# Risky Cohabitation: Understanding and Addressing Over-privilege Risks of Commodity Application Virtualization Platforms in Android

Shou-Ching Hsiao
d10922007@csie.ntu.edu.tw
National Taiwan University
Taipei, Taiwan

Shih-Wei Li
shihwei@csie.ntu.edu.tw
National Taiwan University
Taipei, Taiwan

Hsu-Chun Hsiao
hchsiao@csie.ntu.edu.tw
National Taiwan University &
Academia Sinica
Taipei, Taiwan

## ABSTRACT

The Android system protects its users' privacy via app permissions, which govern apps' access to sensitive data and resources. However, recent research has reported that, during app virtualization, the current Android permission model fails to prevent illegal permission usage: apps can exploit the User ID shared among co-hosted apps in the same virtualized environment to perform unauthorized actions. To the best of our knowledge, such over-privilege issues have not been thoroughly investigated; neither has a practical defense proposed to address them.

To fill those gaps, this paper introduces a taxonomy of over-privilege issues in the context of app virtualization and presents an automated tool called PermLabel to investigate their prevalence and impact in real-world apps. By testing 826 real-world apps, PermLabel identified 254 (31%) as having over-privilege issues. Among the found apps, 244 (96%) have at least one over-privileged runtime permission, which poses a substantial risk to user privacy. Our analysis of these identified cases uncovered common causes and abuses of over-privileged permissions. Notably, the majority of such permissions grant access to sensitive data such as location, camera, and phone state. These issues predominantly stem from the app's code and third-party libraries intended for advertising and analytics. To restore compliance with the Android permission model, we also propose a defense solution, PermSep, that enforces fine-grained permission separation among co-hosted apps while preserving normal app-virtualization usage. Evaluation of PermSep on 17 virtualization apps shows successful blocking of over-privilege, achieving desired properties not attainable by previous work.

## CCS CONCEPTS

• **Security and privacy → Software and application security**;

## KEYWORDS

Over-privilege, App Virtualization, Android Permissions

**ACM Reference Format:**
Shou-Ching Hsiao, Shih-Wei Li, and Hsu-Chun Hsiao. 2024. Risky Cohabitation: Understanding and Addressing Over-privilege Risks of Commodity Application Virtualization Platforms in Android. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/3626232.3653274

*CODASPY '24, June 19–21, 2024, Porto, Portugal*
© 2024 ACM.
ACM ISBN 979-8-4007-0421-5/24/06
https://doi.org/3626232.3653274

## 1 INTRODUCTION

In 2022, the Android system had more than 2.5 billion users [8]. However, the current Android system only supports running one instance of a given app at a time, so users frequently need to log in and out to switch between different accounts, or use multiple mobile devices across various work and leisure scenarios. A recent solution known as *app virtualization* addresses the limitation and allows users to run multiple instances of the same app on one device simultaneously. App-virtualization platforms rely on a *host app* to provide a virtualized environment to execute *plugin apps*. Users can download apps from various app markets and run them as plugins in the host app without modification. These host apps aim to support the execution of Android apps seamlessly and preserve their functionality. The most popular host apps, Dual Space [10] and Parallel Space [22], have been widely used and downloaded from Google Play more than 100 million times.

**Over-privilege during App Virtualization.** Android adopts a permission-based security model, demanding that apps obtain explicit permission to access system resources. Permissions must be declared during installation, with runtime consent requested for potentially dangerous permissions. To enforce this model, Android assigns a unique User ID (UID) to each installed app and manages a list of permissions for each UID. However, recent research [9, 25, 31, 13] highlights a significant security challenge in app-virtualization environments. In such settings, popular host app implementations share the same UID with plugin apps, rendering Android's UID-based permission model unable to differentiate between the permissions of the plugin and host apps. This results in the *over-privilege* problem, where permissions granted to an app—be it a plugin or host app—can be exploited by all cohabitating apps in the virtualization environment. An app is deemed over-privileged if it can perform permission-required operations without declaring or requesting the permissions. Notably, by not declaring permissions, the app evades permission-based security vetting [11, 14] and malware detection [4, 29]. By not requesting permissions, the app further sidesteps user consent. Consequently, over-privileged apps can stealthily access restricted data, such as monitoring the user's location and recording audio, thereby greatly invading user privacy. Therefore, there is an urgent need to investigate the over-privilege problem in real-world apps and develop a defense solution to address the security challenge in the app-virtualization environment.

**Limitations of prior investigation and defense.** Several recent studies [9, 13, 25, 31] touched upon the over-privilege problem. However, these studies were either confined to proof-of-concept examples or lacked comprehensive coverage of all potential types of over-privilege. For instance, PluginPermCheck [13] identified cases of plugin apps exploiting undeclared permissions but did not consider permissions requested at runtime. Other than investigation, some studies have proposed defenses to prevent the problem of overprivilege. However, they often struggled to strike a balance between security and functionality. For example, some defense approaches focused on detecting malicious host or plugin apps through behavioral monitoring [26] or analyzing mismatched certificates [31]. Unfortunately, they overlook scenarios where benign host and plugin apps inadvertently contribute to over-privilege issues. Other defense approaches aimed to protect benign plugin apps by enabling them to identify whether they operate within app-virtualization environments; if detected, these apps halt execution [9, 19]. However, this hinders the use of benign apps, undermining the intended functionality. Projects like Boxify [6] and NJAS [7] leveraged app virtualization to build a sandboxed environment to isolate rogue apps. They employed a proxy-based approach and relied on a user-level runtime to proxy interactions between the sandboxed app and the operating system. These works did not aim to address over-privilege. To address the issue, they could potentially be extended to ensure that a plugin app's access to resources aligns with the required permission. However, these approaches require high engineering efforts to interpose apps' resource accesses and customize permission models. They cannot easily scale to support different stock Android apps using various API versions with different permission requirements.

