Custom Permission Misconfigurations in Android: A Large-Scale Security Analysis

Rui Li*, Wenrui Diao†, and Debin Gao*
*Singapore Management University
{ruili, dbgao}@smu.edu.sg
†School of Cyber Science and Technology, Shandong University
diaowenrui@link.cuhk.edu.hk

***Abstract*—Android's popularity is due to its openness and vast app ecosystem. Global developers can use Android Studio and rich Android APIs to create their apps. Within this ecosystem, Android permissions play a crucial role in managing access to resources, with system permissions controlled by system apps and custom permissions declared by third-party apps. However, the security of custom permissions has not received enough attention from the mobile security community, resulting in a lack of thorough evaluation of security practices for app developers using custom permissions. This study systematically evaluated the misconfiguration of custom permissions by Android app developers. It is based on ten configuration guidelines derived from the Android development documentation, OS source code, and related research papers to ensure proper functioning and adherence to best security practices of custom permissions. The study established the corresponding violation rules and built a dataset containing 174,740 APK files for large-scale measurement and analysis of guideline violations. The measurement results indicate that misconfiguration of custom permissions by Android app developers is quite common, with approximately 29.02% of the 92,461 apps involving custom permissions having configuration guideline violations. The two most common errors in custom permission configuration are 1) putting custom permissions into a defective custom group and 2) protecting components with undeclared custom permissions. Such misconfigurations can lead to various issues, including private app data leaks, app installation failures, or incomplete implementation of app functions.**

I. Introduction

Android's popularity is attributed to its openness, which has led to a diverse and extensive app ecosystem. Android provides official app development tools such as Android Studio and a rich set of Android APIs to empower developers to create their apps. Google offers detailed development documentation [3] to assist developers in app building, including guidance on configuring the app's manifest file, implementing app components, and more. Additionally, Google outlines best security practices [20] in the development documentation to ensure the security of apps, covering aspects such as app data storage, app permissions usage, and inter-app interactions. For apps published on Google Play, Google provides the App Security Improvement Program (ASIP) to support developers in building more secure apps [5].

Despite previous research [23], [26], [33] showing that misconfiguration issues in apps remain widespread, developers continue to make errors when configuring the app manifest file, such as incorrectly placing elements and attributes or misspelling them. Among the security issues caused by these configuration errors, permission-related problems are the most common, including over-privileged requests and improper component protection [30]. Since the scope of custom permissions is theoretically limited to the third-party app itself, their security has been largely overlooked by the Android security community [27]. Consequently, the security practices of app developers using custom permissions have yet to be systematically evaluated. Although prior research on manifest file misconfigurations [33], [26] has partially addressed issues with custom permissions, it has not comprehensively combined explanations from the Android development documentation with the OS source code to study how developers misconfigure custom permissions empirically.

Our work. This work systematically evaluated the misconfigurations of custom permissions by Android app developers. Specifically, this work first comprehensively analyzed the Android development documentation and OS source code and combined it with previous related research to investigate the requirements, limitations, and best security practices related to custom permission configuration. From this investigation, ten **C**ustom **P**ermission **C**onfiguration **G**uidelines (**CPC-Guideline**s) were derived. These guidelines cover the declaration and use of custom permissions (see Section III for details). Subsequently, this work collected 174,740 APK files from Google Play and third-party markets for large-scale measurement, detecting violations of these ten CPC-Guidelines and analyzing their potential causes and security implications. Specifically, this work aims to answer the following three Research Questions (RQs):

> **RQ#1:** ***What are the practical security implications of CPC-Guideline violations?***
>
> **RQ#2:** ***What is the current state of CPC-Guideline violations in the wild?***
>
> **RQ#3:** ***What are the differences in CPC-Guideline violations between Google Play and third-party market apps?***

Our measurement and analysis results show that violating CPC-Guidelines can lead to sensitive data leaks, app installation failures, and reduced app functionality. Attackers may exploit these issues by declaring or requesting improperly

configured custom permissions to access protected resources. Among the 92,461 apps involving custom permissions, 29.02% have violations, often due to reliance on third-party app templates or development guidelines, leading to misconfiguration to propagate. The violation rate is lower for Google Play apps (20.55%) compared to third-party market apps (37.95%).

Contributions. The contributions of this work are as follows.

- ***Systematic configuration guidelines summary***. By comprehensively investigating the Android development documentation, Android OS source code, and research related to app configuration, we systematically summarized custom permission configuration guidelines that ensure proper functioning and adherence to best security practices of custom permissions.
- ***Large-scale security assessment***. We constructed a dataset of over 170K APK files to perform a large-scale measurement study on Android app developers' misconfiguration of custom permissions.

This study revealed how app developers misconfigured custom permissions, leading to potential security risks. It aims to offer better guidance on properly implementing custom permissions and suggestions for improving the design of Android development documentation and Android Studio.

II. Background

In this section, we provide the necessary background of the configuration related to custom permissions in apps.

To protect user data, Android has implemented various security mechanisms, with permissions being a core feature. Permissions are defined in the app's manifest file and enforced by the system. Android has two types of permissions: *system permissions*, declared by system apps to protect system resources, and *custom permissions*, declared by third-party apps to protect their resources for inter-app sharing [9]. System apps are pre-installed, while third-party apps are built by developers worldwide and published on app markets. For third-party apps, custom permission configuration in the manifest files involves both declaration and use. Table I lists the manifest elements and attributes related to custom permissions.

A. Custom Permission Declaration

As shown in Table I, three elements are relevant to the custom permission declaration.

