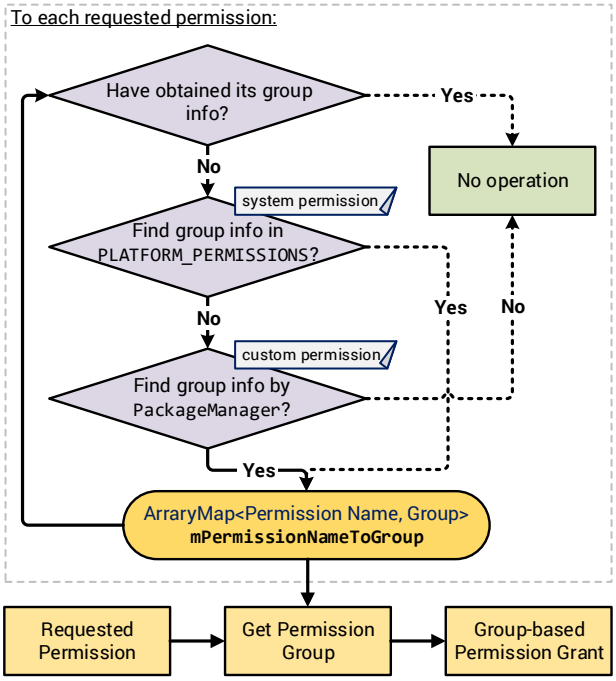

Fig. 5: Inconsistent permission-group mapping.

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 also 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. Note that, when the user installs the updated `app-ds1-d`, PMS scans the package and adds the updated custom permission `com.test.cp` into 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 runtime 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.

The root cause of the attack case described in Section III-A is also **DS#1**. It creates a dangling custom permission during app updating. 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 to another group) [18].

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`.

B. 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 Figure 5, when Android OS processes a `dangerous` permission granting request, it will query the group (members) information of the requested permission through `ArrayMap mPermissionNameToGroup` [3]. 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 it 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 through querying `PLATFORM_PERMISSIONS` and `PackageManager`. Since `PLATFORM_PERMISSIONS` is a

hard-coded <system permission, system group> mapping array defined in PermissionController [21], 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 mainly relies on AndroidManifest.xml, the core manifest file of the system [1], 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 AndroidManifest.xml, 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.

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

Then, the adversary creates an updated version of app-ds2, and it defines and requests a dangerous custom permission com.test.cp. Also, app-ds2 requests all dangerous system permissions, as shown below.

```
<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"
    />
```

Listing 3: Updated version of app-ds2.

Next, the user installs this updated version of app-ds2, and the system automatically grants it with all dangerous system permissions without user permitting.

As mentioned before (see Figure 5), to each requested permission, the system will add its group members information to ArrayMap mPermissionNameToGroup. To system permissions (Line 6-10), the <permission, group> mapping looks like:

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

Listing 4: Mapping mPermissionNameToGroup.

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

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

Listing 5: Updated mapping mPermissionNameToGroup.

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 [17], [8] in the source code of Android OS, the UNDEFINED group was introduced as a dummy group to prevent apps querying 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.

C. DS#3: Custom Permission Elevating

As illustrated in Figure 6, 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 permission is not the system, this permission will be overridden. However, we find that:

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

That is to say, if an app has been granted with 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 becomes possible. For instance, on an Android 9 device, the adversary creates an app app-ds3, which defines

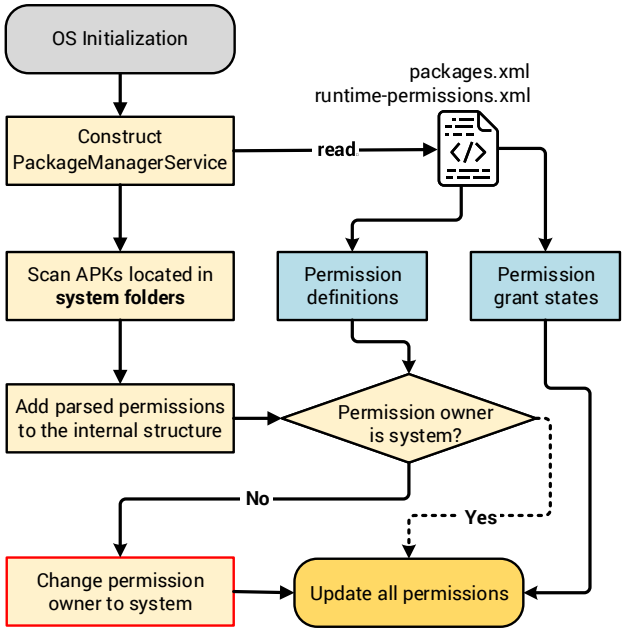

Fig. 6: Custom permission elevating.