**Our Work.** Our goal is to thoroughly investigate and address the over-privilege problem in real Android apps. We aim to address the following research questions: **RQ1**: How prevalent are over-privilege issues in real-world apps? **RQ2**: What are the code's origins leading to these over-privilege issues? **RQ3**: What behaviors ensued from the over-privileged permissions? **RQ4**: What are over-privileged permissions' practical consequences and root causes? To answer these questions, we presented a taxonomy of over-privilege issues, considering the misalignment among declared, requested, and required permissions from both host and plugin apps' perspectives. Then, we developed an automated tool for detecting the defined issues. When testing 630 popular apps from different app stores and 196 malware samples, we found that 194 (31%) apps and 60 (31%) malware samples have over-privilege issues. Nearly all of them have at least one over-privileged *runtime* (dangerous) permission, signifying significant risks to user privacy (**RQ1**). The over-privilege issues stem from both the application's code and various libraries. Additionally, over-privileged libraries present more significant privacy threats in virtualization environments due to their broader permissions and the potential for stealthy abuse (**RQ2**). The most common over-privileged permissions are for accessing device location or retrieving phone state (**RQ3**). We also observed some malware samples that potentially abuse over-privilege to enhance malicious operations, and apps' third-party libraries are a primary root cause of over-privilege problems (**RQ4**). These results confirm the risks of over-privilege during app virtualization.

Based on our findings, we introduced a defense solution, PermSep, to support fine-grained permission isolation. We aim to support off-the-shelf host apps instead of implementing a new host app, to proxy plugin apps' resource accesses to prevent over-privilege. The latter limits portability and usability. PermSep extends the Android framework to leverage its existing permission model and enforcement mechanisms against plugin apps in an app-virtualization environment. To ensure both security and functionality, PermSep distinguishes the identity of plugin apps from host apps and aligns the permission of plugin apps with corresponding native apps—most host apps launch a plugin app by cloning from its *native app*.

In sum, this paper makes the following contributions:

- *Comprehensive study on over-privilege risks.* It presents the first systematic evaluation of the over-privilege problem within the app-virtualization environment, addressing risks overlooked by previous research.
- *New analysis tool.* It develops PermLabel, an automated tool to discover both over-privileged install-time and runtime permissions in Android apps.
- *Real-world evaluations.* It uncovers common causes and abuses of over-privileged permissions in real-world apps, confirming their prevalence and substantial privacy impact in the real world.
- *Practical defense solution.* It proposes PermSep, a defense solution to prevent over-privilege by separating permission between the host and plugin apps, which can be used with off-the-shelf host apps. The evaluation of PermSep prototype shows that it effectively prevents over-privilege and achieves the desired properties of ensuring accurate runtime permission context and the normal functionalities of both install-time and runtime permissions lacking in the previous work.

## 2 BACKGROUND

### 2.1 Android Permission Model

The Android OS delegates resources based on User IDs (UIDs) and enforces access control by checking the UID's authority. Each Android app obtains a unique UID upon installation, and by default, only limited resources are granted to that UID. To extend access beyond such restricted set of resources, apps should obtain specific permissions. Below, we describe the two primary types of permissions and the prerequisites to obtain them.

**Install-time vs. runtime permission.** Since the release of Android 6.0 (API 23), permission has been divided into install-time and runtime permissions. Install-time permissions manage resources that impinge less on user privacy, such as accessing the Internet, retrieving Wi-Fi or Bluetooth statuses, and using extensions of location providers. Runtime permissions, on the other hand, manage resources that are more privacy-sensitive, such as accessing location data, recording audio, and accessing camera hardware.

**Prerequisites to obtain permissions**. Before using permissions, apps should obtain permission first; otherwise, the system will block that permission usage. To use install-time permissions, app developers should *declare* them in the app manifest file. Then, the declared permissions are granted to the app automatically upon installation. To use runtime permissions, on the other hand, developers should both *declare* them in the manifest file and *request* runtime user consent. In contrast to install-time permissions, the core purpose of requesting runtime consent is to clearly associate permissions with context (e.g., the exact usage time and the app name), and thus enable users to make informed decisions about whether to grant permission. Google Play Console, a platform that Google provides for app developers, emphasizes that permission should only be used in alignment with the context the user has consented to [12].

**System Components that Handle Permissions.** The Android permission model relies on the Package Installer (PI) and Activity Manager Service (AMS) to handle permissions. We introduce these system components briefly to enhance comprehension of our measurement tool and defense solution, which interact with them.

PI, a system application, handles permission requests and renders permission-request dialog boxes on the user interface. AMS functions as a background system service responsible for validating a UID's permission status. AMS operates in two distinct scenarios: *permission enforcement* and *permission self-checking*. Permission enforcement occurs when a service (e.g., Location Manager Service) validates the permission statuses of the caller before serving permission-required API calls. Conversely, permission self-checking occurs when an app scrutinizes its own permission statuses before executing a permission-required API call. In cases where the corresponding permissions are not granted yet, the app should request them before calling. Understanding these call paths enables us to accurately identify *required permissions*, i.e., permissions that the app must have in order to call specific APIs, as we will explain in the following sections.

## 2.2 App Virtualization

App virtualization involves a **host app** and **plugin apps**. The former creates an app-level sandbox-like environment in which various plugin apps can execute. While the specific implementations may differ, we note four common features of commodity host apps, presented below. This commonality may be attributed to the fact that a considerable number of them [9] are built upon two open-source frameworks, VirtualApp [5] and DroidPlugin [23].

**Cloning native apps.** Users can clone any app as a plugin app without modifying those apps. To distinguish the same app between the installed one (running on the Android OS) and the virtualized one (running in the host app), we call the prior as the *native app* and the latter as the plugin app.