Define a custom permission. An app declares a custom permission in its manifest file using the `<permission>` element [16]. The common attributes of this element include:

- `android:name`: specify the custom permission's name.
- `android:protectionLevel`: specify the custom permission's protection level. There are three main levels for a custom permission: `normal`, `signature`, and `dangerous`.
- `android:permissionGroup`: put a custom permission into a permission group.
- `android:description`: explain the usage purpose of a custom permission.

Define a custom group. Custom permissions can be put into a custom group. When declaring a custom group using the `<permission-group>` element, the app can specify its name and explain the functionality covered by this group through `android:name` and `android:description`, respectively [17].

Define a permission tree. The `<permission-tree>` element allows an app to declare a namespace for custom permissions. The `android:name` attribute specifies the base name for all permissions in the tree [18].

B. Custom Permission Use

There are nine elements related to the use of custom permissions, as listed in Table I.

Protecting app components. An app can declare four types of components: Activity, Broadcast Receiver, Service, and Content Provider, through the `<activity>/<activity-alias>`, `<receiver>`, `<service>`, and `<provider>` elements, respectively. App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter the app. Thus, to restrict access to the app, it can use custom permissions to protect its components by specifying the `android:permission` attribute of the component elements. These elements are contained in the `<application>` element. The app can also specify `android:permission` of the `<application>` element to protect all declared components. For the `<provider>` element, extra `<path-permission>` sub-elements, `android:readPermssion` and `android:writePermission` attributes can be set to perform the fine-grained access control of the Content Provider [4].

Request custom permissions. If an app needs to access the feature protected by a custom permission, it should request this permission in its manifest file through the `<uses-permission>/<uses-permission-sdk-23>` element. The latter element specifies that the app wants a particular permission, but only if it is installed on a device running Android 6.0 (API level 23) or higher [21].

III. Custom Permission Configuration Guidelines

We systematically examined Android development documentation, OS design, and relevant research to formulate **C**ustom **P**ermission **C**onfiguration **G**uidelines (**CPC-Guideline**s) that ensure both proper functionality and security of custom permissions. These guidelines cover the declaration and use of custom permissions.

- *Android development documentation*. We searched the official documentation using keywords like "custom permission", "security", and "best practice", extracting relevant guidelines for custom permission configuration.
- *Android OS source code*. We analyzed the implementations of the app installation process and identified a preparation stage [19] where the system validates the app's manifest file. Restrictions, such as preventing the declaration of existing permissions, are enforced, and violations result in installation failures.

TABLE I: Custom permission-related elements in app manifest files.

Usage	Element	Parts of Attributes†	Contained In
Permission Declaration	<permission>	name, protectionLevel, permissionGroup, description, ...	<manifest>
	<permission-group>	name, label, description, ...	
	<permission-tree>	name, label, ...	
Permission Use	<uses-permission>	name, maxSdkVersion	<manifest>
	<uses-permission-sdk-23>	name, maxSdkVersion	
	<application>	enabled, permission, ...	
	<activity>	name, enabled, permission, exported,...	<application>
	<activity-alias>	name, targetActivity, enabled, exported, permission, ...	
	<receiver>	name, enabled, exported, permission, ...	
	<service>	name, enabled, exported, permission, ...	
	<provider>	name, authorities,enabled, exported, permission, readPermission, writePermission, ...	
	<path-permission>	path, permission, readPermission, writePermission, ...	<provider>

†: The attributes omit the "android:" prefix.

- *Related research*. We surveyed prior studies on Android app misconfigurations, summarizing key recommendations for custom permission settings.

A. Custom Permission Definition

The configuration guidelines for declaring custom permissions include:

Guideline#1: ***Custom permissions should be declared effectively.***

The app's manifest file is a hierarchical XML file with specific requirements for element placement [4]. Table I outlines the placing rules for common manifest elements. Errors such as misplacing elements or misspelling element types prevent the system from correctly parsing these declarations, rendering them ineffective. Previous research [33] shows such errors are common. To ensure custom permission declarations function properly, developers must avoid these mistakes.

Guideline#2: ***Custom permission names should be prefixed with the app's package name.***

Android prohibits apps from declaring existing system permissions unless they share the same signature as the original declaring app. Violations will result in blocked app installations or ignored permission declarations. To avoid naming conflicts, Android recommends prefixing custom permission names with the app's package name [10].

Guideline#3: ***The protection levels of custom permissions should be set to `signature`.***

The signature permission is granted based on the app's signature without user intervention, providing a seamless experience. It restricts access to apps with the same signature, ensuring stricter security. Therefore, Android recommends setting the custom permission to signature-level [20].

Guideline#4: ***`Dangerous`-level custom permissions should have usage descriptions.***

The dangerous-level permission requires user approval during the app running. When requested, the system presents a prompt with contextual information based on the usage description of the dangerous permission or its belonged group [16], [13]. If no description is provided, the prompt will display: "*Allow app to perform an unknown action?*", which may lead users to deny the permission request. To avoid this, Android advises developers to set clear usage descriptions for dangerous custom permissions [20].

Guideline#5: ***When placing custom permissions in a custom group, this group should have a good definition.***

A *good* custom group declaration means that this declaration exists, and 1) the group name should be started with the app's package name; 2) the group's usage description should be provided for dangerous custom permissions.

Previous research [28] shows that placing permissions in undefined groups can cause dormant permission issues. To address this, Android 12 prohibits permissions in undefined groups. Additionally, if permission is in a custom group, the app must share the same signature as the group owner [6]. To avoid installation failures, developers should define custom groups when using them. Besides, similar to Guideline#2, group names also have name conflicts. As mentioned in Guideline#4, the group's usage description would be displayed to users during dangerous permission requests.

