Born with a Silver Spoon: On the (In)Security of Native Granted App Privileges in Custom Android ROMs

Chao Wang*, Yanjie Zhao*, Jiapeng Deng, and Haoyu Wang†
Huazhong University of Science and Technology, Wuhan, China
chaowang_@hust.edu.cn, yanjie_zhao@hust.edu.cn, jiapeng@hust.edu.cn, haoyuwang@hust.edu.cn

***Abstract*—The customization and fragmentation of the Android ecosystem have fostered its prosperity and highlighted the growing importance of conducting security audits on these customized systems. This significance is driven by the distinct strategies that Original Equipment Manufacturers (OEMs) deploy to enhance device performance and user experience, which are important to their competitive differentiation. A key aspect of these strategies includes system-level optimizations for super apps and other widely used apps, marking a competitive trend among OEMs.**

Granting privileges to such apps often stems from trust in these apps. However, without proper validation of apps' identities, this can lead to severe implicit trust vulnerabilities, providing a convenient pathway for malicious apps to impersonate privileged ones and gain their access rights. For malicious developers, exploiting these vulnerabilities is both cost-effective and potentially highly rewarding. In this study, we undertook a comprehensive analysis of 686 custom Android ROMs from 46 OEMs, aimed at uncovering potential security risks associated with implicit trust vulnerabilities in apps. Our investigation identified 3,085 instances where third-party app package names were embedded within the ROMs. Alarmingly, only seven of these instances had implemented adequate authentication mechanisms to mitigate the associated risks, exposing 3,078 potential vulnerabilities that exhibited an increasing trend over time. We have reported 22 manually confirmed cases to seven relevant OEMs. As of the time of writing this paper, four vulnerabilities have been explicitly acknowledged by the OEMs, and one has been assigned a CVE ID.

1. Introduction

The Android operating system (OS), commanding over a 70% share of the global mobile market [37], has emerged as a focal point in mobile security research [34], [30]. Among the various components of Android security, the analysis of Read-Only Memory (ROM) images stands out as a critical area of study. ROMs, which house the system framework and pre-installed apps, provide a pivotal perspective for evaluating system vulnerabilities and security risks [24], [27]. OEMs usually release **custom ROMs**, tailored versions of the Android OS that are modified and optimized for particular hardware [19]. Unlike official Android versions from Google, these custom ROMs incorporate extra features, enhancements, and adjustments to boost performance, enhance user experience, and ensure device uniqueness. These customizations, while facilitating brand differentiation, necessitate extensive security and compatibility evaluations to maintain high standards of security and functionality [29].

One of the key competitive advantages among OEMs lies in the performance and personalization of customized systems. As apps grow in size and complexity, evolving into what is now known as **super apps** [16], [40], [39], ensuring a seamless user experience for these apps has become an implicit battleground for OEMs. Consequently, custom ROMs often grant special access and enhanced capabilities to **super apps** (e.g., Alipay, WeChat, and Slack) and other widely used **popular apps** (e.g., Facebook, YouTube, and WhatsApp) that exceed those available to regular apps. These privileges can include advanced system controls and access to protected hardware features. However, granting such privileges without stringent scrutiny introduces significant security risks.

Unfortunately, authenticating third-party apps in custom ROMs is not straightforward. Unlike **system apps**, which can be efficiently validated using UIDs or system signatures, the most common and convenient method for verifying **third-party apps** involves the usage of **package names**. However, this approach is far from secure as package names can be easily spoofed. While authenticating apps through third-party developers' signatures could provide an efficient method, it also introduces significant challenges. Managing the signatures of third-party developers is not only complex but also labor-intensive. As a result, many custom ROMs do not rigorously implement the **signature verification** method for authenticating third-party apps.

Due to the lack of robust authentication measures in many custom ROMs when granting privileges to certain apps, vulnerabilities are inadvertently introduced. These oversights provide opportunities for malicious entities to

*Chao Wang and Yanjie Zhao contributed equally to this work.
†Haoyu Wang is the corresponding author (haoyuwang@hust.edu.cn).
The full name of the affiliation for all authors is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.

achieve substantial gains with minimal effort and cost. For example, by impersonating privileged apps, an app could exploit these vulnerabilities to escalate its privileges. This unauthorized privilege escalation can lead to a variety of malicious activities, from data breaches to bypassing system security protocols, ultimately compromising the device's integrity and security. In this paper, we refer to such vulnerability as Implicit Trust App vulnerability (**ITA vulnerability**).

Previous research [23], [27] has explored the issues of "hanging attributes", which occur when **system apps** are deleted or renamed without timely updates to the corresponding code segments in the ROMs. With super apps prevalent, OEMs now actively adapt and optimize their customized ROMs for certain **third-party apps** to enhance the user experience [1]. However, current studies have not yet conducted a thorough investigation into vulnerabilities related to privileges granted to such **non-system components**. To some extent, vulnerabilities in non-system components can be exploited more easily than those in system apps and are more likely to be overlooked, yet their potential for causing significant risks is equally substantial.

To fill this gap, we aim to tackle the challenge of detecting **ITA** vulnerabilities in custom Android ROMs through a systematically automated methodology. To the best of our knowledge, our study is the first to identify the trend of custom ROMs granting privileges to third-party apps and to conduct a focused security analysis on the privileges granted to non-system components. A prevalent practice among Android OEMs of embedding app whitelists into ROMs to grant special privileges to certain apps enables us to use static analysis techniques to precisely identify and locate **hardcoded package names** in ROM images, as well as the sequence of operations executed. Therefore, our approach includes the following three key stages: firstly, the construction of a dataset, which involves the meticulous collection of 686 ROM images together with a list of 200 super/popular apps for subsequent analysis; secondly, the preprocessing of ROM images to prepare them for examination; and thirdly, the implementation of a detector that leverages static analysis techniques to identify **ITA** vulnerabilities.

Through this methodical process, we successfully identified a total of 3,085 instances where package names from the super/popular app list were embedded within the ROMs. Notably, only seven of these instances had implemented adequate authentication mechanisms capable of effectively mitigating the associated risks. This led us to uncover 3,078 potential **ITA** vulnerabilities. Among these, 3,025 instances contained at least one third-party app that was not pre-installed in the affected ROM, and in 2,497 instances, **there were discrepancies between the regions where the privileged apps were popular and the target regions for which the ROMs were compiled**. This greatly increased the possibility of the vulnerabilities being exploited. We have reported 22 manually verified cases to seven related OEMs. At the time of writing this paper, three OEMs have engaged positively and are in ongoing discussions. Four vulnerabilities have been formally recognized by the OEMs, and one has been assigned a CVE ID. The findings of our research shed light on the prevalence of **ITA** vulnerabilities within custom Android ROMs. To facilitate a deeper understanding of these vulnerabilities, we provide a classification scheme, followed by a detailed explanation of each **ITA** vulnerability category identified.

Contribution. Our main contributions are as follows:

- **Novel Discovery.** We unearth a previously underexplored security vulnerability within custom Android ROMs, i.e., the **ITA** vulnerability.
- **Systematic Tool.** We have presented a systematically automated framework, designed for identifying **ITA** vulnerabilities in custom Android ROMs.
- **Comprehensive Evaluation.** Our methodology has been evaluated on a dataset comprising 686 custom ROM images, leading to the identification of 3,078 instances of potential **ITA** vulnerabilities. We have reported 22 verified cases to seven OEMs. As of this paper's writing, three have responded positively and are discussing further. Four vulnerabilities have been officially acknowledged, with rewards issued, and corrective actions are underway.

2. Background And Motivation

2.1. Android Firmware

Android firmware refers to the foundational software and operating system (OS) running on a device, while read-only memory (ROM) typically denotes a specific version of this firmware. A ROM encompasses the Linux-based Android OS, hardware drivers, pre-installed apps (including system apps), and the bootloader and recovery mode. Besides official releases, ROMs may also be custom versions from third parties, offering extra features and enhanced performance. Starting with Android 10, Google implemented dynamic partitioning [5], which allows system partitions such as system, product, and vendor partitions to dynamically resize, thereby improving storage utilization, and simplifying the process of system updates. Additionally, different OEMs may employ specific encoding and encryption formats to secure firmware contents, boosting device security.

2.2. Android Access Control

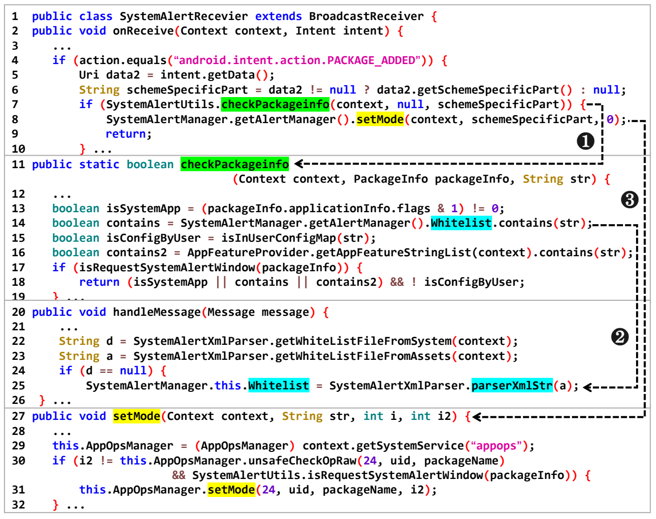
In the Android OS, the access control mechanism is structured around a framework that incorporates User Identifiers (UID), app signatures, package names, and permissions, offering a multifaceted approach to security and privacy management. Apps receive a unique **UID** upon installation for secure operations and resource management. **App signature** verifies authenticity and integrity, permitting updates or interactions solely from apps with a matching cryptographic key (verifying the app's developer). **App package name** distinctively identifies the app on the device and in the app marketplace.

Permissions play an essential role within the Android framework, categorized into the following three main classifications according to the Android Developer Documentation [20]: **(1) Install-time permissions** are automatically granted during the app installation process, streamlining the process while maintaining a level of user awareness. **(2) Runtime permissions** require explicit user consent for accessing sensitive resources while an app is running—a mechanism introduced in Android 6.0 (i.e., API level 23). Furthermore, Android augments its access control framework with the inclusion of AppOps [18], a subsystem that users can access by navigating to the special app access page in the system settings. This enhancement is supported by **(3) special permissions** [21], regulated via AppOpsManager, allowing for precise management of app operations.