**Acting as a proxy.** Because plugin apps are not installed on the Android OS, they are conceptually invisible to the system. In order to execute normally, these plugin apps rely on the host app to proxy their requests to the intended Android systems services. As Figure 1 illustrates, the requests made by plugin apps can be broadly categorized into two types, depending on whether they are proxied transparency or routed through a host-server process. The first type (❶) involves straightforward parameter modifications, while the second type (❷) needs to be dispatched to a host-server process to customize both the sending and returning parts, e.g., launching plugin apps. The second type introduces additional IPC calls and an extra layer of indirection compared to the first type, which complicates the identification of the actual initiators. Understanding the two request types and the extra layer of indirection in the second type is essential for us to devise methods to correctly attribute permission requests to the plugin apps.

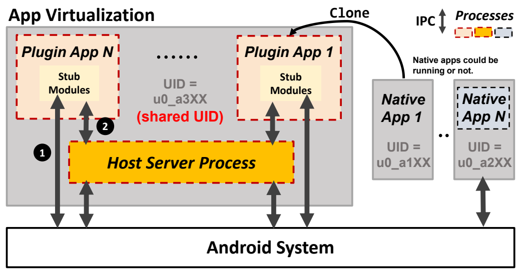


Figure 1: App virtualization vs. native apps.

**Sharing UID with all plugin apps.** The host app uses itself as a placeholder to help the plugin app launch and instantiate processes. Due to this mechanism, the plugin app is executed inside the process initially assigned to the host app, therefore sharing the UID.

**Overclaiming permissions.** Android OS regards all permission operations executed with this shared UID as emanating from the host app. To support any plugin app the user may load in the future, host apps typically declare as many permissions as possible. The most popular host apps, Dual Space and Parallel Space, respectively declared 102 and 94 system permissions in advance [31], which cover almost all permissions on the system (e.g., Android 9 has 103 permissions). Many host apps also preemptively request runtime permissions. For example, among the 17 host apps we tested, 14, 8, and 1 of them request `Storage`, `Telephone`, and `Location` at launch, respectively.

## 2.3 Problem Description: Over-privilege

Ideally, only apps following Android's permission requirements should be allowed to access permission-required resources. However, due to host apps' insecure practices of *sharing UID* and *overclaiming permissions* during app virtualization, a host app may obtain permissions that are not requested, and plugin apps may obtain permissions that are not declared or not requested.

We aim to understand and address **over-privilege risks during app virtualization**. In the context of app virtualization, an application is deemed *over-privileged* when it can stealthily utilize extra permissions while hosting or being hosted by other applications. The term *stealthily* is employed to signify that the app neither explicitly declared these permissions in its manifest file nor initiated any permission-request dialog boxes, but opportunistically uses permissions granted to apps habituating in the same virtualization environment. Over-privileged apps can circumvent the existing security mechanisms, such as app vetting by the app stores and user consent through permission-request dialog boxes, as there are no apparent disclosures or indications.

The exploitation of over-privilege poses severe consequences. Malicious apps or third-party libraries can circumvent existing security mechanisms and covertly escalate privileges when operating within virtualization environments. Even security-conscious app developers who diligently leverage the Android permission model to limit third-party library access may find such protection ineffective when their apps are executed within virtualization environments.

*Assumptions.* We assume the underlying operating system is benign. We consider over-privilege by *host apps* and *plugin apps*, which occurs either inadvertently or through malicious intent. To categorize these cases, we present a taxonomy based on the type of permission (install-time or runtime) and its authorization status (declared and/or requested). The categories are summarized in Figure 2. For ease of description, these categories are prefixed by the app type (host or plugin) when applicable. The green arrow represents apps following Android's permission requirements, while the red arrow means the opposite. When apps run on the Android OS (case (a) in Figure 2), they cannot use permissions in those red-arrow cases. However, in an app-virtualization environment, there are different over-privileged categories that red-arrow cases can lead to permission abuse (case (b) and (c) in Figure 2).

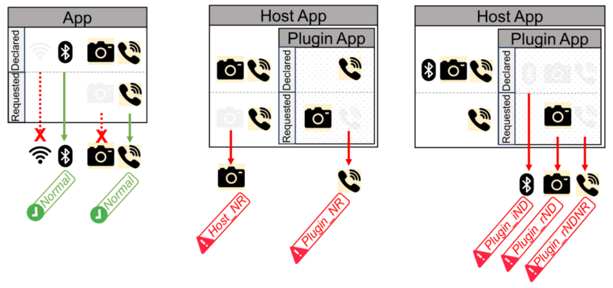
**Host apps**. We assume host apps provide typical app virtualization functionalities but may sneak on using runtime permissions that were initially declared for their hosted plugin apps. Over-privilege occurs when the host app abuses permissions once they have been granted to plugin apps by the system, which is categorized as NR ("not requested"). The categories under "not declared" are inapplicable to host apps because permissions can only be declared during the installation of the host apps, and the permissions specified in the manifest file of plugin apps have no impact. In contrast, host apps can leverage permissions requested by plugin apps at runtime. We do not consider host apps maliciously exploiting the app-virtualization functionalities to request permissions in disguise of plugin apps, i.e., through injecting malicious payload into plugin app processes.

**Plugin apps**. We consider four cases for plugin apps that can lead to over-privilege of install-time and runtime permissions. If a plugin app uses permission that the host app has declared but itself does not, that install-time/runtime permission is classified as `iND`/`rND` (case (c) in Figure 2). If a plugin app does not request runtime permission before use, that permission is classified as `NR`. The over-privilege occurs once the host app or co-hosted plugin apps in the same app-virtualization environment already request the same runtime permission and obtain consent with their permission context. As stated earlier, the host apps we studied commonly request `Storage`, `Telephone`, and `Location` permissions before any plugin apps execute. If a plugin app neither declares nor requests runtime permissions before use, the permission is categorized as `rNDNR`.

During app virtualization, a host or plugin app can utilize permission self-checking APIs to check the permission statuses of the shared UID. By doing so, apps can remain stealthy and avoid triggering exceptions when attempting to abuse over-privileged permissions that have not been granted.