Guideline#6: ***Do not declare twin custom permissions.***

Previous research [29] identifies twin custom permissions as an attack vector for the Evil Twins vulnerability. To mitigate this, Android 12 prohibits the installation of apps declaring twin custom permissions [7].

Guideline#7: ***Declare the minimum custom permissions necessary to meet security requirements.***

Android offers pre-defined system permissions that typically meet most apps' security needs without requiring new permission declarations [20].

B. Custom Permission Use

The configuration guidelines involved in the use of custom permissions include:

Guideline#8: ***Custom permissions should be used effectively.***

As with Guideline#1, developers should avoid spelling errors and incorrect placement of elements and attributes to ensure proper use of custom permissions.

Guideline#9: ***Custom permissions used to protect components should be defined.***

If custom permissions are used to protect components but are undefined, it can lead to dangling attribute references, which pose security risks [22]. Attackers can exploit these dangling references. Therefore, developers must define custom permissions to prevent such vulnerabilities.

Guideline#10: ***Do not use custom permissions to protect inaccessible components.***

If a component is inaccessible, permission protection is unnecessary, as other apps cannot access it. Android advises adding permissions only when a component becomes public, to avoid later removal that could affect users [20]. Using custom permissions for inaccessible components may suggest a misunderstanding or over-caution by the developer [26].

IV. Methodology

This section outlines the methodology for evaluating CPC-Guideline violations, as depicted in Figure 1. The process involves the following steps:

- **Violation rule establishment.** Following the summarized CPC-Guidelines, rules were defined to detect violations.
- **APK dataset construction.** A large dataset of APKs from Google Play and third-party markets was constructed.
- **Configuration extraction.** Manifest files were parsed to extract relevant data for further measurement and analysis.
- **Measurement and analysis.** We statistically examine violations, analyzing their causes and security implications.

A. Violation Rule Establishment

For each guideline presented in Section III, we established corresponding violation rules, where Rule#N corresponds to Guideline#N.

⚠ **Rule#1:** ***The app contains an ineffective custom permission declaration.*** Specific violations include:

- *Typo in element type (Elem-typo).* The developer misspelled `<permission>`.
- *Incorrect placement of element (Elem-misplaced).* The developer incorrectly placed the `<permission>` element. The correct location is listed in Table I.
- *Missing of element name (Name-missing).* The developer did not set `android:name` of the `<permission>` element.

⚠ **Rule#2:** ***The custom permission name is not prefixed with the app's package name.***

⚠ **Rule#3:** ***The protection level of the custom permission is not set to*** `signature`. Specific violations include:

- *No protection level set (No-protect).* The protection level (`android:protectionLevel`) is not explicitly specified, defaulting to `normal`.
- *Inappropriate protection level (Not-signature).* The protection level is set to something other than `signature`, such as `dangerous`.

⚠ **Rule#4:** ***The usage description*** (`android:description`) ***of the*** `dangerous`***-level custom permission is missing.***

⚠ **Rule#5:** ***The app assigns custom permissions to a defective custom group.*** Specific violations include:

- *Undefined group (No-defCg).* The developer did not declare the group by the `<permission-group>` element.
- *No prefixing group name with the app's package name (No-prefixCg).* The group name does not start with the app's package name.
- *Group's usage description is missing (No-desCg).* When defining the belonged group of `dangerous` custom permissions, the developer did not specify `android:description` of the `<permission-group>` element.

⚠ **Rule#6:** ***The app declares twin custom permissions.*** Specific violations include:

- *Twin custom permission declarations (Twin-perms):* The developer declared two custom permissions with the same name but different attributes.
- *Duplicate custom permission and permission tree (Twin-perm&tree):* The developer declared a custom permission and a permission tree with the same name.

⚠ **Rule#7:** ***The app declares unnecessary custom permissions.*** Specifically, these permissions are neither requested by the `<uses-permission>`/`<uses-permission-sdk-23>` element nor used to protect any components through the permission attributes[1].

⚠ **Rule#8:** ***The app uses custom permissions ineffectively.*** Specific violations include:

- *Typo in permission attribute (Attr-typo).* The developer misspelled permission attributes.
- *Typo in element type (Elem-typo).* The developer misspelled `<uses-permission>`, `<uses-permission-sdk-23>`, or permission attribute-supported elements (as listed in Table I) which are protected by custom permissions.

[1]Permission attribute: `android:permission`, `android:readPermission`, or `android:writePermission`.

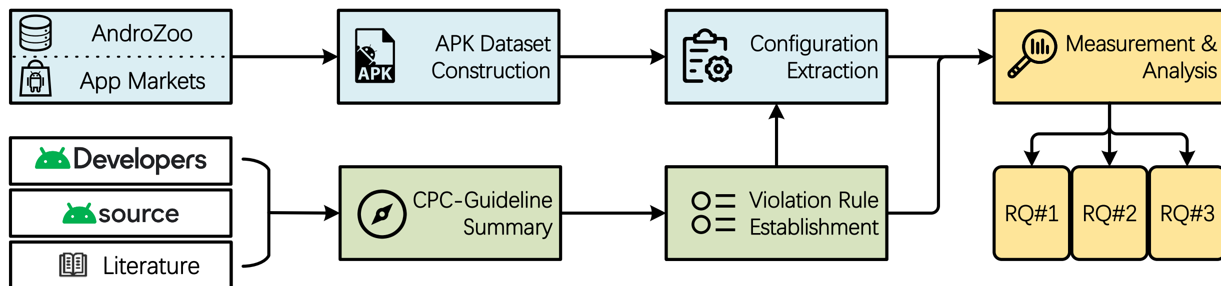

Fig. 1: Overview of measurement methodology.