2.3. Motivating Examples

We now demonstrate two authentic examples to underscore the potential security threats arising from insufficient authentication mechanisms and sole reliance on third-party package names for access control or security checks.

Example I: Exploiting Permission Control Vulnerability. As described in §2.2, the process of granting special permissions typically involves users navigating to the special app access page within the system settings. However, we discovered a critical vulnerability in a custom ROM (denoted as ROM A) from a well-known Android OEM that contravenes this security policy. Specifically, ROM A automatically sets the AppOps mode of the special permission `android.permission.SYSTEM_ALERT_WINDOW` to "allow" for certain applications upon installation. This authorization is based solely on the application's package name, as illustrated in Figure 1.



```
1  public class SystemAlertRecevier extends BroadcastReceiver {
2  public void onReceive(Context context, Intent intent) {
3     ...
4     if (action.equals("android.intent.action.PACKAGE_ADDED")) {
5         Uri data2 = intent.getData();
6         String schemeSpecificPart = data2 != null ? data2.getSchemeSpecificPart() : null;
7         if (SystemAlertUtils.checkPackageinfo(context, null, schemeSpecificPart)) {
8             SystemAlertManager.getAlertManager().setMode(context, schemeSpecificPart, 0);
9             return;
10        } ...
11 public static boolean checkPackageinfo
                                (Context context, PackageInfo packageInfo, String str) {
12    ...
13    boolean isSystemApp = (packageInfo.applicationInfo.flags & 1) != 0;
14    boolean contains = SystemAlertManager.getAlertManager().Whitelist.contains(str);
15    boolean isConfigByUser = isInUserConfigMap(str);
16    boolean contains2 = AppFeatureProvider.getAppFeatureStringList(context).contains(str);
17    if (isRequestSystemAlertWindow(packageInfo)) {
18        return (isSystemApp || contains || contains2) && ! isConfigByUser;
19    } ...
20 public void handleMessage(Message message) {
21     ...
22     String d = SystemAlertXmlParser.getWhiteListFileFromSystem(context);
23     String a = SystemAlertXmlParser.getWhiteListFileFromAssets(context);
24     if (d == null) {
25         SystemAlertManager.this.Whitelist = SystemAlertXmlParser.parserXmlStr(a);
26  } ...
27 public void setMode(Context context, String str, int i, int i2) {
28    ...
29    this.AppOpsManager = (AppOpsManager) context.getSystemService("appops");
30    if (i2 != this.AppOpsManager.unsafeCheckOpRaw(24, uid, packageName)
                        && SystemAlertUtils.isRequestSystemAlertWindow(packageInfo)) {
31        this.AppOpsManager.setMode(24, uid, packageName, i2);
32    } ...
```

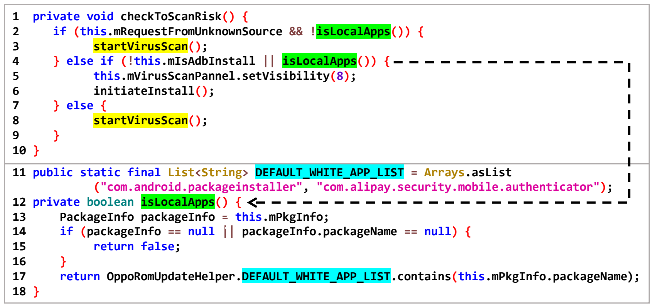
Figure 1: A vulnerability identified in the system app, `SecurityPermission`, on ROM A, involving the automatic granting of permissions.

In Android OS, the installation, update, and uninstallation of apps usually trigger specific broadcast intents. System services and system apps respond to these events by listening to these broadcasts and automatically performing a series of actions, such as security checks and permission management, in reaction to these occurrences. As shown in Figure 1, the system app (i.e., `SecurityPermission`) has the `android.permission.MANAGE_APP_OPS_MODES` permission to call the AppOpsManager and monitor the `android.intent.action.PACKAGE_ADDED` action as shown in line 4 to consider automatic authorization upon the installation of an app. Upon receiving the broadcast of app installation, it extracts the package name of the installed app from the intent, as demonstrated in lines 5 to 6. This package name is then passed to the `checkPackageInfo` function (step ❶) to determine if it is included in an authorized whitelist (at line 14). This whitelist is read from a local file (step ❷), which may be located in the system path or within the system app's resource directory from lines 22 to 23. Note that in line 17, there is a check for the floating window permission as declared in the installed app's manifest. The evaluation passes if the app meets any of the following criteria: it is a system app, it is included in the whitelist, or it is listed in the `AppFeatureStringList`. Additionally, the app must not be present in the `UserConfigMapList` to pass this verification.

It is noteworthy that the system app only verifies the installed app's package name. Therefore, by forging an app with a package name that appears on the whitelist, one can bypass the validation process. Once authenticated, the system app, as shown in line 8, obtains an instance of `alertManager` and invokes the `setMode` function (step ❸). As illustrated in line 31, this involves calling the `setMode` function of `AppOpsManager` to control the AppOps modes. The first parameter, `24`, represents the `android.permission.SYSTEM_ALERT_WINDOW` permission, and the fourth parameter, `i2`, is set to zero, indicating "allow". A final check is performed before this call: line 30 checks if the calling UID corresponds to the package name and whether the requested operation code is abnormal. Unfortunately, this fails to authenticate the true identity of the installed app, leading to the emergence of a **`ITA`** vulnerability.

Example II: Bypassing Antivirus. To ensure user safety, routine virus scans are usually performed before an app is allowed to be installed on an Android smartphone. The `PackageInstaller` app, serving as a system app that manages the user installation interface, typically conducts a series of scans before installation. Unfortunately, we have discovered a whitelist mechanism within the `PackageInstaller` app of a custom ROM developed by another renowned Android OEM, designated as ROM B. This mechanism allows certain apps to circumvent the built-in virus-scanning feature.

As illustrated in Figure 2, the `checkToScanRisk` function (lines 1 to 10) plays a critical role in determining whether to perform a virus scan prior to app installation. This function consists of three branches. The initial `if` statement, located at line 2, evaluates whether the source of installation is unknown (with any source aside from system



```java
1  private void checkToScanRisk() {
2     if (this.mRequestFromUnknownSource && !isLocalApps()) {
3           startVirusScan();
4     } else if (!this.mIsAdbInstall || isLocalApps()) {
5           this.mVirusScanPannel.setVisibility(8);
6           initiateInstall();
7     } else {
8           startVirusScan();
9     }
10 }
11 public static final List<String> DEFAULT_WHITE_APP_LIST = Arrays.asList
           ("com.android.packageinstaller", "com.alipay.security.mobile.authenticator");
12 private boolean isLocalApps() {
13     PackageInfo packageInfo = this.mPkgInfo;
14     if (packageInfo == null || packageInfo.packageName == null) {
15           return false;
16     }
17     return OppoRomUpdateHelper.DEFAULT_WHITE_APP_LIST.contains(this.mPkgInfo.packageName);
18 }
```

Figure 2: A vulnerability identified in ROM B's system app, `PackageInstaller`, enabling the bypass of its virus scanning functionality.

app installations considered unknown) and confirms that the app being installed does not qualify as a `LocalApp`. The implementation for determining a `LocalApp` is shown in lines 12 to 18. If the package name of the installing app is in the `DEFAULT_WHITE_APP_LIST`, initialized in line 11, it simply returns `True` without any additional verification. Notably, besides `com.android.packageinstaller` is a system app, `com.alipay.security.mobile.authenticator` is a third-party app that included in this whitelist. Consequently, this inclusion allows bypassing the conditional statement in line 2 by masquerading the package name as `com.alipay.security.mobile.authenticator`, thereby proceeding to the conditional check in line 4. During the evaluation of the conditional statement in line 4, the app with the disguised package name will successfully pass the `isLocalApps` function's verification, circumventing the virus scan and moving directly to the installation interface.

The exploitable vulnerabilities demonstrated in Example I and Example II are both identified as **ITA** vulnerabilities. For these two motivating examples, we provide the exploitation process (in Appendix §A) to substantiate the exploitability of such **ITA** vulnerabilities. The existence of **ITA** vulnerabilities, coupled with their exploitability and associated potential risks, serve as the motivation for this paper.

2.4. Threat Model

We consider attackers aim to gain privileges designed for special apps in ROMs but lack proper authentication by spoofing package names of popular/super apps. Our assumptions are as follows:

First, at least one app from the list of third-party privileged apps involved in an identified **ITA** vulnerability does not exist on the target device. Due to the ease of installing and removing third-party apps, coupled with the fact that 3,025 out of 3,078 detected **ITA** vulnerabilities involve at least one third-party app that was not pre-installed in the affected ROM, and 2,497 instances of regional mismatches between privileged apps and target ROMs (see §4.3), achieving this is not difficult.

Secondly, attackers can pre-analyze specific Android ROMs in advance to understand how to exploit identified vulnerabilities. They can observe different ways of abusing privileged capabilities, e.g., some privileged apps being automatically whitelisted after installation, allowing them to run continuously in the background without being killed by system optimizations, or when privileged apps requesting location information could obtain more accurate and real-time data. Attackers can selectively exploit privileges to facilitate malicious activities.

3. Approach

In this section, we will present our approach that aims to automate the identification of **ITA** vulnerabilities from Android custom ROMs. The overall workflow is described in Figure 3. §3.1 introduces the construction of our necessary datasets. §3.2 details the pre-processing process of the collected ROM images. §3.3 is dedicated to the implementation of our static analysis-based detection method. Finally, §3.4 addresses the filtering and categorization of the results.

3.1. Dataset Collection

Rom Dataset. In light of the diminishing availability of official ROM images and the suboptimal timeliness of ROM images provided by sites like `firmwarefile.com` [9], we primarily utilized the Android Dumps repository [2] as our main source for downloading ROM images. Android Dumps is a well-maintained open-source repository of phone images, encompassing approximately 250 mobile phone brands, offering rich choices and better timeliness. Despite the extensive resources provided by Android Dumps, there were still gaps in their collection, particularly with certain brands and models. To enhance the comprehensiveness of our dataset, we supplemented missing ROMs from brands such as Huawei and Sony, primarily sourcing these images from `firmwarefile.com` and `xpericheck.com` [15].