## 3 INVESTIGATING OVER-PRIVILEGE DURING APP VIRTUALIZATION

Having defined a comprehensive taxonomy for over-privilege issues during application virtualization, we sought to find out which if any of them exist in real-world apps. To that end, we developed PermLabel, an automated tool designed to unveil over-privilege issues. This section delineates PermLabel's design (Section 3.1), highlights key findings from testing over 800 apps across multiple app markets (Section 3.2), and provides an in-depth analysis covering each over-privilege type (Section 3.3).



**(a) Normal** **(b) Not Requested** **(c) Not Declared**

**Figure 2: Taxonomy of Over-privilege. A required permission can be mapped into one of these categories based on its type (install-time or runtime) and authorization status (declared and/or requested).**

### 3.1 PermLabel

As outlined in Section 2.1, install-time permissions must be declared before use, and runtime permissions must be declared and requested before use. Violations of these prerequisites may lead to one or more over-privilege issues as defined in our taxonomy.

Consequently, to assess an app's exposure to over-privilege issues, it is essential to ascertain whether and when the app declares, requests, and utilizes permissions.

PermLabel adopts established methods for identifying declared and requested permissions [28]. Declared permissions are retrieved from the app manifest file parsed upon installation, and categorized into install time or runtime based on the protection levels indicated by the Android OS. Requested permissions are obtained by dynamically executing the app, extracting permission names from intents sent through AMS, as Android OS transforms every permission request into an intent encoding the permission.

The primary technical challenge faced by PermLabel lies in identifying permission use (i.e., the *required permissions*). Existing static code analysis methods encounter two critical limitations. Firstly, popular app-virtualization platforms often employ obfuscation, rendering static code analysis ineffective. Secondly, static analysis assumes a permission-to-API mapping for which no official document exists, potentially impacting the evaluation's precision.

To address these limitations, PermLabel uncovers required permissions by monitoring calls invoking AMS and identifying those performing **permission enforcement**. The subtlety lies in accurately distinguishing permission enforcement from permission self-checking, both of which invoke AMS, but the latter has yet to call permission-required APIs. PermLabel dynamically executes apps, combining information on the timestamp, target UID, permission name and type, and the call stacks generated by the stack-tracing function to make this distinction accurately. Further details will be presented in Section 3.1.2.

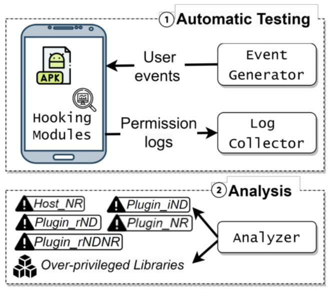


Figure 3: PermLabel workflow.

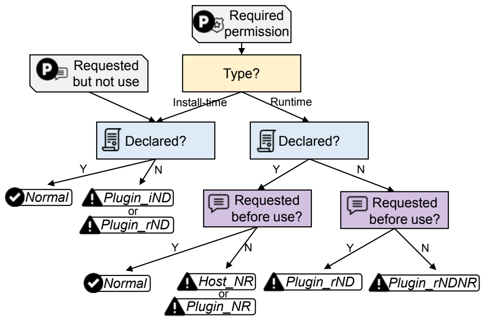


Figure 4: Flow chart of permissions labeling.

While PermLabel's dynamic approach effectively overcomes code obfuscation and the lack of reliable permission-to-API mapping, it is acknowledged that it may potentially underestimate over-privilege issues due to partial code coverage during testing, as we will discuss in Section 5.

*3.1.1 PermLabel Workflow.* PermLabel's workflow comprises two stages—automatic testing and analysis—as shown in Figure 3. The automatic-testing stage incorporates a user-event generator and collects permission logs used for permission labeling. With the collected information, the analysis stage inspects whether an app violates the permission model by finding misalignment among the declared, requested, and required permissions. It then labels the app with corresponding over-privilege issues and the possible culprits (e.g., which libraries) of the over-privilege issues.

**Automatic Testing**. PermLabel includes the hooking modules on the mobile device and the event generator and log collector on the computer. The hooking modules log permission enforcement and request events. We implemented hooks on the AMS interfaces to ensure comprehensive log collection, which centrally manages all permission-related events. The log information comprises the timestamp, target UID, permission name and type, and call stacks, generated by the stack-tracing function. The call stacks identify required permissions and determine potential code origins of over-privilege issues.

On the computer side, PermLabel incorporated DroidBot [17], a widely used tool to generate user events for the tested apps and trigger permission-required operations. Then, the log collector identifies the permission events generated by hooking modules through the tested app's UID and records them to log files.

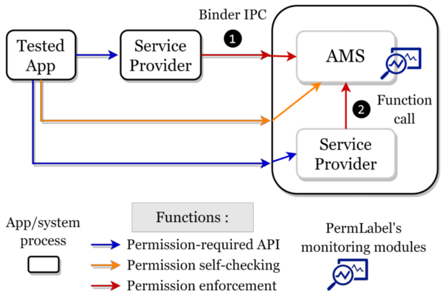


Figure 5: Permission self-checking and enforcement.

**Analysis**. After finishing app testing, we label permissions in the collected logs. The first step in the labeling process was to parse the raw logs and transform our focused permission information into structured events, which include the declared, requested, and required permission sets. Because we need to check the order in which permission events occurred, we bind each event with a timestamp. Finally, PermLabel iterates through each permission item in the *required* set and *requested* set using the decision diagram shown in Figure 4. If a required permission is labeled as over-privilege, we can infer that the tested app includes operations that can lead to over-privilege during app virtualization. Additionally, given that runtime permissions are considered dangerous, we also verify whether they are requested without being explicitly declared, regardless of whether our tool identifies them as required. This precaution is taken to account for the possibility that the app might utilize these permissions beyond our testing scope or after software updates.