- *Incorrect placement of permission attribute (Attr-misplaced).* The developer placed permission attributes in unsupported elements.
- *Incorrect placement of element (Elem-misplaced).* The developer misplaced `<uses-permission>`/`<uses-permission-sdk-23>` elements or permission attribute-supported elements which are protected by custom permissions. The correct locations are listed in Table I.

⚠ **Rule#9:** ***The app uses undefined custom permissions to protect components. That is, the developer did not declare these permissions using the*** `<permission>` ***elements.***

⚠ **Rule#10:** ***The app uses custom permissions to protect inaccessible components.***

A component is inaccessible when the developer specified its `android:enable` to `false` or specified `android:enable` to `true` and `android:exported` to `false`. If the developer did not set these two attributes, the default value of `android:enable` is `true`. For `android:exported`, the default value is `true` of `<activity>`/`<activity-alias>`, `<service>`, and `<receiver>` elements with intent filters. For `<provider>` elements, if the app's target SDK $\geq$ 17, the default value is `false`; otherwise, it is `true`.

Remarks. To minimize redundant violation statistics, we impose two restrictions: 1) since Guideline#1 covers ineffective custom permission declarations, only violations of Guideline#2~#7, #9 are considered if custom permissions are declared effectively; 2) since Guideline#8 covers ineffective custom permission use, only violations of Guidelines#7, #9, #10 are considered if custom permissions are used effectively.

B. APK Dataset Construction

We collected APK files from the Google Play Store and third-party markets. Since bulk downloading from Google Play is restricted, we used the AndroZoo dataset [24] for Google Play APK files. For third-party markets, we developed an automated crawler to collect APK files from four markets: 2265, Lenovo, Leyou, and Mdpda. In total, we gathered 109,240 apps from Google Play and 65,699 from third-party markets. After deduplication and filtering out invalid APK files[2], we obtained 109,239 unique Google Play apps and 65,501 unique third-party market apps, totaling 174,740 apps.

[2]An APK file is considered valid if it has been signed and its manifest file can be successfully parsed.

C. App Configuration Extraction

After constructing the APK dataset, we extracted app configuration data from the manifest files of each APK. This data includes basic app information and details on custom permission declaration and use.

Basic information. The extracted app basic information includes: 1) package name: the `package` attribute in the `<manifest>` element; 2) version: the `android:versionCode` and `android:versionName` attributes in the `<manifest>` element; 3) target SDK: the `android:targetSdkVersion` attribute in the `<uses-sdk>` element; 4) signing information: the certificate's subject, issuer, and SHA1 digest. This information is used to detect and analyze CPC-Guideline violations.

Permission declaration and use. We extracted two types of custom permission configuration data based on violation rules, considering spelling errors and incorrect placements:

- *Custom permission declaration.* Data from `<permission>`, `<permission-group>`, and `<permission-tree>` elements, or similar elements.
- *Custom permission use.* Data from 1) `<uses-permission>` and `<uses-permission-sdk-23>` elements or similar elements; 2) elements containing permission attributes or similar attributes.

We ensured that the extracted permissions and groups used were custom, not system.

Implementation. Androguard [2] was used to unpack APKs and extract manifest data. For custom permission configuration data, we integrated the Python fuzzy matching library, Fuzzywuzzy [11], to match relevant configurations. Additionally, we collected system permission and group declarations from Android 2.3.3~14 images (API levels 10~34) to identify system permissions and groups used by apps.

D. Measurement and Analysis

After extracting configuration data from each app in the APK dataset, we detect CPC-Guideline violations and analyze their causes and security implications. This study addresses the following three Research Questions (RQ):

> **RQ#1:** ***What are the practical security implications of CPC-Guideline violations?***

❸ What are the practical security risks of violating the extracted CPC-Guidelines? Can attackers exploit these violations for practical attacks?

RQ#2: ***What is the current state of CPC-Guideline violations in the wild?***

❸ Among the 174,740 collected apps, how many violate CPC-Guidelines, and what is the violation status of each Guideline?

RQ#3: ***What are the differences in CPC-Guideline violations between Google Play and third-party market apps?***

❸ Section I mentions that Android's App Security Improvement Program (ASIP) helps Google Play developers create more secure apps [5]. Does ASIP result in fewer violations for Google Play apps than third-party market apps?

V. Security Analysis And Measurement

Following the methodology in Section IV, we analyzed the security implications of CPC-Guidelines and conducted a large-scale measurement based on the constructed APK dataset. This section summarizes the results and addresses the three research questions outlined in Section IV-D.

A. RQ#1: What are the practical security implications of CPC-Guideline violations?

Guideline#1: ***Custom permissions should be declared effectively.*** Violations of Guideline#1 can cause custom permission declarations to fail. The protection these permissions provide will be ineffective, leading to sensitive data leaks.

Guideline#2: ***Custom permission names should be prefixed with the app's package name.*** Violations of Guideline#2 may result in different apps with different package names declaring the same custom permissions. If apps have different signatures, only one can be installed, leading to other installation failures. If signatures match, the installation order determines the permission owner, causing some permission protections for apps' resources not as desired, which may lead to data leaks.

Guideline#3: ***The protection levels of custom permissions should be set to `signature`.*** Violations of Guideline#3 can result in insufficient protection. The `normal`-level custom permission is granted automatically, causing the protection for sensitive data to be weak. Using the `dangerous` custom permission requires user consent and risks being denied, potentially impacting app functionality.