Super/Popular App List. As introduced in §1, **ITA** vulnerabilities often arise when custom ROMs aim to afford special privileges to super apps and other popular apps. Our subsequent analysis will thus concentrate on these specific third-party apps. We began by harvesting the top 100 apps from Google Play's U.S. region, utilizing web crawling techniques on the renowned advertising network website, AppBrain [4]. Acknowledging China's significant mobile market share, the lack of Google Play access for Chinese users, and the popularity of OEMs in the Chinese market, along with its unique market characteristics, we similarly extracted the top 100 apps from Tencent App Store [13], a leading app marketplace in China. We carefully confirmed that well-known super apps were included within these collections. These two datasets were merged to create our super/popular app list.

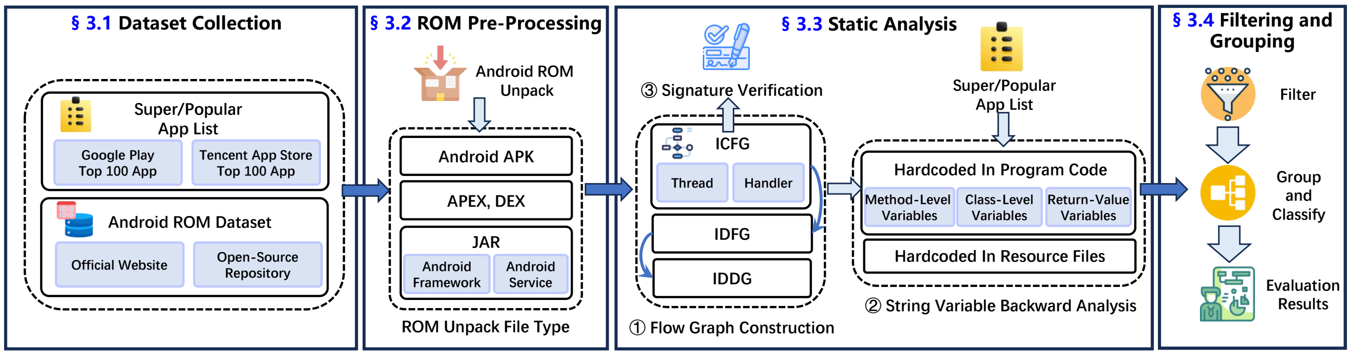


Figure 3: The overview of our methodology.

3.2. ROM Pre-Processing

Android ROMs exhibit multiple variations due to version updates and third-party customizations. As mentioned in §2.1, apart from different OEMs adopting various encoding and encryption formats, Google has also implemented dynamic partition for Android 10 and higher versions. This integration consolidates `system.img`, `vendor.img`, and `product.img` into a larger partition, known as `super.img`. Therefore, the process of unpacking ROM images is not standardized and requires special handling for different formats.

Upon decompressing the ROM, in most cases, we can directly access the image files. However, some OEMs use custom image formats, necessitating specific tools for further unpacking. For example, Oppo's `.ofs` and OnePlus's `.ops` formats require the use of `oppoDecrypt` [12] for unpacking, while Sony's `.sin` format necessitates `flashtool` [10]. For Android 10 and later versions, the next step involves converting the `super.img` file into an `.ext4` format using `Simg2img` [14]. Subsequently, `imjtool` [11] is employed to extract the individual partition image files contained within the `super.img` for further processing. In contrast, for versions prior to Android 10, we directly obtain image files such as `system.img` and `vendor.img`. The last step involves extracting content according to the format of the image files. The Enhanced Read-Only File System (EROFS) [6], a read-only file system, has gained popularity as a high-performance compressed file system in recent years. It excels in both random and sequential access times compared to other file systems. Extracting content from this compressed file type requires `erofs-utils` [7]. For standard `.ext4` format image files, `extfstools` [8] is used for content extraction.

After unpacking the images, it is essential to analyze all the encompassed framework files and pre-installed apps (including system apps) such as those in `.jar`, `.apk`, `.dex`, and `.apex` formats. The Android Pony EXpress (APEX) [3] container format, introduced with Android 10, plays a significant role in the installation process of lower-level system modules, such as the `PermissionController` module. Upon examining the APEX file, we discovered that the `apex_payload.img` enclosed is an `ext2` format system image. We employ `extfstools` [8] for unpacking the `ext2` file to extract the `.apk` files. Additionally, we preserve various metadata property files contained within the ROM, such as `build.prop`, which are crucial for understanding a ROM's configuration and specifications.

3.3. Static Analysis

Our preliminary analysis revealed that ROMs frequently hardcode the package names of super apps and other popular apps when conferring special privileges. Thus, our static analysis aims to efficiently pinpoint instances where these hardcoded package names are used to grant privileges without necessary authentication. Starting with the creation of a comprehensive flow graph guarantees the thoroughness and precision of our analysis. Subsequently, we perform a backward analysis of string-type variables to accurately determine their actual values. Finally, our examination includes reviewing the context of privilege-granting functions to identify the implementation of robust authentication (e.g., signature verification). Our static analysis is conducted utilizing Soot [38] and FlowDroid [25] as the foundational frameworks.

Flow Graph Construction. In constructing interprocedural control-flow graphs (ICFGs), FlowDroid and Soot are essential but may not fully capture the complexities of multi-threaded operations, notably those involving `Thread` and `Handler`. To mitigate this, we improve the completeness of the ICFGs by explicitly linking methods like `thread.run()` and `handleMessage(android.os.Message)` with their actual callers, thereby enhancing the graph's resilience and granularity. Leveraging the refined ICFGs, we develop the interprocedural data-flow graph (IDFG) by meticulously tracing data movements across the ICFG via Definition-Use (Def-Use) chains. The IDFG excels in mapping how data values and their effects are traced back from specific program points, such as the usage of a variable, to their origins. Nonetheless, it is insufficient; a thorough grasp of variable behaviors across different scopes and contexts is essential. We thus analyze data entity dependencies within programs using the interprocedural data-dependency graph (IDDG), constructed based on the ICFG and IDFG. Maintaining the IDDG is a non-trivial task that requires an in-depth understanding of variable interactions, including the initialization and updating mechanisms of various data types such as `String[]`, `ArrayList`, `HashSet`, and `HashMap`, etc. For instance, the IDDG precisely tracks how these variables are populated and modified through methods like `.add()` and `.put()`, providing a comprehensive overview of their lifecycle.

String Variable Backward Analysis. When a custom ROM intends to grant privileges to third-party apps, it often requires evaluating package names. Thus, variables used in conditional statements are critically important. In some exceptional scenarios, the system may directly allocate privileges to certain apps without performing any conditional evaluations. To ensure a comprehensive analysis, we first evaluate the conditions specified within `if` statements and then examine the remaining control flow for instances where hardcoded string variables are directly utilized as parameters. By conducting a backward data flow analysis from these points, we can identify the sources of the variables and obtain their values. We consider instances associated with package names of the collected super/popular apps in §3.1. This approach is particularly effective for variables whose values are not directly assigned or are generated dynamically. Depending on the source of the string variables, we categorize them into two types, i.e., those originating from the **program code** and those loaded from **resource files**. We now discuss each separately:

(i) Hardcoded in program code. Hardcoded string variables in program code are explicitly defined, including string literals, numerical constants, and arrays. We categorized these based on their origin into three scenarios. (1) **Method-level variables**, local to the method, include string constants (e.g., if `Packagename.equals("com.tencent.mm")`) and string arrays. We deduce these variables' actual values through IDDG analysis. (2) **Class-level variables** include global or static fields within the class or static fields from external classes, covering string constants or arrays. Their values, often set in `<init>` or `<clinit>`, require multi-function analysis for accurate reconstruction. (3) For **return-value variables** (e.g., the return value of `isLocalApps()` in Figure 2), we trace their origins through backward data flow analysis to identify their source functions and uncover their actual values.

(ii) Hardcoded in resource files. Our method extends beyond analyzing variables directly embedded in the code. We also examine local resource files, where whitelist entries are often stored as key-value pairs in formats such as XML and JSON. These files may be located in the APK's assets directory or within the ROM's configuration files. To comprehensively capture both package names and their associated tags/attributes, we identify file access patterns typically used for reading these resources. We look for specific constructor invocations (e.g., `<java.io.FileInputStream: void <init>(java.lang.String)>`) and usage of Android's `android.content.res.AssetManager`. By recognizing these patterns, we can determine which files are being accessed and subsequently analyze their contents for relevant whitelist entries.

Signature Verification. Previous research [23], [27] identifies three key methods for app authentication that coexist with package names in ROM: UID verification, app signature verification, and configuration verification. However, UIDs lack distinctiveness for third-party apps, only differentiating them from system apps. Given that third-party apps can be installed on any device, configuration verification isn't exclusive to authentication. Thus, signature verification is usually the most effective method for third-party app identity verification.

We identify package name verification nodes in ICFGs, then conduct bidirectional data flow analysis: tracing the package name propagation both forward and backward to check if the package name is passed as an argument to functions that retrieve signature information for comparison. Specifically, we focus on calls to `getPackageInfo()` with the `GET_SIGNATURES` flag, and accesses to signature-related fields such as `packageInfo.signingInfo` and `packageInfo.signatures`. Additionally, we scrutinize the usage of `getSigningCertificateHistory()`. Instances lacking these signature authentication mechanisms are flagged as potential vulnerabilities.

3.4. Filtering and Grouping

To facilitate a more comprehensive analysis of the identified vulnerabilities, we implemented a preliminary classification step using a Large Language Model (LLM). For this task, we locally deployed a Mistral-7B-Instruct-v0.2 model. Our methodology began with randomly selecting a sample of 100 **ITA** vulnerabilities. These samples were then manually classified into six categories based on their code functionality, as detailed in Table 1. This manually classified set served as our ground truth, also providing a robust foundation for evaluating the performance of our LLM-based classification approach.

TABLE 1: Categorization of the ITA vulnerabilities.

Category	Description
Resource Priority Management	High priority in resource handling and processing.
System Permission Management	Default granting or Suggestion of system permissions.
Antivirus Scan Exemption	Exempts trusted apps from antivirus scans.
App Persistence Mechanisms	Whitelist mechanisms for app survival.
Precise Information Access Control	Grants access to accurate and detailed data.
Specialized Capability Provisioning	Provides apps with unique capabilities.

Using insights gained from this manual classification process, we designed prompts for each category, incorporating key functional characteristics. We employed a few-shot approach for the LLM classification. This involved providing the LLM with descriptions of each category and, for each vulnerability, inputting the source code of the vulnerable function. The LLM was tasked with classifying the function into one of the six predefined categories or as

`Unknown`. Additionally, the LLM was required to output a confidence score for each classification decision.