To provide further insights into over-privileged permissions, PermLabel examines the call stacks leading to the permission usage, which may originate from either the tested application or the library code. Specifically, we trace to the beginning of the call stack after filtering out the known system packages. We manually curate a library dataset that contains the information of the library name, its package name, and the library class. Then, we classify the library based on the package name extracted from the call stack. If the calling function is not in the dataset but is a subset of the app package name, we identify the caller as the tested app itself.

*3.1.2 Identification of Required Permissions.* To extract required permissions, we leverage an observation that permission-required calls from apps always trigger permission enforcement at the service provider's end. Any app or service that receives permission-required calls on the Android OS can act as the service provider. Instead of monitoring all these service providers and presumably omitting some providers/events, PermLabel collects these enforced permissions through the AMS.

As shown in Figure 5, the AMS serves permission self-checking calls and performs permission enforcement for permission-required operations. PermLabel must distinguish the two from the collected log. In Figure 5, ❶ and ❷ are the two permission-enforcement calls that PermLabel records. Due to their different implementations, we can identify the callers by inspecting the **Binder client** for ❶ and **function call stacks** for ❷. In the **first** type, service providers do not execute in the same process as the AMS, so they launch a Binder

IPC to call permission-enforcement functions in the AMS. For example, the location system service enforces location permission by launching a Binder IPC to the AMS whenever an app requests the device's location. Accordingly, PermLabel can identify this type of permission enforcement as one in which the checking target is the tested app, and the AMS's caller (the Binder client) is not. In the **second** type, service providers are executed in the same process as the AMS, so the Binder client *is* the tested app for both permission enforcement and self-checking. Thus, we should also check whether the AMS's call stack contains a permission-enforcement path to avoid recording permission self-checking.

## 3.2 Results

**App collection.** To ensure the representativeness of our evaluation, we carefully chose our app samples through the following process. We selected popular host apps with over one million downloads from Google Play, resulting in 17 apps. Due to the space constraint, the details of the dataset and implementation will be available at https://github.com/csienslab/overprivilege-app-virtualization. We adopted a sampling approach to curating the plugin apps, selecting 10% from the top 100 apps in each category from five widely used app stores. After excluding apps that our tool could not execute, we arrived at 630 apps. The distribution across app stores is as follows: Google Play (189 apps), Huawei Market (88), 360 Market (119), Tencent Myapp (50), and Wandoujia (184).

We also collected malware as our test targets. Although users would not intentionally install malware, over-privilege may help malware bypass the app store's vetting and camouflage as benign apps, increasing its chances of being loaded by users. For malware, we selected 196 samples detected by at least five anti-virus engines from Androzoo [2]. This selection broadly represents both benign and malicious apps. To our knowledge, none of such malware has been reported for targeting app-virtualization settings.

**PermLabel settings.** With PermLabel deployed, we executed these samples on an ASUS Zenfone M2 (X01AD) running Android 9.0. We ran each app for 40 minutes using DroidBot's greedy depth-first search policies.

**RQ1**: How prevalent are over-privilege issues in real-world apps?

For plugin apps, we found that 194 apps (31%) and 60 malware samples (31%) had *at least one* labeled permission. Furthermore, among the 194 apps, 25 was from Google Play (13% of all the tested Google Play apps), 25 was from Huawei Market (28%), 63 was from 360 Market (53%), 15 was from Tencent Myapp (30%), and 66 was from Wandoujia (36%). Among the apps and malware with over-privilege issues, 186 (96%) and 58 (97%) of them respectively had at least one over-privileged *runtime* permission (i.e., those except `Plugin_iND`). The significant number of runtime permissions indicates the potential invasion of user privacy caused by over-privilege. Due to the presence of multiple over-privileged permissions in certain apps and malware, the cumulative count of permissions exceeds the number of apps, which amounts to 250 for apps and 84 for malware.

As for host apps, even though they have overclaimed many install-time and runtime permissions, we found that 8 out of 17 host apps still attempt to use `LOCATION` runtime permission without request (`Host_NR`).

**Result 1**: The high proportion of over-privileged *runtime* permissions implies negative impact on user privacy.

**RQ2**: What are the code's origins leading to these over-privilege issues?

We analyzed the cause of over-privilege by tracing the root of permission-required call stacks.

We manually classified the libraries into 12 categories according to their functionalities (e.g., the information on public repositories [20]). Apps that obfuscate their package names are excluded from this analysis due to the lack of information. In total, 74 kinds of libraries were labeled. The results indicate that the app's code and various in-app components (libraries) can contribute to the identified issues. While it is recognized that libraries can abuse permissions granted to the apps [24, 30, 32], we underscore that such abuse is more pronounced in the context of app virtualization. This is because, during app virtualization, libraries can use not only the permissions granted to the specific app but also those of all apps (including the host and other plugin apps) running in the same virtualized environment. This results in a *much broader over-privileged permission scope*. In particular, since host apps typically preemptively declare almost all permissions, libraries will be able to use these permissions stealthily without declaration.

Figure 6 compares the four over-privilege types caused by different library types. In addition to the app's code, we find that a high percentage of over-privilege leveraging `NR` is caused by Advertising/Analytics libraries (18 different kinds of libraries were found) and Utility libraries (13 different kinds were found). Specifically, these libraries attempt to track users by using device information (requiring `READ_PHONE_STATE`) and provide customized services by obtaining users' locations (requiring `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION`), whereas the apps using these libraries do not request permissions. This suggests that the tracking fails on the Android OS but may succeed during app virtualization, resulting in over-privilege.

Some over-privileged permissions are observed to be used by the app and libraries within the same app. We found that 52 apps used the same over-privileged permission multiple times.

**Result 2**: The over-privilege problem is primarily caused by libraries, posing more substantial privacy threats in virtualization environments than native ones.