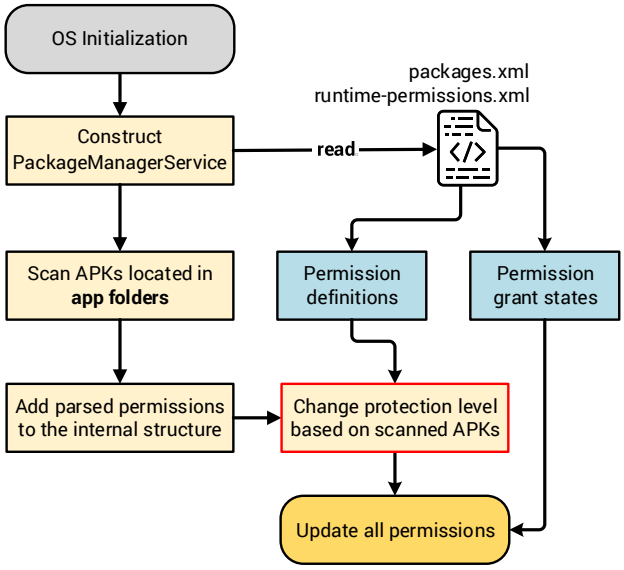

Fig. 7: Inconsistent permission definition.

and requests a custom permission `ACTIVITY_RECOGNITION`, as follows.

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

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

Listing 6: Define and request `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 carries out OTA OS update, and later the device reboots with running Android 10. After finishing OS initialization, `app-ds3` has been granted with the `ACTIVITY_RECOGNITION` permission (`dangerous` system permission) automatically, say privilege escalation.

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, on 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 was that, during the OS initialization, the OS will override all permissions declared by the system, say taking the ownership [4]. 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`.

D. DS#4: Inconsistent Permission Definition

An app installation operation may also update an existing custom permission defined by itself. During this process, if the 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 Figure 1). 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 of refreshing the permission granting status based on the source package in the system, a privilege escalation issue may occur. As illustrated in Figure 7, during the OS initialization, 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 permission definition recorded by the system will be updated. After the OS refreshes all permission granting status, the corresponding apps will be granted with the updated custom permissions.

Note that, different from **DS#3**, since the exploit of **DS#4** does not need a new permission introduced in Android 10, an operation leading OS initialization is enough (e.g., device reboot), and the OS upgrading operation is not necessary.

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` which changes the protection level of `com.test.cp` to `dangerous` and puts `com.test.cp` 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 reboot 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. Our further investigation shows that **DS#4** was introduced when Google fixed the vulnerability discovered by Tuncay et al. [41]. Google's fix only considered how to break the attack flow with the minimum code modifications but ignored the consistency issue [5].

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`.

VII. Secure Custom Permissions

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

A. Mitigation

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

To **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.

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

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

To **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.

B. 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 [41]. 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 II-B). 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** and **DS#4**. Instead, we propose the following two guidelines without differentiating permission types and avoiding logical errors.

> **Guideline#1:** *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** and the two attack cases (changed protection level and permission owner) discovered by Tuncay et al. [41].

> **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 (changing protection level or group) may trigger permission upgrading. The permission-group mapping is inconsistent in **DS#2**, and the protection level is inconsistent in **DS#4**.

VIII. 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 VI 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 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 which are duplicate from the view of critical paths. Thus, it needs to trade off the diversity against the effectiveness of test cases.

IX. 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 [39]. 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*" [14].

Xing et al. [45] discovered the Pileup flaw, which achieves privilege escalation through OS upgrading. One 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. [41] 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. [37] extracted and analyzed the custom permissions, both declared and requested, by pre-installed 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. [27] proposed a self-organizing map-based methodology to analyze the permission model of the early version of Android OS. Wei et al. [43] studied the evolution of the Android ecosystem (platform and apps) to understand the security implications of the permission model. Fragkaki et al. [36] developed a framework for formally analyzing Android-style permission systems. Backes et al. [25] 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. [49] analyzed the design of the permission system, especially the introduction of runtime permissions. More recently, Tuncay et al. [42] 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. [31] proposed DroidCap to achieve per-process permission management, which removes Android's UID-based ambient authority. Raval et al. [40] 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 [29], SemaDroid [47], SweetDroid [30], and Dr. Android [38].