Guideline#4: ***`Dangerous`-level custom permissions should have usage descriptions.*** Violating Guideline#4 may lead to users rejecting permission requests due to unclear purposes, causing the app to malfunction and negatively impacting the user experience.

Guideline#5: ***When placing custom permissions in a custom group, this group should have a good definition.*** Violating Guideline#5 can cause the custom permission to be put into a defective custom group, like a dormant group discussed in [28], and further leading to installation failure on Android 12 or higher or sensitive data leaks on Android 11 or lower by exploiting the dormant group vulnerability.

Guideline#6: ***Do not declare twin custom permissions.*** Violating Guideline#6 may prevent app installation on Android 12 or higher. On older versions, it could cause exploitation by attackers using the Evil Twins vulnerability [29], leading to unauthorized access to sensitive resources.

Guideline#7: ***Declare the minimum custom permissions necessary to meet security requirements.*** Violating Guideline#7 may result in the dangling use of declared permissions, leading to failed protection and exposure of sensitive resources. Excessive permissions may also increase the risk of name conflicts, hindering app installation.

Guideline#8: ***Custom permissions should be used effectively.*** Violating Guideline#8 can render custom permission use ineffective, causing permission request failure and impacting app functionality. It may also result in unprotected components, leading to data leaks.

Guideline#9: ***Custom permissions used to protect components should be defined.*** Violating Guideline#9 may lead to dangling references in component permissions. If attackers declare the dangling custom permissions, they can gain control of these permission and access the protected components.

Guideline#10: ***Do not use custom permissions to protect inaccessible components.*** While violating Guideline#10 may not have direct security implications, as noted in Section III-B, removing redundant component permissions in future updates could negatively impact existing users.

Answer to RQ#1

Violating CPC-Guidelines can lead to the leakage of sensitive user information, affect app installation, and impair the app's functionality. Attackers can exploit these violations by declaring or requesting misconfigured custom permissions in the target app.

B. RQ#2: What is the current state of CPC-Guideline violations in the wild?

Among the 174,740 apps collected, 92,461 (52.91%) involve (i.e., declare or use) custom permissions and 26,828 (29.02%) of these apps violate CPC-Guidelines. This highlights the issue of custom permission misconfiguration is common.

The violation status for each guideline is detailed in Table II. Guideline#5 and Guideline#9 have the highest violation rates, around 80%, indicating two common custom permission misconfigurations: 1) placing custom permissions in defective groups and 2) neglecting to declare custom permissions when protecting components. In contrast, Guideline#6 has the fewest violations, likely due to Android Studio's error detection for twin custom permissions, preventing the app from being built.

Guideline#1: ***Custom permissions should be declared effectively.*** A total of 65,668 apps make 112,229 custom permission declarations, with 167 (0.15%) violating Guideline#1. Table III details these violations, showing that the majority (98.20%) result from Elem-misplaced, where developers incorrectly placed

TABLE II: Violation statistics of each guideline.

Guideline#	1	2	3	4	5	6	7	8	9	10
App Involved Amount†	65,668	65,540	65,540	280	558	65,540	65,540	91,482	16,091	16,091
App Violated Amount‡	160	11,020	6,781	99	453	48	4,836	1,255	12,747	5,442
Violation Rate§	**0.24%**	**16.81%**	**10.35%**	**35.36%**	**81.18%**	**0.07%**	**7.38%**	**1.37%**	**79.22%**	**33.82%**

†: The amount of apps involving CPC-Guideline#. ‡: The amount of apps violating CPC-Guideline#.
§: The violation rate of Guideline# = the amount of apps violating Guideline# / the amount of apps involving Guideline#.

TABLE III: Violation statistics of Guideline#1.

Rule#1	Elem-typo	Elem-misplaced	Name-missing
Amount†	3	164	0
Percentage	1.80%	98.20%	0.00%

†: The amount of ineffective custom permission declarations.

TABLE IV: Violation statistics of Guideline#3.

Rule#3	No-protect	Not-signature
Amount†	3,424	5,327
Percentage	39.13%	60.87%

†: The amount of effective custom permission declarations.

TABLE V: Protection levels triggering Rule#3: Not-signature.

Protection Level	`normal`	`dangerous`	`instant`
Amount†	4,839	486	2
Percentage	90.84%	9.12%	0.04%

†: The amount of effective custom permission declarations.

the `<permission>` element inside the `<application>` element. Further analysis revealed that 91.02% of these violations are linked to third-party push services, with 45.51% associated with apps having the package name prefix `com.appbyme.app`. These apps were found to originate from the DIY mobile app platform Appbyme, suggesting that developers likely propagated this misplacement when using app templates provided by the platform to integrate third-party push services.

Guideline#2: ***Custom permission names should be prefixed with the app's package name.*** Among 112,062 effectively declared custom permissions, with 13,902 (12.41%) violating Guideline#2. These violations fall into two primary categories:

- *Third-party service specific.* Custom permissions are primarily used for third-party services [28], which often require particular permission names declared in the manifest. For instance, 3,914 (28.15%) custom permissions related to the message push service provided by GeTui [12], which requires permission names like `getui.permission.GetuiService.{package name}`.
- *Duplicate with system permission names.* There are 3,845 (27.66%) custom permissions sharing names with system permissions. Such permission declarations will be ignored by Android.