**RQ3**: What behaviors ensued from the over-privileged permissions?

We further listed the permissions and the corresponding APIs that are found used by apps in Table 1. The mapping was assembled from the Android API reference [3]. Though not exhaustive due to the absence of an official mapping, we utilize it to offer insights into the ramifications of exploiting these over-privileged permissions.

This mapping shows that apps and malware could abuse over-privileged permissions to call multiple APIs unauthorizedly. For example, the install-time permission `CHANGE_WIFI_STATE` and `BLUETOOTH_ADMIN` could be abused to modify the Wi-Fi state and Bluetooth settings, respectively. For abusing runtime permissions, we have pointed out the prevalence of obtaining user location and phone states in the previous section. Additionally, several malware

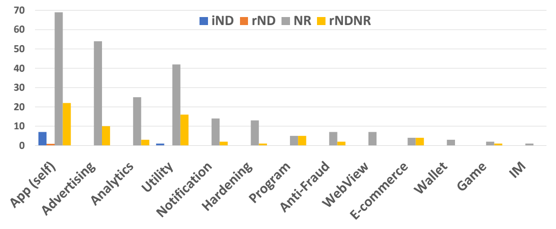


**Figure 6: The number of over-privileged permissions caused by apps and various types of libraries.**

**Table 1: Over-privileged permissions and APIs.**

| Permission | GP | HM | 360 | TM | WJ | Mal | Permission-required APIs |
|---|---|---|---|---|---|---|---|
| ACCESS_WIFI_STATE | 0 | 0 | 2 | 0 | 2 | 1 | getConnectionInfo, getWifiState |
| CHANGE_WIFI_STATE | 1 | 1 | 1 | 1 | 0 | 2 | startScan |
| BLUETOOTH_ADMIN | 2 | 0 | 3 | 2 | 0 | 0 | startDiscovery, startScan, stopScan |
| ACCESS_LOCATION_EXTRA_COMMANDS | 1 | 1 | 3 | 0 | 0 | 3 | sendExtraCommand |
| RESTART_PACKAGES | 0 | 0 | 0 | 0 | 3 | 0 | restartPackage |
| KILL_BACKGROUND_PROCESSES | 0 | 0 | 0 | 0 | 2 | 0 | killBackgroundProcesses |
| ACCESS_FINE_LOCATION | 1 | 0 | 1 | 0 | 1 | 0 | getLastKnownLocation |
| ACCESS_COARSE_LOCATION | 5 | 5 | 9 | 7 | 10 | 7 | requestPermissions, startScan, getAllCellInfo, getCellLocation |
| CAMERA | 10 | 15 | 2 | 4 | 9 | 4 | getCameraInfo, open |
| READ_EXTERNAL_STORAGE | 1 | 0 | 0 | 0 | 0 | 0 | requestPermissions |
| WRITE_EXTERNAL_STORAGE | 0 | 1 | 0 | 0 | 0 | 1 | requestPermissions |
| READ_PHONE_NUMBERS | 1 | 1 | 0 | 0 | 0 | 4 | getLine1Number |
| READ_PHONE_STATE | 15 | 21 | 79 | 14 | 85 | 51 | requestPermissions, getRunningAppProcesses, getSubscriberId, getActiveSubscriptionInfo |
| READ_SMS | 1 | 1 | 0 | 0 | 0 | 4 | getLine1Number |
| RECORD_AUDIO | 0 | 0 | 1 | 0 | 1 | 1 | requestPermissions |
| READ_CALL_LOG | 0 | 0 | 0 | 0 | 0 | 1 | requestPermissions |
| WRITE_CALL_LOG | 0 | 0 | 0 | 0 | 0 | 1 | requestPermissions |
| PROCESS_OUTGOING_CALLS | 0 | 0 | 0 | 0 | 0 | 1 | requestPermissions |
| CALL_PHONE | 0 | 0 | 0 | 0 | 0 | 2 | requestPermissions |
| USE_SIP | 0 | 0 | 0 | 0 | 0 | 1 | requestPermissions |

**GP**: Google Play, **HM**: Hwauei Market. **360**: 360 Market, **TM**: Tencent MyApp, **WJ**: Wandoujia, **Mal**: Malware

samples may have attempted to evade app vetting and hide their malicious intention of managing phone calls and accessing phone IDs, as we found several over-privileged permissions labeled as `rND` (e.g., `CALL_PHONE` or `READ_CALL_LOG`) aroused by `requestPermissions`. In other words, these malware can evade permission-based malware detection but perform these suspicious operations in an app-virtualization environment.

Worse, we found 38 known apps and 11 malware had multiple labeled permissions; some apps and malware had up to 4 and 9 over-privileged permissions (e.g., P9 and P1 in Table 2), respectively. As such, the malware P1 could stealthily collect users' private data and send it to the attacker's server through a Wi-Fi network by chaining multiple over-privileged permissions.

> **Result 3**: When running in app-virtualization environments, many apps can be abused to sneakily access user data, obtain device resources, and modify phone statuses.

**Table 2: Illustrative examples.**

| Source | ID | Labels | Permissions | App / Library |
|---|---|---|---|---|
| **Mal** | P1 | rND | READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS, READ_PHONE_NUMBERS, CALL_PHONE, USE_SIP | Self |
| | | NR | ACCESS_COARSE_LOCATION, READ_PHONE_STATE | Analytics |
| | | rNDNR | READ_SMS | Anti-Fraud |
| | P2 | iND | ACCESS_LOCATION_EXTRA_COMMANDS | Utility |
| | P3 | rNDNR | ACCESS_COARSE_LOCATION | Notification |
| | P4 | iND | CHANGE_WIFI_STATE | Utility |
| **WJ** | P5 | rND | ACCESS_FINE_LOCATION | Utility |
| | P6 | NR | CAMERA | Utility |
| | P7 | rNDNR | READ_PHONE_STATE | Advertising (Multiple) |
| **360** | P8 | NR | READ_PHONE_STATE | Self, Advertising, Notification, Utility, WebView |
| **HM** | P9 | iND | ACCESS_LOCATION_EXTRA_COMMANDS | Self |
| | | iND | CHANGE_WIFI_STATE | Unknown (Obfuscated) |
| | | NR | ACCESS_COARSE_LOCATION | Unknown (Obfuscated) |
| | | rNDNR | READ_PHONE_STATE | Self, Utility, Programming |
| **GP** | P10 | rND | READ_PHONE_STATE | Self |
| | H1 | NR | ACCESS_FINE_LOCATION | Advertising |

## 3.3 Illustrative Examples

During our examination of apps and malware, we observed some malware samples exhibit additional malicious behaviors when executed on host apps, and many apps' over-privileged permissions were associated with third-party libraries as shown in Figure 6.