4. Evaluation

This section presents our comprehensive evaluation results. We begin in §4.1 with a detailed overview of the datasets utilized and the specifics of our running environment. §4.2 offers an in-depth analysis of our tool's performance, encompassing both its time efficiency and effectiveness. Finally, §4.3 offers an analysis of the overall distribution of classification results, providing insights into the patterns and trends observed in our findings.

4.1. Evaluation Setup

Rom Dataset. Ultimately, we collected 686 custom ROMs from 46 different OEMs for analysis. These ROMs span Android API levels from 15 to 34 and involve over 323 different smartphone models. By extracting the `ro.product.locale` property from each ROM's compiled `build.prop` file, we were able to determine the target region for which the ROM was compiled. Our dataset encompassed ROMs originating from more than 13 regions worldwide.

Research Scope. Our research scope encompasses vulnerabilities within system frameworks and system apps. For system apps, we place a specific focus on analyzing APKs related to critical system functions such as security features, permission management, and antivirus capabilities (e.g., `PermissionController`, `PackageInstaller`).

Running Environment. Our experiments are conducted on a server running Ubuntu 22.04 with two 64-core AMD EPYC 7713, 512 GB RAM, and two NVIDIA 3090 16GB GPUs.

4.2. Performance Evaluation

Runtime Performance. Our analysis of custom ROMs exhibited varied execution times, reflecting the diversity in ROM complexity and size. On average, the processing of a single ROM required approximately 3,420 seconds. The distribution of analysis durations across our dataset showed: 38.8% were processed in under 1,000 seconds, while the majority (54.4%) required between 1,000 and 10,000 seconds. Only a small fraction (6.8%) exceeded 10,000 seconds in processing time. Notably, 93.2% of all analyzed ROMs were processed within 2.8 hours, indicating that our methodology achieved a balance between thoroughness and computational efficiency for the vast majority of cases.

In our analysis of 686 ROMs, timeouts affected a total of 98 files across all ROMs. Given that each ROM contained an average of 102.46 files and these timeouts did not occur in critical framework or service files, their effect on our main analysis results was negligible.

Effectiveness. We identified a total of 3,085 unique instances where package names from the super/popular app list are hardcoded into the ROMs. Notably, only **seven** instances implemented sufficient authentication measures, revealing **3,078** potential **`ITA`** vulnerabilities.

To validate the effectiveness of our tool, we conducted a 5% sampling of the 3,078 identified **`ITA`** vulnerabilities. It is crucial to note that we consider any use of package names without additional verification as a True Positive (TP). We found no False Positives (FP) in the sampled set. Our rigorous verification process yielded the following results: Our analysis revealed three primary categories of package name usage: direct comparison with constant strings (55 cases), comparison with string constants or lists in fields (47 cases), and invocation of auxiliary verification functions (52 cases). Within the last category, we observed further subcategories: 10 cases returned whitelist lists for additional package name comparison, 19 involved direct comparison with constant strings within the called function, and 23 compared with strings or lists in fields within the called function.

Notably, across all 154 verified cases, the call stack depth never exceeded three levels. This observation indicates that our backward analysis of variables did not encounter complex situations that might lead to false positives. Furthermore, our findings align with the mitigation strategies employed by vendors, where adding signature verification is a widely recognized approach. Typically, this verification is performed either near the comparison point or before/after the call chain. We did not encounter any exceptional cases that deviated from this pattern. Accurately assessing false negatives in our study is challenging due to the Android framework's vast codebase (100,000+ conditional branches). This complexity, a common obstacle in static analysis of large-scale systems, limits our ability to guarantee complete coverage [41], [22], [36].

Accuracy of LLM Classification. We evaluated our LLM-based classification using 100 manually classified vulnerabilities as ground truth, the same set used for prompt design. The LLM achieved an accuracy of 89%.

4.3. Evaluation Results

Distribution of OEMs. As illustrated in Figure 4, the distribution of OEMs associated with **`ITA`** vulnerabilities reveals a complex landscape of device optimization requirements. OEMs such as iQOO, Xiaomi, OnePlus, and Oppo are frequently associated with such issues, indicating a significant need for app optimization specifically tailored to enhance device performance. Notably, brands like iQOO, Oppo, and Realme have an average of over 100 hardcoded third-party app package names per ROM. In contrast, when we applied our detection framework to devices from Google, **no instances** of hardcoded third-party app package names were found, highlighting a stark difference in the approach to device optimization between these OEMs.

Distribution of Locations and Android SDK levels. Figure 5 illustrates that **`ITA`** vulnerabilities are not confined to any single region but rather present a global issue. The distribution of these vulnerabilities is particularly pronounced

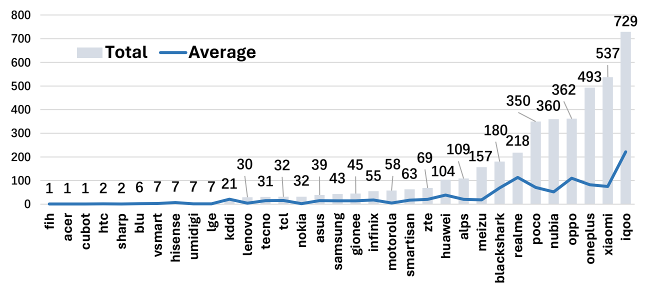


Figure 4: Distribution of the OEMs carrying `ITA` vulnerabilities, combining a bar chart that shows the total number of vulnerabilities identified per OEM and an overlaid line chart depicting the average number of `ITA` vulnerabilities per ROM for each OEM.

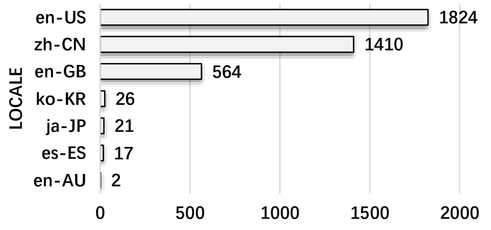


Figure 5: Distribution of the locations associated with `ITA` vulnerabilities.

in the United States (en-US) and the United Kingdom (en-GB), as well as in China (zh-CN). This pattern may be correlated with the user demands or optimization strategies of OEMs from these regions.

To investigate the distribution pattern of vulnerabilities across different Android SDK levels, we conducted a longitudinal analysis of ROMs compiled by four top OEMs listed in Figure 4 for specific regions. These OEMs were selected based on their significant market share and influence in the Android ecosystem. Figure 7 illustrates that the number of vulnerabilities exhibits an increasing trend with the upgrade of SDK levels among these sampled OEMs. Despite some deviations from the general pattern, possibly due to different optimization considerations, this trend is notable. Given the widespread use of devices from these major OEMs, our findings suggest that a substantial portion of Android users may be increasingly exposed to potential vulnerabilities related to privileged access granted to certain apps.

Distribution of Super/Popular Apps. As shown in Figure 6, the distribution of the `ITA` vulnerabilities in apps prominently showcases the dominance of social media and instant messaging platforms. To guarantee that super/popular apps operate seamlessly and with high efficiency, device optimizations are crucial at the operating system level. This encompasses the allocation of resources with higher priority, the improvement of background processing capabilities, and the automatic conferment of certain privileges to these extensively utilized apps. We incorporate a detailed `ITA` vulnerability frequency distribution of the super/popular apps in Appendix §B.

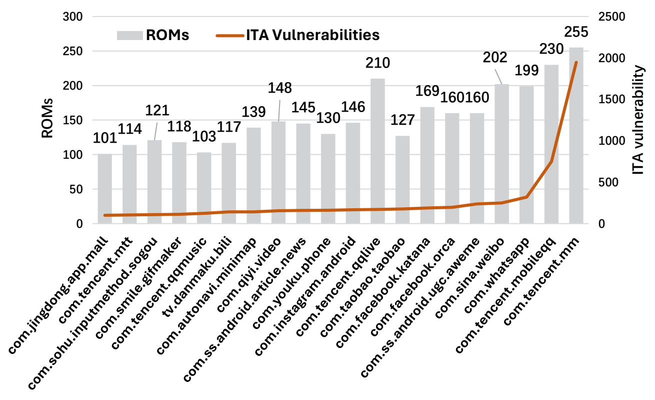
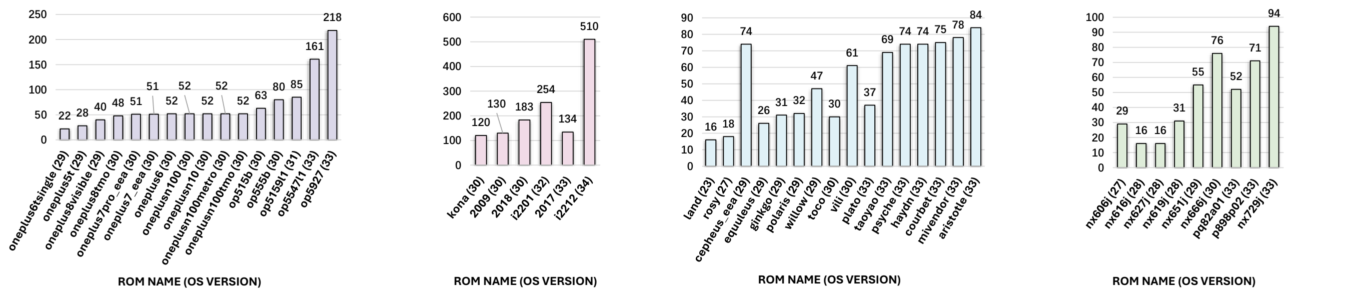


Figure 6: Distribution of the top 20 apps associated with `ITA` vulnerabilities. The bar chart depicts the number of ROMs in which each package name appears, and the overlaid line chart tracks the count of `ITA` vulnerabilities for each corresponding app.

Distribution of Super/Popular Apps Presence and Regional Alignment. Among the 3,078 potential `ITA` vulnerabilities, 3,025 are associated with at least one third-party app not pre-installed on the affected ROMs. Crucially, unlike system apps, these third-party apps can be easily uninstalled by users at any time. Additionally, in 2,497 cases, there were inconsistencies between the regions where the privileged apps were prevalent and the target regions for which the custom ROMs were compiled. For example, `ITA` vulnerabilities related to `com.tencent.mm` were found embedded in ROMs compiled for the US market, despite the app's primary user base being in China. The **presence of easily removable third-party apps not on the pre-installed list**, along with **regional inconsistencies in app popularity across different areas** [33], [32], makes it easier for attackers to find devices without the relevant privileged popular/super apps installed. This allows them to cunningly exploit regional differences to carry out attacks.