Guideline#3: ***The protection levels of custom permissions should be set to `signature`.*** Among the 112,062 effective custom permissions, 8,751 (7.81%) violated Guideline#3. Table IV details these violations, while Table V shows the protection level distribution of the 5,327 custom permissions that trigger Rule#3: Not-signature, with `normal` being the most common. As noted in Section IV-A, if a permission's protection level is not explicitly set, it defaults to `normal`. Consequently, 8,263 (94.42%) out of the 8,751 custom permissions violating Rule#3 would be parsed as `normal` ones. Further analysis revealed two main reasons for this phenomenon:

- *Apps come from third-party templates.* Developers use third-party templates or open-source frameworks for app development, propagating `normal` custom permissions. For instance, 98 `normal` custom permissions were found in apps prefixed with `com.wta.NewCloudApp.jiuwei`, all from the newCloudApp framework [14], which is used for remote operations via the Newland IoT Cloud Platform.
- *Apps are developed by the same company.* Apps from the same company are often developed using proprietary templates, leading to the widespread use of `normal` custom permissions. For instance, 204 custom permissions without protection levels (default to `normal`) were found in apps from Sinyee Inc, all sharing the same signature certificate.

Guideline#4: ***`Dangerous`-level custom permissions should have usage descriptions.*** There are 486 effective `dangerous`-level custom permissions, with 160 (32.92%) violating Guideline#4. This may stem from unfamiliarity with Android's group-based runtime permission management.

Guideline#5: ***When placing custom permissions in a custom group, this group should have a good definition.*** Of the 112,062 effective custom permissions, 615 (0.55%) are assigned to custom groups, with 491 (79.84%) violating Guideline#5. Table VI details these violations. Although custom groups are rarely used, it is very common for developers not to define these groups. Further analysis revealed that 399 (81.26%) of the violations involve group names ending with `andpermission`. AndPermission is a third-party open-source framework for permission management. We found that AndPermission's manifest file puts the custom permission into an undefined custom group named `${applicationId}.andpermission` [1]. Thus, the primary reason for these violations is the integration of third-party frameworks that propagate misconfiguration.

Guideline#6: ***Do not declare twin custom permissions.*** Among 65,540 apps with effective custom permissions, 48 (0.07%) violate Guideline#6 by declaring 48 pairs of twin

TABLE VI: Violation statistics of Guideline#5.

Rule#5	No-defCg	No-prefixCg	No-desCg
Amount†	441	45	5
Percentage	89.82%	9.16%	1.02%

†: The amount of effective custom permission declarations.

TABLE VII: Violation statistics of Guideline#6.

Rule#6	Twin-perms	Twin-perm&tree
Amount†	47	1
Percentage	97.92%	2.08%

†: The amount of twin custom permission pairs.

custom permissions[3]. Violations of Guideline#6 are rare, which may be due to the nonsupport of Android Studio to build apps with twin declarations. The developers need to build them through APK repackaging [29]. Table VII details these violations. Majority violations trigger Twin-perms because only five apps declare both permission trees and custom permissions.

Guideline#7: ***Declare the minimum custom permissions necessary to meet security requirements.*** Of the 112,062 effective custom permissions, 5,249 (4.68%) unused ones violate Guideline#7, and 681 (12.97%) are due to misspellings in the custom permission names or misplacing of the `<uses-permission>` element when using them. For the remaining 4,568 (87.03%) really unused permissions, our further analysis showed that the main reason for these violations is that apps are developed by the same subject (having the same certificate subject), causing the misconfiguration to propagate.

Guideline#8: ***Custom permissions should be used effectively.*** A total of 91,482 apps use custom permissions 649,571 times. Among these, 1,338 (0.21%) use instances violate Guideline#8, as detailed in Table VIII. The majority (82.59%) violations stem from Attr-misplaced, where most developers misplaced `android:permission` inside the `<action>` element. This type of element specifies the generic action components can respond to [8]. Therefore, it suggests developers intended to protect components with custom permissions but were unfamiliar with Android's permission enforcement rules.

Guideline#9: ***Custom permissions used to protect components should be defined.*** A total of 16,091 apps effectively use custom permissions 82,991 times to protect 74,653 components. Among these, 35,164 (42.37%) use instances violate Guideline#9 due to permission undefined, impacting 34,844 components, and 1,290 (3.67%) violations are due to misspellings in the custom permission names or the misplacing of the `<permission>` element. For the remaining 33,874 (96.33%) use instances of really undefined custom permissions, our further investigation identified two main reasons for these violations:

[3]To avoid double-counting, in an app, all custom permissions with the same name but different attributes, or all custom permissions and permission trees with the same name, are counted as one pair of twin custom permissions.

TABLE VIII: Violation statistics of Guideline#8.

Rule#8	Attr-typo	Elem-typo	Attr-misplaced	Elem-misplaced
Amount†	1	23	1,105	209
Percentage	0.07%	1.72%	82.59%	15.62%

†: The amount of ineffective custom permission use.

- *Apps integrate third-party services or frameworks.* Some third-party services or frameworks provide app development guidelines containing custom permission misconfiguration. For instance, 11,738 (34.65%) custom permission use instances related to the OPPO Push service. OPPO Push's official document instructs developers to use undefined custom permission to protect components [15].
- *Developers are unfamiliar with permission enforcement rules.* Around 1,417 (4.18%) used permissions have names like `normal`, `signature`, `False`, `True`, and `FriendsOnly`. These strings indicate that developers intended to protect components but were unfamiliar with the permission enforcement rules against the components.

Guideline#10: ***Do not use custom permissions to protect inaccessible components.*** Of the 82,991 effectively custom permission use instances to protect components, 18,600 (22.41%) violate Guideline#10, involving 18,237 inaccessible components. Further investigation showed that the main reason for these violations is that apps are developed by the same company. Simultaneously violating Guideline#9 and #10 – using undeclared custom permissions to protect inaccessible components – reflects a misunderstanding of the Android permission model. We identified 4,257 apps that have this issue with various signatures, indicating it is widespread.