These observations raise an interesting follow-up question: **RQ4**: What are the practical consequences and root causes of over-privileged permissions?

**Samples for in-depth analysis.** We conducted an in-depth analysis of several apps and malware to answer this question. Besides those that stimulated our observations, we chose some cases for each label in our taxonomy, summarized in Table 2. `P1` to `P4` are malware samples detected by at least ten anti-virus engines on the VirusTotal website, which also abuse over-privileged permissions. `P5` to `P10` are popular plugin apps that are representative examples of over-privilege for various permissions. `H1` is a popular host app with over one hundred million downloads.

**Practical consequences.** To better understand the privacy consequences and chained effects in practice, let us consider a scenario where `P1`, `P3`, `P5`, and `P9` are plugin apps inside the host app `H1`. Consider a frequently used permission, `location` (`ACCESS_COARSE/FINE_LOCATION`). `P5` does not declare it, `H1` , `P1`, and `P9` do not request it, and `P3` neither declares nor requests it. However, when `P5` runs in the host app, it can successfully request the permission due to the host app's overclaiming permissions. Afterward, both the malware (`P1` and `P3`), plugin app (`P9`), and the host app (`H1`) can obtain the device location without requesting user consent. Consider another frequently used permission, `Telephone` (`READ_PHONE_STATE`). `H1` requests this permission before loading any plugin app. This allows plugin apps `P1`, `P7`, `P8`, and `P9` to access phone statuses without requesting user consent.

**Root causes.** By examining selected app and malware samples, we identified three potential root causes for the over-privilege problem. First, malware may intentionally seek to amplify its malicious activities. Second, in benign apps, the mixed codebase can inadvertently introduce over-privilege, even if not purposefully added by developers. Last, developers may overlook changes in permission declarations and requests during app version updates. Below, we delve into each of these causes.

*3.3.1 Enhancing malicious behaviors.* We found malware samples that appear to amplify malicious behaviors by exploiting over-privilege during app virtualization, as evidenced by multiple permissions labeled as `ND` or `NR`. One prominent example is `P1`, exhibiting 9 over-privileged *runtime* permissions, the highest among the malware samples examined. This malware is categorized as a Trojan on the VirusTotal website, performing malicious activities stealthily. Specifically, it had 6 permissions labeled as `rND` related to telephony features. After being granted these over-privileged permissions, it could read and write call logs and monitor outgoing calls in real time. This malware also utilized analytics libraries to track victims. Another malware, `P2`, was classified as spyware Play Protect. When `P2` was executed in the host apps, it could escalate `ACCESS_LOCATION_EXTRA_COMMANDS` and call `sendExtraCommand` to inject GPS status into the location system. As these align with its known malicious purpose of spying on personal data, `P2` might intentionally exploit the over-privilege problem.

*3.3.2 Third-party libraries.* As briefly explained in RQ2, libraries can have a broader over-privileged permission scope during app virtualization compared to native environments. Without app virtualization, over-privileged libraries could only exploit permissions on the premise that apps have. In comparison, during app virtualization, *libraries in a plugin app* could gain more illegal authorizations *from the host app* regardless of the permission statuses of that plugin app. For example, we found `P6` had over-privileged `CAMERA` permission, which its video-capturing library utilizes. The illegal authorization of this permission is privacy-invasive because the library could initiate an instance of the camera directly calling `Camera.open` API. `P7` is another example representing the over-privilege caused by multiple advertising libraries. When `P7` executes in the host app, the over-privileged `READ_PHONE_STATE` lets these libraries successfully read phone statuses that they were originally not allowed. In addition to plugin apps, we also found the advertising library used by the host app `H1` can obtain user location unauthorizedly. After a plugin app has requested `ACCESS_FINE_LOCATION`, the library in `H1` can successfully obtain a device's precise GPS location. This is problematic because users are unaware that the location permission granted to a plugin app is *also* being abused by the host app.

It is important to highlight that even though app developers can refrain from declaring or requesting unnecessary permissions for libraries, this precaution becomes ineffective during app virtualization. This situation is frequently observed in ad frameworks, analytic libraries, and trackers.

*3.3.3 App development and update issues.* `P9` has 4 over-privileged permissions, spanning across install-time and runtime. Apart from the third-party libraries mentioned previously, we found that the app itself also caused over-privilege problems. Since this app is well-known for providing travel services, we do not regard it as with malicious intentions. Instead, it may be due to developers' negligence about required permissions. We also found improper app version management that may cause `Plugin_rND`. When updates took place, the developers presumably removed both the permission-using code and the permission declaration but neglected to remove the permission-requesting code. For example, the current version of `P10` (6.1.15) does not declare `READ_PHONE_STATE`, whereas its older versions (e.g., 3.5.96) declare it in the manifest file. As such, `P10` (6.1.15) could trigger the permission dialogues for its undeclared runtime permissions in all host apps.

**Result 4**: Malware can enhance malicious behaviors by abusing over-privileged permissions, whereas apps' third-party libraries or app development issues make apps become over-privileged inadvertently.

## 4 DEFENSE: PERMSEP

Our investigation uncovered numerous real-world instances of apps and malware that can abuse over-privilege issues on commodity app-virtualization platforms.