Permission Usage. From the aspect of app developers, some researchers focused on studying whether permissions were used correctly in Android apps. Felt et al. [33] developed a tool – Stowaway to detect over-privilege in apps, and they found about one-third are over-privileged. Au et al. [24] 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. [46] designed and implemented Permlyzer, a framework for automatically analyzing the use of permissions in Android apps. Fang et al. [32] 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 [34]. Bonné et al. [28] 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. [44] shows the visibility of the requesting app and the frequency at which requests occur are two significant factors in designing a runtime consent platform.

X. 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 2,384 attack cases with 30 critical paths successfully. Our further investigation showed these effective cases could be attributed to four fundamental design shortcomings lying in the Android permission framework. We also demonstrated concrete attacks and proposed general design guidelines to secure Android custom permissions.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (Grant No. 61902148 and 91546203), Major Scientific and Technological Innovation Projects of Shandong Province, China (Grant No. 2018CXGC0708 and 2019JZZY010132), and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] AndroidManifest.xml. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/core/res/AndroidManifest.xml.

[2] Apktool. https://ibotpeaches.github.io/Apktool/.

[3] AppPermissions.java. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/src/com/android/packageinstaller/permission/model/AppPermissions.java.

[4] Bug: 11242510. https://android.googlesource.com/platform/frameworks/base/+/3aeea1f.

[5] Bug: 33860747. https://android.googlesource.com/platform/frameworks/base/+/78efbc95412b8efa9a44d573f5767ae927927d48.

[6] Building Android. https://source.android.com/setup/build/building.

[7] Define a Custom App Permission . https://developer.android.com/guide/topics/permissions/defining.

[8] Give platform permissions a dummy group. https://android.googlesource.com/platform/frameworks/base/+/2a01ddbb4ea572ec82687dc0d9602eff36cc0886.

[9] Google Play Instant. https://developer.android.com/topic/google-play-instant.

[10] JPush. https://docs.jiguang.cn/en/jpush/guideline/intro/.

[11] PackageManager. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/services/core/java/com/android/server/pm/.

[12] PackageManagerService.java. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java.

[13] PermissionController. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/.

[14] Permissions Are Install-Order Dependent. https://issuetracker.google.com/issues/36941003.

[15] Permissions overview: Permission groups. https://developer.android.com/guide/topics/permissions/overview#perm-groups.

[16] Permissions overview: Protection levels. https://developer.android.com/guide/topics/permissions/overview#normal-dangerous.

[17] Remove grouping for platform permissions. https://android.googlesource.com/platform/frameworks/base/+/17eae45cf9a3948ed268e51bf13528ad82a465f0.

[18] Runtime Permissions: Defining custom permissions. https://source.android.com/devices/tech/config/runtime_perms#defining-custom-perms.

[19] Security Updates and Resources: Severity. https://source.android.com/security/overview/updates-resources#severity.

[20] Signing JAR Files. https://docs.oracle.com/javase/tutorial/deployment/jar/signing.html.

[21] Utils.java. https://cs.android.com/android/platform/superproject/+/android-10.0.0_r30:packages/apps/PermissionController/src/com/android/packageinstaller/permission/utils/Utils.java.

[22] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR), Austin, TX, USA, May 14-22, 2016*, 2016.

[23] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014*, 2014.

[24] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012*, 2012.

[25] 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 *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016*, 2016.

[26] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification," in *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, N. Bjørner and F. S. de Boer, Eds., 2015.

[27] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS), Chicago, Illinois, USA, October 4-8, 2010*, 2010.

[28] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft, "Exploring Decision Making with Android's Runtime Permission Dialogs using In-context Surveys," in *Proceedings of the 13th Symposium on Usable Privacy and Security (SOUPS), Santa Clara, CA, USA, July 12-14, 2017*, 2017.

[29] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. B. Srivastava, "ipShield: A Framework For Enforcing Context-Aware Privacy," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA, USA, April 2-4, 2014*, 2014.

[30] X. Chen, H. Huang, S. Zhu, Q. Li, and Q. Guan, "SweetDroid: Toward a Context-Sensitive Privacy Policy Enforcement Framework for Android OS," in *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society (WPES), Dallas, TX, USA, October 30 - November 3, 2017*, 2017.