Answer to RQ#2

Of the 174,740 collected apps, 52.91% declare or use custom permission, and 29.02% of them have at least one CPC-Guideline violation. Guideline#5 and #9 have the highest violation rates, indicating developers commonly 1) place custom permissions in defective groups and 2) protect components with undefined custom permissions. These violations often result from using third-party app templates or service guidelines for app development.

C. RQ#3: What are the differences in CPC-Guideline violations between Google Play and third-party market apps?

Among the 109,239 Google Play apps collected, 47,498 (43.48%) involve the declaration and use of custom permissions, with 9,763 (20.55%) having CPC-Guideline violations. In contrast, among the 65,501 third-party market apps collected, 44,963 (68.64%) involve custom permissions, with 17,065 (37.95%) having violations.

From the perspective of the guidelines, Table IX presents the violations of each guideline in Google Play and third-party market apps. The most significant difference between them is in Guideline#2, indicating that Google Play app developers

TABLE IX: Violation statistics of each guideline in Google Play and third-party market apps.

Guideline#		1	2	3	4	5	6	7	8	9	10
Violation Rate§	Google Play	0.05%	8.70%	4.60%	34.05%	71.01%	0.13%	5.38%	1.27%	74.95%	32.73%
	Third-party Market	0.45%	25.39%	16.42%	37.89%	82.62%	0.01%	9.49%	1.48%	82.06%	34.54%

§: The violation rate of Guideline# = the amount of apps violating Guideline# / the amount of apps involving Guideline#.

are more likely to define custom permissions starting with the package names. This may be because third-party market app developers' preference for using third-party services results in declaring more service-specific permission names violating Guideline#2, as discussed in Section V-B.

From the perspective of the apps, Figure 2 illustrates the number of guidelines violated by Google Play and third-party market apps, respectively. It shows that more than half (64.30%) of Google Play apps violate only one guideline.

Overall, the violation state for Google Play apps is slighter than that for third-party market apps. Based on AISP, Android provides better security assurance for Google Play apps.

Answer to RQ#3

Among Google Play apps involving custom permissions, 20.55% have guideline violations, compared to 37.95% for third-party market apps. It shows that Google Play apps perform better in custom permission configuration, with the most notable difference in violations of Guideline#2.

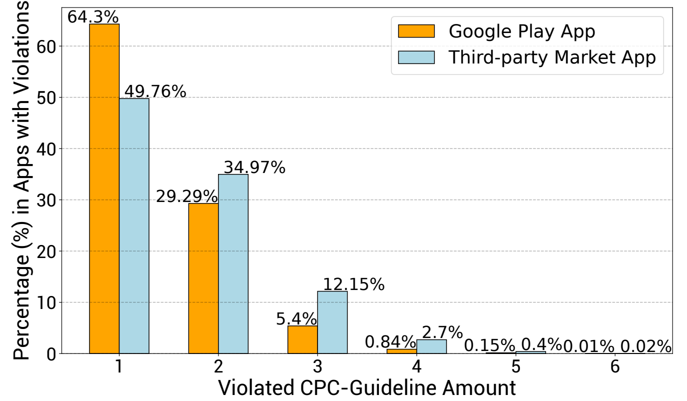

Fig. 2: Statistics by the number of violated guidelines in Google Play and third-party market apps with violations.

VI. Mitigation Measures

To address custom permission misconfigurations, we propose several mitigation measures.

For Google. Google should consider the following measures to improve the security and correctness of custom permission implementations in Android apps.

- *Provide more precise and comprehensive guidelines on custom permission security in the Android documentation.* Information on custom permissions in the current Android documentation is scattered across various sections, such as the manifest introduction [4] and compatibility documentation [6]. Some security practices, like Guideline#6, are undocumented. A more systematic guide documentation would help developers use custom permissions securely.
- *Enhance inspection mechanisms in Android Studio.* While Android Studio checks for errors like misspellings, it lacks prompts for best practices related to custom permissions, such as those in Guideline#2~#5. Integrating comprehensive checks combined with the documentation would help developers avoid misconfigurations.

For Android app developers. Android app developers are encouraged to adopt the following practices to reduce the risk of custom permission misconfigurations.

- *Regularly review the Android documentation.* Reviewing the Android documentation regularly helps developers stay informed about the latest security requirements and make timely adjustments to their apps.
- *Inspect third-party app templates.* When using third-party templates, developers should check them for custom permission misconfigurations and make necessary corrections.
- *Use the latest version of Android Studio.* With each update, Android Studio provides more comprehensive checks. Using the latest version helps avoid basic configuration errors like misspellings and incorrect placements.

VII. Related Work

In this section, we review the related work on Android custom permissions and app misconfigurations.

Custom permissions. Li et al. [27] systematically evaluated the design and implementation of Android custom permissions. The authors designed a tool called CuperFuzzer to detect vulnerabilities, which revealed severe design flaws in the Android permission framework that could allow malicious apps to gain unauthorized access to system resources. They also identified the Evil Twin flaw [29] in the Android manifest processing procedure, which can be exploited through twin custom permission elements to achieve privilege escalation. Tuncay et al. [31] identified vulnerabilities arising from interactions between system and custom permissions and proposed a modular design called Cusper to address these issues. Unlike the aforementioned studies, our work focuses on misconfigurations by app developers rather than design and implementation flaws in the Android OS.