LLM Classification Results. To ensure the reliability of our analysis, we filter the classified results from the LLM based on their confidence scores. Specifically, we consider only those instances with a confidence score above or equal to 80 (out of 100) to be correctly classified. Cases with a confidence score below 80, as well as those that cannot be categorized into any of the six defined categories, are uniformly classified under the `Unknown` category. The classification results produced by the LLM indicate that `Resource Priority Management` is the most frequently identified category, with 1,622 instances. This is closely followed by `Specialized Capability Provisioning` with 770 instances. Other categories identified include `Precise Information Access Control` with 260 instances, `App Persistence Mechanisms` with 127 instances, `System Permission Management` with 77 instances, and `Antivirus Scan Exemption` with 14 instances.



(a) ROMs compiled by OnePlus for US. **(b) ROMs compiled by iQOO for US.** **(c) ROMs compiled by Xiaomi for GB.** **(d) ROMs compiled by Nubia for CN.**

Figure 7: Trends of four top OEMs carrying `ITA` vulnerabilities.

Among the classifications, 5 instances were explicitly labeled as `Unknown` by the LLM, and another 203 instances exhibited a confidence score below 80. In total, this accounts for 208 instances categorized under the `Unknown` category. Factors leading to this will be discussed in §6.1.

5. Case Study

To provide a more nuanced understanding of these vulnerabilities and their potential impact, we have selected representative cases from each category for in-depth examination. These cases illustrate the varied nature of the vulnerabilities and highlight the specific security risks they pose to the Android ecosystem. In the following subsections (§5.1 - §5.6), we present analyses of these exemplary cases. The related sources and privileged apps are shown in Table 2.

5.1. Resource Priority Management

We identified 1,622 potential instances that are related to the `Resource Priority Management` category, a phenomenon where specific packages receive preferential treatment within the system due to their elevated priority based on unique package names. This system-level prioritization becomes evident when, faced with multiple apps capable of fulfilling a particular intent, the system strategically chooses those apps with higher priority, potentially setting them as the default choice. To illustrate this concept, we present **three** examples (Cases 1 to 3 in Table 2) that highlight how this prioritization mechanism operates, showcasing its impact on app selection and default app settings within the Android ecosystem.

(i) Priority to Handle Intents. `ResolveInfo` is an essential data structure in Android that offers comprehensive details about components such as activities, services, or broadcast receivers capable of responding to intents. This structure is pivotal for identifying which components can handle specific intents within the Android ecosystem.

In **Case 1**, the `initSystemDefaultLauncher()` initializes the system's default apps, and the `PackageInstaller` assigns default handlers for different intent (e.g., video or music) based on a predefined whitelist. This process automatically sets the third-party app `com.tencent.qqlive` as the default video player, prioritizing it over the system player `com.zte.videoplayer`.

Calling the `queryIntentActivities(Intent)` method the system will return a list of `ResolveInfo` objects, each containing details about a component that can handle the intent. The `compare()` function in **Case 2** boosts the priority of `ResolveInfo` objects whose names match those in the `mSpecialApps` whitelist during comparison, effectively granting certain apps a significant advantage in handling specific intents or actions.

(ii) Priority to Show Notification. In **Case 3**, the notificationManager employs a specialized method `enqueueClassified()` to elevate the priority of certain notifications, and aims to enhance user responsiveness. This method specifically prioritizes notifications from WeChat (`com.tencent.mm`), especially those related to "red packet" (i.e., lucky money), due to their engaging and time-sensitive nature. The prioritization extends beyond social apps to include persistent and system app notifications, ensuring crucial information is prominently displayed.

Observation #1: These cases try to enhance the user experience by setting default apps or prioritizing key notifications, thus ensuring crucial information and time-sensitive interactions are foregrounded. However, malicious apps can acquire high-priority attributes, seizing resources and prioritization that should belong to legitimate apps.

5.2. System Permission Management

Automatic granting of runtime or special permissions is a practice often reserved for special system apps. This bypassing of Android's permission management mechanism indeed indicates a high level of trust in these apps, a trust that is based on the assumption that these apps are highly secure and critical to the device's operation. However, when this special treatment is extended to third-party apps, the

TABLE 2: Representative cases of `ITA` vulnerabilities in custom Android ROMs.

Category	No.	Function Signature	Privileged Apps	Brand	Region	Version	Source
Resource Priority Management	**1**	`<com.android.packageinstaller.permission.utils.ZtePermHelper:initSystemDefaultLauncher()>`	com.tencent.qqlive	OEM-A	CN	29	ZPermissionController.apk
	2	`<com.amigo.internal.app.ResolverComparator:compare()>`	com.tencent.mtt, com.tencent.mm, com.sina.web, com.tencent.mobileqq	OEM-B	CN	25	amigoframework.jar
	3	`<com.android.server.notification.NotificationClassifyManager:enqueueClassified()>`	com.tencent.mm, com.tencent.mobileqq	OEM-C	US	30, 32, 33, 34	(OEM-C)-services.jar
System Permission Management	**4**	`<com.vivo.clipboard.VivoFileHelper:takePkgPersistableUriPermission()>`	com.tencent.mm, com.tencent.mobileqq	OEM-C	US	34	(OEM-C)-framework.jar
	5	`<com.lbe.security.service.provider.PermissionManagerProvider:adaptGetInstalledPackagePermission()>`	com.tencent.mm, com.tencent.mobileqq, com.greenpoint.android.mc10086.activity	OEM-D	GB, CN	30, 31, 33	AuthManager.apk
Antivirus Scan Exemption	**6**	`<com.(OEM-E).antimal.BatchInstallationAnalyze:checkWhiteAppList()>`	`com.eg.android.AlipayGphone`, com.tencent.mm, com.tencent.mobileqq	OEM-E	US	29, 30, 31	AntimalPlugin.apk
App Persistence Mechanisms	**7**	`<com.android.server.am.OomAdjusterImpl:computeOomAdjLocked()>`	com.tencent.mm	OEM-B	CN	25	services.jar
	8	`<com.transsion.whatsappmode.WhatsappModeLice:addPowerSaveTempWhitelistAppDirect()>`	com.whatsapp	OEM-F	US	29	os-services.jar
	9	`<com.android.server.DeviceIdleController$1:onReceive()>`	com.tencent.wework, com.tencent.mm, com.tencent.mobileqq, com.alibaba.android.rimet	OEM-G	US	30	service-jobscheduler.jar
	10	`<com.android.server.notification.NotificationManagerService:enqueueNotificationInternal()>`	com.sankuai.meituan, com.tencent.mtt, com.tencent.mm	OEM-H	US	30	services.jar
	11	`<com.mediatek.server.am.AmsExtImpl:forceStopRelatedApps()>`	com.whatsapp, com.tencent.mobileqq	OEM-H	US	29	mediatek-services.jar
Precise Information Access Control	**12**	`<com.android.server.wifi.WifiServiceImpl:getConnectionInfo()>`	com.tencent.mobileqq	OEM-B	CN	25	wifi-service.jar
	13	`<com.android.server.location.gnss.(OEM-C)FakeGpsProvider$1:run()>`	com.autonavi.minimap	OEM-C	US	34	(OEM-C)-services.jar
	14	`<com.android.server.LocationManagerService:requestLocationUpdates()>`	com.autonavi.minimap, com.waze, com.tencent.map	OEM-I	AU	23	services.jar
	15	`<com.android.server.location.vcode.TaskGpsSession:onReportLocation()>`	com.tencent.map, com.autonavi.minimap	OEM-C	US	34	(OEM-C)-services.jar
	16	`<com.android.server.location.gnss.(OEM-C)GpsStateMachine:initParameter()>`	com.autonavi.minimap	OEM-C	US	34	(OEM-C)-services.jar
Specialized Capability Provisioning	**17**	`<com.android.server.(OEM-C)ConnectivityServiceImpl:updateNetcoexistWhiteList()>`	com.tencent.mobileqq	OEM-C	US	30	(OEM-C)-services.jar
	18	`<cn.(OEM-J).server.appmgmt.Nubia5GController:enable5GIfNeeded()>`	com.tencent.android.qqdownloader	OEM-J	GB	30	services.jar
	19	`<android.content.ClipboardManager:sendCopyNotify()>`	com.taobao.taobao	OEM-G	US	28	framework.jar
	20	`<com.android.server.am.firewall.(OEM-C)BackgroundActivityController:getWhiteListFromPermissionManager()>`	com.tencent.mm, com.tencent.mobileqq	OEM-C	US	34	(OEM-C)-services.jar

risks increase significantly. We identified 77 potential instances associated with permission-managing practices and have chosen **two** specific examples (Cases 4 to 5 in Table 2) to explore in detail, illustrating the complexities and potential security implications of this automated approach to permission management.

(i) Persistable Uri Permission. In **Case 4**, the `takePkgPersistableUriPermission()` granting persistable `URI` permissions to a selected list of apps, including third-party ones only based on their package name. This function enables these apps to maintain `read` and `write` access to specific content `URI`, such as files and photos, across app restarts and device reboots, without requiring user interaction or explicit permission approval.

(ii) GetInstalledPackage Permission. Case 5 involves the dynamic adjustment of the `GET_INSTALLED_APPS` permission which allows the app to access the list of installed apps. `adaptGetInstalledPackagePermission()` function updates the app's permission configuration, which includes actions such as accepting, prompting, foregrounding, or rejecting the `GET_INSTALLED_APPS` permission. Our investigation revealed a whitelist named `PermissionManager.PERM_ID_GET_INSTALLED_APPS`, which ensures that pre-approved apps receive favorable permission actions.

Observation #2: While these permissions management mechanisms may enhance the user experience by simplifying app configuration steps, they also introduce security vulnerabilities. Malicious apps could impersonate trusted package names, potentially allowing these untrusted apps to access protected data or functionalities.

5.3. Antivirus Scan Exemption

Whitelisting certain trusted apps to bypass antivirus scans is a strategy employed to optimize the performance of antivirus software. Bur if a whitelisted app is compromised or harbors malicious functionality, it can carry out harmful operations undetected, posing threats to the system's integrity and the security of user data. We will detail **one** example (Case 6 in Table 2) out of 14 potential instances where bypassing antivirus detection poses significant security challenges, highlighting the need for a multifaceted security approach.