[31] A. Dawoud and S. Bugiel, "DroidCap: OS Support for Capability-based Permissions in Android," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-27, 2019*, 2019.

[32] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. S. Wang, Z. Qian, and H. Chen, "revDroid: Code Analysis of the Side Effects after Dynamic Permission Revocation of Android Apps," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an, China, May 30 - June 3, 2016*, 2016.

[33] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS), Chicago, Illinois, USA, October 17-21, 2011*, 2011.

[34] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS), Washington, DC, USA, July 11-13, 2012*, 2012.

[35] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *Proceedings of the 20th USENIX Security Symposium (USENIX-SEC), San Francisco, CA, USA, August 8-12, 2011*, 2011.

[36] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and Enhancing Android's Permission System," in *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, 2012.

[37] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An Analysis of Pre-installed Android Software," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland), Virtual, May 18-20, 2020*, 2020.

[38] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. D. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *Proceedings of the 2nd Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Co-located with CCS 2012, Raleigh, NC, USA, October 19, 2012*, 2012.

[39] M. L. Murphy. (2014) Vulnerabilities with Custom Permissions. https://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html.

[40] N. Raval, A. Razeen, A. Machanavajjhala, L. P. Cox, and A. Warfield, "Permissions Plugins as Android Apps," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), Seoul, Republic of Korea, June 17-21, 2019*, 2019.

[41] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter, "Resolving the Predicament of Android Custom Permissions," in *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.

[42] G. S. Tuncay, J. Qian, and C. A. Gunter, "See No Evil: Phishing for Permissions with False Transparency," in *Proceedings of the 29th USENIX Security Symposium, (USENIX-SEC), Virtual, August 12-14, 2020*, 2020.

[43] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission Evolution in the Android Ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC), Orlando, FL, USA, December 3-7, 2012*, 2012.

[44] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android Permissions Remystified: A Field Study on Contextual Integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX-SEC), Washington, D.C., USA, August 12-14, 2015*, 2015.

[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 *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland), Berkeley, CA, USA, May 18-21, 2014*, 2014.
[46] W. Xu, F. Zhang, and S. Zhu, "Permlyzer: Analyzing Permission Usage in Android Applications," in *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, USA, November 4-7, 2013*, 2013.
[47] Z. Xu and S. Zhu, "SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY), San Antonio, TX, USA, March 2-4, 2015*, 2015.
[48] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013*, 2013.
[49] Y. Zhauniarovich and O. Gadyatskaya, "Small Changes, Big Changes: An Updated View on the Android Permission System," in *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, 2016.

APPENDIX

A. New System Permissions in Android 10

Note that, the permission prefixes (`android.permission.`) are omitted in the following lists.

New `normal` system permissions:
1. `REQUEST_PASSWORD_COMPLEXITY`
2. `USE_FULL_SCREEN_INTENT`
3. `CALL_COMPANION_APP`

New `dangerous` system permissions:
1. `ACTIVITY_RECOGNITION`
2. `ACCESS_BACKGROUND_LOCATION`
3. `ACCESS_MEDIA_LOCATION`