App misconfigurations. Yang et al. [33] developed ManiScope to detect misconfigurations in Android manifest files. Analyzing over 2.5 million apps, they found misconfigurations in over 33% of Google Play and 35% of Samsung pre-installed apps, many of which pose serious security risks like data leaks

and component hijacking. Jha et al. [26] analyzed 13,483 Android apps to identify common developer mistakes in Android manifest files, uncovering 59,547 configuration errors across 11,110 apps using a rule-based static analysis tool. Han et al. [25] developed SADroid to detect misconfiguration vulnerabilities, highlighting the security risks due to Android permission model weaknesses. Scoccia et al. [30] studied 574 GitHub repositories and found that permission-related issues often persist for years. Yang et al. [32] analyzed 251,749 apps, revealing developers inconsistently following Google's security guidelines. Unlike the previous studies, our work focuses on the misconfiguration of custom permissions within apps, a topic that has not been systematically explored before.

VIII. Conclusion

This study examined custom permission misconfigurations by Android app developers. We reviewed the Android documentation, OS source code, and related research to identify ten guidelines for proper custom permission usage. Analyzing 174,740 apps from Google Play and third-party stores, we found widespread violations, particularly in using defective custom groups and undefined custom permissions. Such practices can cause data leaks, app installation failures, and functional issues. Custom permission misconfigurations often result from developers using third-party app templates or following service documentation for app development, leading to misconfigurations to propagate. We also proposed several improvements for both Google and Android app developers to avoid custom permission misconfigurations.

Acknowledgements

This work is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Proposal ID: NCR25-DeSCEmT-SMU). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

References

[1] "AndPermission," Available: https://github.com/yanzhenjie/AndPermission/blob/master/permission/src/main/AndroidManifest.xml.

[2] "Androguard," Available: https://github.com/androguard.

[3] "Android for Developers," Available: https://developer.android.com/.

[4] "App manifest overview," Available: https://developer.android.com/guide/topics/manifest/manifest-intro.

[5] "App security improvement program," Available: https://developer.android.com/privacy-and-security/googleplay-asi.

[6] "Behavior changes included in the compatibility framework," Available: https://developer.android.com/about/versions/12/reference/compat-framework-changes#list.

[7] "Bug: 213323615," Available: https://android.googlesource.com/platform/frameworks/base/+/548edbb850227e076735615f83f8e23352b0b82d.

[8] "Building an intent," Available: https://developer.android.com/guide/components/intents-filters#Building.

[9] "Define a custom app permission," Available: https://developer.android.com/guide/topics/permissions/defining.

[10] "Define and enforce permissions: Naming convention," Available: https://developer.android.com/guide/topics/permissions/defining#naming.

[11] "FuzzyWuzzy," Available: https://pypi.org/project/fuzzywuzzy/.

[12] "GeTui: Document Center (in Chinese)," Available: https://docs.getui.com/getui/mobile/android/overview/.

[13] "Increased situational context," Available: https://developer.android.com/training/permissions/usage-notes#increased_situational_context.

[14] "newCloudApp," Available: https://github.com/holai/newCloudApp.

[15] "OPPO Push (in Chinese)," Available: https://open.oppomobile.com/bbs/forum.php?mod=viewthread&tid=1918l.

[16] "<permission>," Available: https://developer.android.com/guide/topics/manifest/permission-element.

[17] "<permission-group>," Available: https://developer.android.com/guide/topics/manifest/permission-group-element.

[18] "<permission-tree>," Available: https://developer.android.com/guide/topics/manifest/permission-tree-element.

[19] "preparePackageLI," Available: https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/services/core/java/com/android/server/pm/InstallPackageHelper.java.

[20] "Security checklist," Available: https://developer.android.com/privacy-and-security/security-tips.

[21] "<uses-permission-sdk-23>," Available: https://developer.android.com/guide/topics/manifest/uses-permission-sdk-23-element.

[22] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), Denver, CO, USA, October 12-16, 2015*, 2015.

[23] Y. Aafer, X. Zhang, and W. Du, "Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis," in *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016*, 2016.

[24] 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.

[25] Z. Han, L. Cheng, Y. Zhang, S. Zeng, Y. Deng, and X. Sun, "Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones," in *Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Beijing, China, September 24-26, 2014*, 2014.

[26] A. K. Jha, S. Lee, and W. J. Lee, "Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, May 20-28, 2017*, 2017.

[27] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[28] R. Li, W. Diao, Z. Li, S. Yang, S. Li, and S. Guo, "Android Custom Permissions Demystified: A Comprehensive Security Evaluation," *IEEE Transactions on Software Engineering*, vol. 48(11), pp. 4465–4484, 2022.

[29] R. Li, W. Diao, S. Yang, X. Liu, S. Guo, and K. Zhang, "Lost in Conversion: Exploit Data Structure Conversion with Attribute Loss to Break Android Systems," in *Proceedings of the 32nd USENIX Security Symposium (USENIX-SEC), Anaheim, CA, USA, August 9-11, 2023*, 2023, pp. 5503–5520.

[30] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission Issues in Open-Source Android Apps: An Exploratory Study," in *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, OH, USA, September 30 - October 1, 2019*, 2019.

[31] 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.

[32] S. Yang, Q. Hou, S. Li, and W. Diao, "Do App Developers Follow the Android Official Data Security Guidelines? An Empirical Measurement on App Data Security," in *Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC), Seoul, Republic of Korea, December 4-7, 2023*, 2023, pp. 71–80.

[33] Y. Yang, M. Elsabagh, C. Zuo, R. Johnson, A. Stavrou, and Z. Lin, "Detecting and Measuring Misconfigured Manifest in Android Apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, November 7-11, 2022*, 2022.