Bypass Antimalware Scan. In **Case 6**, the anti-malware manager employs the `BatchInstallationAnalyze` class to analyze apps during batch installation. The class utilizes the `checkWhiteAppList()` function to perform a comparative analysis between the installing apps and a predefined whitelist, which is maintained by the `antiMalConfigManager`. During this process, each app's package name is checked against the entries in the whitelist. If a match is found, the app is deemed safe and is subsequently excluded from the list of potential malware.

Observation #3: This approach allows for quick exemption of recognized apps from further analysis, enhancing system performance. However, this strategy also poses significant security risks, as it depends heavily on the accuracy and integrity of the whitelist. This could potentially become a critical component in complex attacks, enabling the installation of arbitrary malicious apps on user devices.

5.4. App Persistence Mechanisms

App developers often want to keep their apps running in the background at all times. The most common method is to acquire a wakelock, which prevents the device from entering sleep mode and ensures the app remains running continuously. The system may also maintain a whitelist to prevent apps from being terminated by system optimisation. These methods are crucial for apps that require ongoing functionality or real-time updates, such as instant messaging, location services, or music streaming apps. However, abuse can lead to severe battery drain, reduced device performance, and security vulnerabilities, especially if malicious apps exploit these loopholes undetected. In this category, we have identified 127 potential instances of **`ITA`** vulnerabilities, and select **five** examples (Cases 7 to 11 in Table 2) for detailed analysis.

(i) Special Cleanup Exemption Whitelist. Managing apps' background activity is crucial for optimizing battery life and system performance. The OS often initiates task cleanups to terminate background processes as part of power-saving measures or to free up memory. However, certain apps require continuous operation in the background to function properly, such as those providing real-time services.

To address this need, `computeOomAdjLocked()` function in **Case 7** and `addPowerSaveWhitelistAppInternal()` function in **Case 8** play key roles in allowing specific apps to bypass standard system cleanups. A carefully curated list of apps deemed essential is exempt from routine background process termination. This approach ensures that vital apps can continue their operations uninterrupted, even during aggressive system power-saving actions or memory management procedures.

Android's Doze mode serves as an additional power-saving feature that selectively suspends non-essential app activities to conserve battery life when the device is idle. In **Case 9**, we examine the `DozeWhitelistUserApps` class, which encompasses a carefully selected list of apps permitted to maintain their background activities even as the device enters Doze mode.

(ii) Wake Lock Privilege Allocation. The use of wakelock in Android systems is a strategic choice to ensure that essential apps, remain active and functional even when running in the background. By preventing the device from entering a deep sleep state, wake locks allow these apps to continue their network communications, ensuring that messages and notifications are delivered promptly. In **Case 10** `enqueueNotificationInternal()` reveal a deliberate implementation of wake locks for apps identified on a white list, with special attention given to `com.tencent.mm` and `com.tencent.mobileqq`.

(iii) MediaTek-Specific App Whitelisting. MediaTek is a leading semiconductor company that produces chipsets for smartphones and incorporates the `mediatek-service.jar` or `mediatek-framework.jar` into the OEMs' ROM. These jars provide essential system-level services and functionalities for MediaTek-powered devices, ensuring their proper operation and compatibility with the MediaTek chipset. The `forceStopRelatedApps()` function, highlighted in **Case 11**, strategically terminates apps that heavily consume system resources, unless they are listed in a predefined whitelist, safeguarding essential app functionality. Beyond the aforementioned example, another interesting discovery is the existence of a whitelist that extends across 19 OEMs, found in `mediatek-framework.jar`, for `com.tencent.qqlive` to optimize in Screen Resolution Tuner.

Observation #4: While these practices aim to maintain seamless operation and connectivity, they also introduce potential risks, such as battery drain and security vulnerabilities, by allowing apps, potentially including malicious ones, to bypass standard system restrictions.

5.5. Precise Information Access Control

For certain special apps, the system may grant specialized access that encompasses critical data such as accurate geolocation tracking even under restrictive conditions like fake or power-saving modes, ensuring these privileged apps can perform their functionalities with heightened accuracy and reliability. Such precise information access is especially pivotal for apps that depend on real-time data for navigation, offering them an uninterrupted and enhanced operational capability that is not universally available to all apps. With the identified 260 potential instances, we select **five** examples (Cases 12 to 16 in Table 2) for detailed analysis.

Location. The utilization of whitelists to grant precise location information to certain apps, while bypassing energy-saving, fake location modes, and enhancing location accuracy, is grounded in the need for these apps to maintain high functionality. This is particularly true if the access control mechanisms are not robust enough to prevent misuse by apps with spoofed identities or by those that have been maliciously altered. Next, we introduce the three typical privileges.

- **Bypassing Fake Location Modes:** Android systems could use the provision of fake location information as a mechanism to protect user privacy. However, for apps that depend on precise navigation and location-based services, access to genuine, accurate location data becomes crucial. ROMs from OEM-B (**Case 12**) enable mapping apps to circumvent fake location modes, thereby granting them access to authentic location information. In the example of OEM-C (**Case 13**), according to the predefined whitelist, map apps are provided with the capabilities to set, start, read, and report fake GPS locations.
- **Ignoring Energy-Saving Modes:** The function `requestLocationUpdates()` (**Case 14**) is designed to manage location requests while considering the device's power-saving mode. In energy-saving scenarios, the system typically limits the frequency and accuracy of location updates to conserve battery life. Nevertheless, mapping apps require an exemption from these restrictions to ensure they can continue tracking locations with high accuracy.
- **Accessing Enhanced Location Accuracy:** To ensure the effective operation of navigation and mapping apps, Android systems provide special access to precise location data for map apps. When the `onReportLocation()` (**Case 15**) method is invoked, it specifically caters to mapping apps listed in a predefined selection. For these apps, the method undertakes reverse geocoding to enhance the basic location data with comprehensive address details. Additionally, the initialization process through `initParameter()` (**Case 16**) sets up a `vivoGnss-whitelist`, enabling whitelisted apps to translate the GPS state into a "fused" state. This advanced mode enriches GPS data by integrating it with information from various other sensors and sources, thereby significantly improving location accuracy.

Observation #5: Each whitelist strategy is meticulously designed to ensure that apps heavily reliant on precise location data can operate seamlessly, unaffected by system memory optimization strategies, or able to access accurate geographical information. However, if exploited by malicious apps, this could lead to unauthorized access to sensitive location data and potential misuse.

5.6. Specialized Capability Provisioning

Apps that are granted function-level special capabilities significantly benefit from an expanded operational scope, gaining access to advanced system features and functionalities not typically available. This special status allows these apps to utilize specific system resources more effectively, such as consistently accessing 5G connectivity for enhanced network performance, implementing custom network policies for improved app communication, or initiating activities from the background seamlessly. Additionally, these special capabilities may include bypassing default system restrictions for sensitive permission usage, thus allowing apps to perform sensitive actions without unnecessary notifications or user prompts. With 770 potential instances observed, **four** examples (Cases 17 to 20 in Table 2) are chosen for an in-depth introduction.

(i) Network Access. Ensuring unrestricted network connectivity can lead to increased power consumption, but many specialized apps heavily rely on this privilege. We have found that some systems implement a network access whitelist to address this issue.

- **Uninterrupted Network Access:** The `Netcoexist` feature (**Case 17**) within Vivo's Android framework is engineered to selectively manage network access for specific apps by maintaining a whitelist. This system ensures that privileged apps like QQ receive uninterrupted network access, optimizing their performance while balancing device security and resource management.
- **5G Enablement:** The `mPackageWhiteList` is meticulously compiled to encompass various apps dedicated to network speed testing and downloading. Working within the `enable5GIfNeeded()` (**Case 18**) function, this whitelist guarantees that the selected apps invariably enjoy 5G connectivity. This special provision exempts them from the power consumption constraints that might limit other apps' access to such high-speed signals.

(ii) Notification Removal for Sensitive Behavior. In **Case 19**, the `setPrimaryClip()` method updates the system's clipboard content and selectively exempts certain apps, such as `com.taobao.taobao` and `com.tmall.wireless`, from dispatching copy operation notifications via the `sendCopyNotify()` function.

(iii) Background Activity Controller. The `Background Activity Controller` plays a pivotal role in managing how apps launch activities from the background, which is crucial for maintaining device performance and the user experience. In **Case 20**, the system utilizes the `getWhiteListFromPermissionManager()` function to identify trusted apps, such as WeChat and QQ, that are permitted to initiate background activities.

Conversely, the `getBlackListFromUCS()` method identifies apps that are restricted from launching such activities to prevent harmful or intrusive behavior. This selective permitting process, facilitated by the `allowStartFromBackground()` function, ensures only whitelisted apps can start activities from the background. Additionally, actions against intrusive advertising, like blacklisting activities from advertising SDKs such as Inmobi, highlight proactive steps taken to curb unwanted background pop-ups and ads, further safeguarding the user experience.

Observation #6: Granting specific apps special functionalities expands their operational capabilities and optimizes the user experience. However, while beneficial, introduces potential security risks by creating opportunities for malicious apps to exploit these privileges. Ensuring a careful balance between providing advanced features to trusted apps and maintaining system security and user privacy is crucial to preventing abuse of these special provisions.

6. Discussion

6.1. Limitations

Dataset. Our study aimed to cover a wide range of smartphone OEMs, analyzing ROMs from 46 different OEMs to conduct a comprehensive vulnerability scan. While there was some brand overlap, like Xiaomi and Redmi, our statistical analysis focused on method signatures. Identical vulnerabilities found across different brands were counted as a single instance, even if ROM source information differed. This approach ensured accurate results based on the vulnerabilities' code rather than brand differences. However, our data collection was limited to popular apps in the US and China, potentially omitting popular apps from other regions.

Approach. We used static backward analysis to obtain string variable and array values, focusing on processes that rely on java standard APIs for string array manipulation or resource file loading. However, it falls short of using non-standard third-party APIs for data processing. Fortunately, our approach is effective in most scenarios, given that the majority of ROM operations depend on java standard APIs for implementing basic functionalities. Furthermore, our analysis uses static approaches, unable to detect dynamic updates in resources over the network. This limitation means potential security assessments might miss real-time changes in ROMs. Despite these shortcomings, our comprehensive static evaluation has identified numerous examples.

Evaluation. While our 5% sample of the 3,078 vulnerabilities analysis did not reveal any false positives (FPs), we acknowledge the theoretical possibility of their occurrence in edge cases. Such instances could potentially arise from intricate code structures that our current methodology might not fully capture.