New `signature` system permissions:
1. `WRITE_DEVICE_CONFIG`
2. `MANAGE_ROLLBACKS`
3. `MANAGE_ACCESSIBILITY`
4. `START_ACTIVITIES_FROM_BACKGROUND`
5. `CONTROL_DISPLAY_COLOR_TRANSFORMS`
6. `CONTROL_KEYGUARD_SECURE_NOTIFICATIONS`
7. `MONITOR_DEFAULT_SMS_PACKAGE`
8. `POWER_SAVER`
9. `GET_RUNTIME_PERMISSIONS`
10. `LOCK_DEVICE`
11. `NETWORK_SCAN`
12. `SEND_DEVICE_CUSTOMIZATION_READY`
13. `BIND_CALL_REDIRECTION_SERVICE`
14. `BIND_PHONE_ACCOUNT_SUGGESTION_SERVICE`
15. `RESET_PASSWORD`
16. `NETWORK_SIGNAL_STRENGTH_WAKEUP`
17. `WRITE_SETTINGS_HOMEPAGE_DATA`
18. `MANAGE_DEBUGGING`
19. `REQUEST_INCIDENT_REPORT_APPROVAL`
20. `WRITE_OBB`
21. `INSTALL_DYNAMIC_SYSTEM`
22. `BIND_CONTENT_CAPTURE_SERVICE`
23. `com.qti.permission.DIAG`
24. `MODIFY_DEFAULT_AUDIO_EFFECTS`
25. `REQUEST_NOTIFICATION_ASSISTANT_SERVICE`
26. `REMOTE_DISPLAY_PROVIDER`
27. `SUBSTITUTE_SHARE_TARGET_APP_NAME_AND_ICON`
28. `WIFI_SET_DEVICE_MOBILITY_STATE`
29. `HANDLE_CALL_INTENT`
30. `INTERACT_ACROSS_PROFILES`
31. `WIFI_UPDATE_USABILITY_STATS_SCORE`
32. `CAPTURE_MEDIA_OUTPUT`
33. `NETWORK_CARRIER_PROVISIONING`
34. `BIND_EXPLICIT_HEALTH_CHECK_SERVICE`
35. `RECEIVE_DEVICE_CUSTOMIZATION_READY`
36. `AMBIENT_WALLPAPER`
37. `READ_DEVICE_CONFIG`
38. `ACCESS_SHARED_LIBRARIES`
39. `MANAGE_ROLE_HOLDERS`
40. `OBSERVE_ROLE_HOLDERS`
41. `START_VIEW_PERMISSION_USAGE`
42. `WHITELIST_RESTRICTED_PERMISSIONS`
43. `OPEN_ACCESSIBILITY_DETAILS_SETTINGS`
44. `ADJUST_RUNTIME_PERMISSIONS_POLICY`
45. `APPROVE_INCIDENT_REPORTS`
46. `MANAGE_APP_PREDICTIONS`
47. `SMS_FINANCIAL_TRANSACTIONS`
48. `CAMERA_OPEN_CLOSE_LISTENER`
49. `MANAGE_APPOPS`
50. `MANAGE_TEST_NETWORKS`
51. `GRANT_PROFILE_OWNER_DEVICE_IDS_ACCESS`
52. `BIND_ATTENTION_SERVICE`
53. `CONTROL_ALWAYS_ON_VPN`
54. `START_ACTIVITY_AS_CALLER`
55. `MONITOR_INPUT`
56. `MANAGE_DYNAMIC_SYSTEM`
57. `MANAGE_CONTENT_CAPTURE`
58. `MANAGE_WIFI_WHEN_WIRELESS_CONSENT_REQUIRED`
59. `OPEN_APP_OPEN_BY_DEFAULT_SETTINGS`
60. `PACKAGE_ROLLBACK_AGENT`
61. `BIND_CARRIER_MESSAGING_CLIENT_SERVICE`
62. `NETWORK_MANAGED_PROVISIONING`
63. `MANAGE_COMPANION_DEVICES`
64. `REVIEW_ACCESSIBILITY_SERVICES`
65. `USE_BIOMETRIC_INTERNAL`
66. `RESET_FACE_LOCKOUT`
67. `MANAGE_BIOMETRIC`
68. `MANAGE_BLUETOOTH_WHEN_WIRELESS_CONSENT_REQUIRED`
69. `MANAGE_CONTENT_SUGGESTIONS`
70. `BIND_CONTENT_SUGGESTIONS_SERVICE`
71. `BIND_AUGMENTED_AUTOFILL_SERVICE`
72. `MAINLINE_NETWORK_STACK`
73. `MANAGE_SENSOR_PRIVACY`
74. `BIND_FINANCIAL_SMS_SERVICE`
75. `TEST_MANAGE_ROLLBACKS`
76. `MANAGE_BIOMETRIC_DIALOG`
77. `READ_CLIPBOARD_IN_BACKGROUND`
78. `ENABLE_TEST_HARNESS_MODE`
79. `com.qti.permission.AUDIO`

80. com.qualcomm.qti.permission.
USE_QTI_TELEPHONY_SERVICE
81. com.qualcomm.qti.permission.
ACCESS_USER_AUTHENTICATION_APIS
82. com.android.permissioncontroller.permission.
MANAGE_ROLES_FROM_CONTROLLER

B. System Permission Groups in Android 10

1. android.permission-group.CONTACTS
2. android.permission-group.CALENDAR
3. android.permission-group.SMS
4. android.permission-group.STORAGE
5. android.permission-group.LOCATION
6. android.permission-group.CALL_LOG
7. android.permission-group.PHONE
8. android.permission-group.MICROPHONE
9. android.permission-group.ACTIVITY_RECOGNITION
10. android.permission-group.CAMERA
11. android.permission-group.SENSORS
12. android.permission-group.UNDEFINED