6.2. LLM Adoption and Classification Challenges

The decision to employ Large Language Models (LLMs) for vulnerability classification was driven by several factors. Static analysis techniques struggle with classifying certain types of vulnerabilities, particularly those related to antivirus functionalities, app persistence, and resource prioritization. These challenges arose primarily due to the lack of clear static attributes associated with such vulnerabilities. In contrast, LLMs have demonstrated significant efficiency in handling code-related tasks [31], offering promising capabilities in code interpretation.

To understand classification challenges, we analyzed the five `Unknown` instances and 203 low-confidence cases (mentioned in §4.3). The reason was that some whitelist-checking functions satisfied the scanning conditions but were not invoked by functional methods. Recalling the motivation example, encountering `checkToScanRisk()` (line 1, Figure 2) led to `isLocalApps()` (line 12) for further scanning, revealing a hardcoded package name comparison at line 17. In these instances, `checkToScanRisk()` was a functional method, while `isLocalApps()` was only a whitelist-checking function without subsequent functional operations. We identified `checkToScanRisk()` as a potential vulnerability categorized as an Antivirus Scan Exemption vulnerability based on its context. The function `isLocalApps()` was only recorded in the vulnerability method `checkToScanRisk`'s call stack. However, when whitelist-checking functions like `isLocalApps()` were not invoked, despite satisfying the static analysis conditions, they could not be effectively categorized with their related privileged functionalities, affecting the LLM's classification accuracy.

Given that our primary goal in using LLM classification was to gain a general understanding of the distribution of **`ITA`** vulnerability types, we considered this 89% accuracy rate to be acceptable. This level of accuracy provides a reliable reference point for the overall landscape of **`ITA`** vulnerability categories, aligning with our objective of obtaining a broad perspective on the distribution of **`ITA`** vulnerability categories.

6.3. Mitigating `ITA` Vulnerabilities

The origins of **`ITA`** vulnerabilities are multifaceted, one of which is the advent of super apps. These apps are not only vast in their scope, incorporating numerous mini-apps [39], but also necessitate continuous operation and network connectivity to ensure timely message reception, alongside supporting voice and video functionalities. In addition, popular apps such as video streaming, online shopping, and navigation may demand specific privileges, like hardware acceleration for video playback or enhanced accuracy in location services for navigation apps. To meet these demands, OEMs have optimized their systems to support and improve app functionality, ensuring smooth operation.

As mentioned in §4.3, 2,497 out of 3,078 vulnerability instances exhibited inconsistencies between the privileged apps and the target regions, further amplifying their impact. This heightened impact can be attributed to the fact that, **when compiling system ROMs for other regions, OEMs may not make extensive changes to the system itself**. Instead, they might simply remove some irrelevant pre-installed third-party apps. Consequently, **privileged app optimizations originally designed for a specific region may**

persist in the ROM, despite those apps being unpopular or unused in the target region. It is important to note that our study encompassed a comprehensive analysis of both Chinese and non-Chinese vendors and apps, without any inherent bias. As illustrated in Figure 6, the top 10 apps associated with vulnerabilities include both Chinese-specific apps and globally popular platforms such as WhatsApp, Facebook, and Instagram. However, our findings reveal a higher frequency of vulnerabilities among Chinese vendors compared to their non-Chinese vendors.

Vulnerabilities arise when third-party privileged apps are absent and system security checks rely only on package name verification without thorough checks. Unlike easily authenticated system apps via methods like UID and system signature verification or maintaining blacklists to block apps with system-specific prefixes (e.g., `com.oppo.xxx`), third-party apps render most detection methods ineffective, except signature verification. We hypothesize that maintaining third-party signatures poses significant challenges for OEMs. Potential delays in updating signatures might impede users from using normal apps. We speculate that this difficulty in balancing signature management with uninterrupted access to super/popular apps could contribute to the occurrence of **`ITA`** vulnerabilities.

6.4. Ethical Concern

Securing vendor acknowledgment proved challenging due to bug bounty program complexities. For example, in two cases, despite reproducing vulnerabilities on multiple OPPO devices with video evidence, the vendor required verification on their latest versions before finally acknowledging the issues and awarding bounties. To facilitate quick vendor acknowledgment and communication, we prioritized our approach from the 3,078 potential vulnerabilities identified. Excluding `Resource-Priority-Management` and `Specialized-Capability-Provisioning` categories, and considering device availability constraints, we carefully selected 22 dynamically verifiable samples spanning four categories for reporting to vendors, aiming for quick validation and acknowledgment.

Despite communication challenges, our efforts yielded positive outcomes. Three OEMs have responded constructively, with four vulnerabilities explicitly acknowledged, accompanied by rewards and ongoing fix implementations. Notably, one vulnerability has been assigned a CVE ID (CVE-2024-47xx). We continue to engage with OEMs to facilitate effective resolution of these vulnerabilities. Furthermore, we have provided a comprehensive list of relevant vulnerability information to all manufacturers involved in our study, ensuring widespread awareness and promoting collaborative security efforts across the Android ecosystem.

7. Related Work

Security risks in Android custom ROM. There has already been extensive work on security issues arising from custom Android ROMs. Hou et al. [29] analyzed security patch discrepancies in custom ROMs, while Possemato et al. [35] investigated security enforcement in binary files, SELinux policies, Android init scripts, and kernel hardening. Aafer et al. [24] performed a differential analysis on custom images to identify inconsistent security features. FIRMSCOPE [28] targeted the vulnerabilities related to privilege escalation found in pre-installed apps. Zhou et al. [43] automated the process of detecting certain types of flaws within customized driver protections. The closest works to our study include the research by Aafer et al.[23] and El-Rewini et al.[27], who introduced the concepts of hanging attribute references and evolution-induced access control vulnerabilities caused by `Residual` APIs, respectively. These vulnerabilities can lead to system components being impersonated. Unlike their focus, our work centers on the privileges directly granted to third-party apps by the system, which are susceptible to exploitation at any moment.

Input Validation. The security implications of input validations have been noted by many security researchers. Invetter [41] combines machine learning and static analysis to locate sensitive input validations within system services that could be problematic. Zhao et al. [42] analyze input validation behavior to uncover hidden functionalities that could pose security risks. In IoT devices and apps, insecure input validation also can pose serious vulnerability threats [44], [26]. Unlike their research, we have conducted an extensive study focusing on the authentication of third-party apps specifically within the ROM's framework, system services, and other security-critical components.

8. Conclusion

In conclusion, our research exposes a security flaw within custom Android ROMs, i.e., **`ITA`** vulnerabilities, revealing that special app privileges are often granted without sufficient authentication. Analyzing 686 ROM images, we discovered 3,085 instances where package names from the super/popular app list were embedded within the ROMs, and only seven instances were properly secured. This led us to uncover 3,078 potential **`ITA`** vulnerabilities. This highlights a pressing need for better security practices in ROM development, urging the Android community to address these vulnerabilities to enhance user protection.

Acknowledgement

We sincerely thank our shepherd and reviewers for their insightful comments. This work was partly supported by the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the National NSF of China (grants No.62302181, 62072046), the Xiaomi Young Talents Program, and the HUSTCSE-FiberHome Joint Research Center for Network Security.

References

[1] "Android phone lag problem finally solved, OPPO gets another big credit," 2018, https://baijiahao.baidu.com/s?id=1614189468389398503&wfr=spider&for=pc.

[2] "AndroidDumps," 2023, https://dumps.tadiphone.dev/dumps.

[3] "Apex," 2023, https://source.android.com/docs/core/ota/apex.

[4] "Appbrain," 2023, https://www.appbrain.com/.

[5] "dynamic partitions," 2023, https://source.android.com/docs/core/ota/dynamic_partitions/implement.

[6] "erofs," 2023, https://source.android.com/docs/core/architecture/kernel/erofs.

[7] "erofs utils," 2023, https://github.com/sekaiacg/erofs-utils.

[8] "extfstools," 2023, https://github.com/nlitsme/extfstools.

[9] "firmwarefile," 2023, https://firmwarefile.com/.

[10] "flashtool," 2023, https://www.flashtool.net/.

[11] "imjtool," 2023, https://newandroidbook.com/tools/imjtool.html.

[12] "oppo decrypt," 2023, https://github.com/bkerler/oppo_decrypt.

[13] "qimai," 2023, https://www.qimai.cn/.

[14] "simg2img," 2023, https://github.com/anestisb/android-simg2img.

[15] "Sony xpericheck," 2023, https://xpericheck.com/.

[16] "The Super App Phenomenon: What You Need to Know," 2023, https://www.linkedin.com/pulse/super-app-phenomenon-what-you-need-know-2023-beyond-vikrant-shukla/.

[17] "Apktool," 2024, https://apktool.org/.

[18] "AppOpsManager," 2024, https://developer.android.com/reference/android/app/AppOpsManager.

[19] "List of custom android distributions," 2024, https://en.wikipedia.org/wiki/List_of_custom_Android_distributions.

[20] "Permissions on Android," 2024, https://developer.android.com/guide/topics/permissions/overview.

[21] "Special permissions," 2024, https://developer.android.com/guide/topics/permissions/overview#special.

[22] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "Acedroid: Normalizing diverse android access control checks for inconsistency detection." in *NDSS*, 2018.

[23] 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*, 2015, pp. 1248–1259.

[24] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android {ROMs} via differential analysis," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1153–1168.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[26] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[27] Z. El-Rewini and Y. Aafer, "Dissecting residual apis in custom android roms," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1598–1611.

[28] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "{FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2379–2396.

[29] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, "Large-scale security measurements on the android firmware ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1257–1268.

[30] Q. Hou, W. Diao, Y. Wang, C. Mao, L. Ying, S. Liu, X. Liu, Y. Li, S. Guo, M. Nie *et al.*, "Can we trust the phone vendors? comprehensive security measurements on the android firmware ecosystem," *IEEE Transactions on Software Engineering*, 2023.

[31] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[32] R. Kumar, A. Virkud, R. S. Raman, A. Prakash, and R. Ensafi, "A large-scale investigation into geodifferences in mobile apps," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1203–1220.

[33] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden, "Investigating country differences in mobile app user behavior and challenges for software engineering," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 40–64, 2014.

[34] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong, "Post-gdpr threat hunting on android phones: dissecting os-level safeguards of user-unresettable identifiers," in *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[35] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio, "Trust, but verify: A longitudinal analysis of android oem compliance and customization," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 87–102.

[36] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework." in *NDSS*, 2016.

[37] StatCounter Global Stats, "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide/, 2023, accessed: 2024-01-17.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[39] C. Wang, Y. Zhang, and Z. Lin, "One size does not fit all: Uncovering and exploiting cross platform discrepant {APIs} in {WeChat}," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6629–6646.

[40] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3079–3092.

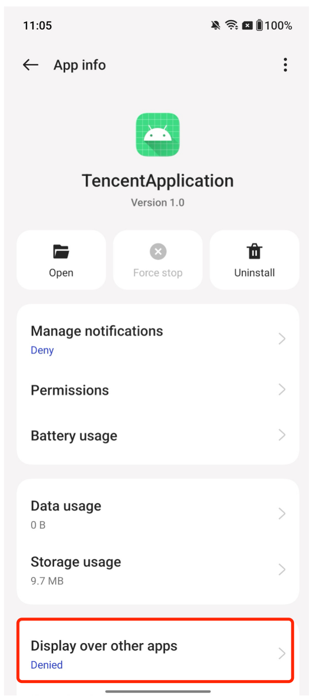
[41] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1165–1178.

[42] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1106–1120.

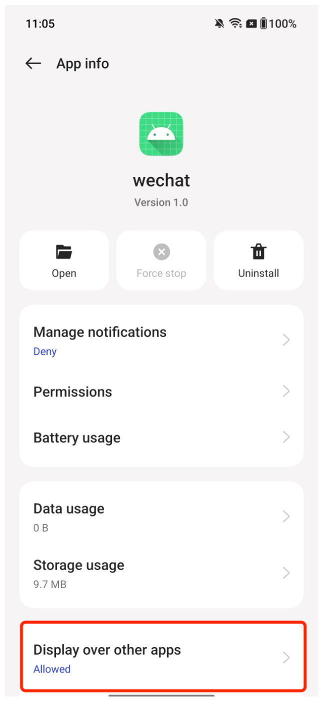
[43] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 409–423.

[44] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1469–1483.

Appendix A. Vulnerability Exploitation for Motivating Examples



(a) The app named "com.tencent.application" with default "denied" display over other apps.

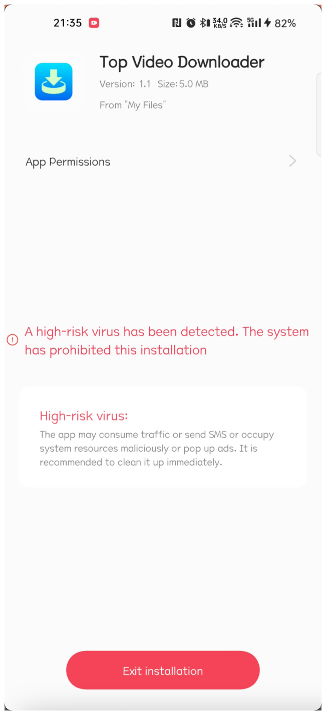


(b) The app named "com.tencent.mm" with default "allowed" display over other apps.

Figure 8: The successful exploitation of the ITA vulnerability in Example I of §2.3.

Example I: Exploiting Permission Control Vulnerability. We developed two apps that requested special permission `android.permission.SYSTEM_ALERT_WINDOW`. One, with the package name `"com.tencent.mm"`, is on the whitelist, and the other, named `"com.tencent.application"`, is not. Apart from their package names, the two apps are implemented identically. After installation, `"com.tencent.mm"` automatically granted the privilege to display over other apps (Figure 8b), while the other app was disallowed (Figure 8a).

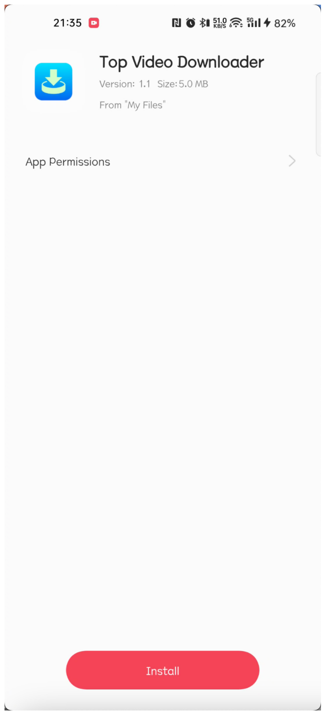
Example II: Bypassing Antivirus. We utilized Apktool [17] to repackage a malicious app, altering its package name to `com.alipay.security.mobile.authenticator`, a package name listed in the `DEFAULT_WHITE_APP_LIST` in line 11 of Figure 2 (`com.android.packageinstaller` is a system app and cannot be impersonated). We then proceeded to install the original malware and the modified version sequentially for testing. We observed that the original



A high-risk virus has been detected. The system has prohibited this installation

High-risk virus:
The app may consume traffic or send SMS or occupy system resources maliciously or pop up ads. It is recommended to clean it up immediately.

(a) The original malware that can not be installed.



(b) The malware with the modified package name successfully bypasses the virus scan.

Figure 9: The successful exploitation of the ITA vulnerability in Example II of §2.3.

malware installation was delayed for approximately five seconds for a virus scan, and ultimately the system prevented its installation due to the presence of a high-risk virus (Figure 9a). However, the app with the modified package name bypassed the virus scan entirely and proceeded directly to the standard installation interface (Figure 9b).

Appendix B. Distribution of Top 50 Super/Popular Apps

The distribution of the top apps, as indicated by their appearance frequency, provides insight into the popularity and usage of these apps across the dataset. We now discuss the characteristics of this distribution according to Figure 10.

Dominance of Social Media and Messaging Platforms. The list is dominated by prominent social media and messaging apps such as WeChat (`com.tencent.mm`) with 1,947 appearances, QQ (`com.tencent.mobileqq`) with 748 appearances, and WhatsApp (`com.whatsapp`) with 320 appearances. This highlights the global and regional dominance of these platforms for communication.

Variety of App Categories. The appearance frequency distribution encompasses a diverse array of app categories, extending beyond the confines of social media and messaging. It includes video streaming platforms such as Tencent Video (`com.tencent.qqlive`) and Youku (`com.youku.phone`), online shopping portals like Taobao (`com.taobao.taobao`) and Pinduoduo (`com.xunmeng.pinduoduo`, internationally known as Temu's Chinese version), navigation services exemplified by Amap (`com.autonavi.minimap`), and entertainment apps, notably Bilibili (`tv.danmaku.bili`), a widely recognized Chinese video sharing platform.

Market Influence Indicated by High Appearance Frequency. Apps with the highest frequency of appearances, such as WeChat, QQ, and WhatsApp, underscore their considerable influence within the market. Specifically, the prominent positioning of WeChat and QQ highlights their pivotal role in shaping China's digital landscape, whereas WhatsApp's notable presence underscores its extensive global footprint.

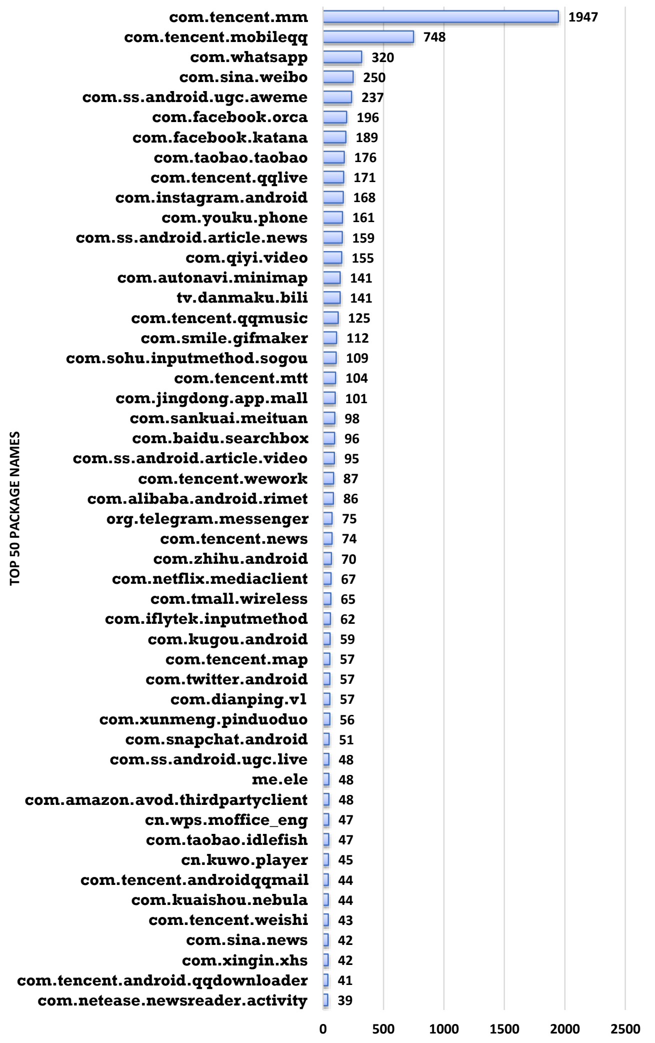


Figure 10: Distribution of the `ITA` vulnerability appearance frequency of the top 50 apps.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper investigates security vulnerabilities within custom Android ROMs, focusing on Implicit Trust Apps (**`ITA`**) vulnerabilities. The authors analyze 686 custom ROMs from 46 OEMs, uncovering thousands of potential vulnerabilities where special privileges are granted to third-party apps without proper authentication.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

- The paper makes important contributions to the field by identifying and analyzing **`ITA`** vulnerabilities in custom Android ROMs, which expose users to significant security risks.
- The authors conducted an extensive analysis, uncovering over 3,000 vulnerabilities, and disclosed these to vendors, with several being confirmed and addressed.
- The use of static analysis tools like FlowDroid and Soot to identify these vulnerabilities presents a useful methodology for future research.

C.4. Noteworthy Concerns

1) More proof-of-concept exploits should be provided to enhance the impact of the findings
2) The study is largely limited to Chinese vendors and apps, which reduces the generalizability of its findings

Appendix D. Response to the Meta-Review

We thank the anonymous shepherd and reviewers for their insightful comments. Regarding more proof-of-concept exploits, we acknowledge limitations in providing extensive demonstrations due to difficulties in obtaining specific devices and ROM versions. We are committed to expanding our testing as more devices become available and to continuing vendor communications. Concerning the limits on Chinese vendors and apps, as noted in §6.3, our study aimed for comprehensiveness and our study encompassed both Chinese and non-Chinese vendors and apps. Our findings objectively show a higher vulnerability frequency in Chinese vendors compared to non-Chinese ones.