Building upon these findings, we develop PermSep, a practical solution aimed at preventing *all* these over-privilege types. The core concept behind PermSep involves separating the permissions of host and plugin apps, enabling each to adhere to the Android permission model. By leveraging common practices among commodity host apps, we ensure that this separation can be achieved while preserving app functionalities and user convenience.

### 4.1 Desired Properties

In this section, we define the desired properties (DPs) in addition to preventing over-privilege. We discuss in Section 4.5 that PermSep fulfills the properties. To the best of our knowledge, none of the prior work can achieve all four DPs simultaneously.

**DP1. Preserving app functionality.** While our defense solution blocks over-privilege, we expect it to have no negative impact on app functionality and other legal permission operations. Thus, PermSep should be able to *distinguish* whether permission is legal rather than adopting an all-or-nothing strategy.

**DP2. Accurate permission context.** As stated by the Android permission model, permissions usage should align with the context users have consented to. Users should make decisions based on accurate context, i.e., correct names of requesting apps via the permission dialog. Thus, our defense solution should be capable of identifying permission requesters during runtime.

**DP3. Modification-free for host/plugin apps.** Off-the-shelf host apps have attracted large numbers of users. Accordingly, the defense should be compatible with existing host and plugin apps.

**DP4. Low overhead.** The performance overhead incurred by the defense solution should be low to promote user adoption.

### 4.2 Key Ideas

We focus on separating the permissions of host and plugin apps, such that they declare and request permissions on their own behalf. We can then leverage Android's existing permission-checking mechanisms to address over-privilege. Based on this this, we can preserve app functionalities and legal permission operations.

To this end, we propose PermSep, a defense system that enforces the permission model separately to the host and plugin apps. Making this happen required us to 1) identify which apps called permission functions and thereby 2) enforce the Android permission model toward the correct identity. We leverage two key observations that motivated PermSep's design.

**Identifying plugin apps through PIDs.** First, based on the observation that each plugin app runs in different processes, we utilize their unique PIDs to identify which apps called permission functions. A challenge is that a plugin app could execute in multiple processes and its PID may change dynamically. Thus, we need to ensure the process information is recorded every time a plugin app's process is created.

**Aligning permissions with native apps.** Second, based on the observation that a plugin app is cloned from a *native app*, we use the package name to align permissions between them so that we can utilize the existing Android permission model. Compared with PluginPermCheck that re-invents permission control [13], this design choice brings advantages for permission management and user convenience. Further, unlike PluginPermCheck that only checks static permission declaration, we address dynamic permission requests, enforcing permission for both install-time and *runtime* permissions. Besides, compared with creating new identities to maintain each plugin app's permission from the initial state, we avoid users' burden of repeatedly granting permissions for the same apps, either the native apps or the multiple-cloned plugin apps.

Repackaging attacks [15, 16, 21] is an evasion method to hide malicious behaviors by repackaging legitimate apps during app virtualization. These repackaged apps have the same package name as legitimate apps but *different certificates*. Android OS prohibits repackaged apps from executing by verifying that the signatures of the repackaged apps are the same as their original apps. However, some host apps allow users to upload APK files without any security check. PermSep addresses repackaging attacks by ensuring the plugin app's certificate is identical to that of the native app.

## 4.3 Workflow

At a high level, PermSep's operation consists of the three major stages illustrated in Figure 7. In the first two stages, PermSep *identifies* plugin apps and *collects* their distinguished identities when the host app launches and instantiates plugin apps. These are preparation stages that form the foundation for permission management before plugin apps start running. Then in the third stage, while plugin apps execute, PermSep *manages permissions* for requesting and checking according to the previously-recorded information. It distinguishes the current targets (the host app or plugin apps) during app virtualization and uses their identities in the permission functions. By ensuring that the Android permission model consumes different identities of the host app and plugin app, PermSep separates the permissions accordingly.

**Stage 1: Identify Plugin Apps**. PermSep first identifies which plugin apps the host app launches at runtime. Inspired by VAHunt [26], PermSep identifies essential steps taken by host apps to launch plugin apps to retrieve plugin apps' packages. It includes two sequential steps: 1) extracting the saved components from the requests launched by plugin apps, and 2) saving each one in another request that is wrapped with a "declared" component of the host app. PermSep identifies the two steps performed at runtime.

**Stage 2: Collect Distinguished Identities**. With the key observation that each plugin app runs in different processes, PermSep can

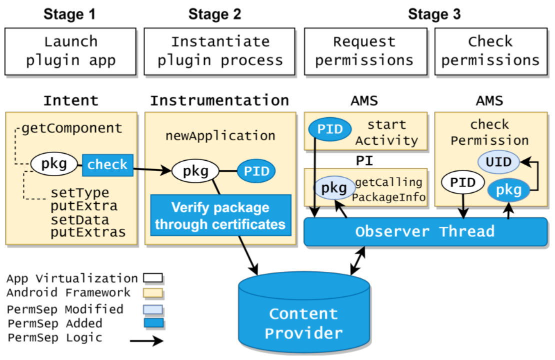


**Figure 7: A prototype of PermSep.**

distinguish plugin apps through PIDs for permission management. As such, we need to associate plugin apps' package names with their PIDs during *process instantiation*. We utilize this operation because it is performed every time a plugin app's process is created. Besides, to avoid repackaged attacks, we verify the certificates before saving these data into the PID-lookup table.

**Stage 3: Manage Permissions**. PermSep aims to manage both install-time and runtime permissions, including the whole life cycle from granting to checking. For install-time permission, the granting is statically determined upon app installation, PermSep only needs to mediate permission-checking. For runtime permission, the granting is dynamically determined through permission-requesting, PermSep should interpose both requesting and checking. The core concept is to force permission-requesting and permission-checking to use a distinguished identity of the plugin app instead of the UID of the host app.

## 4.4 Implementation

We implemented a prototype of PermSep using hooking methods with the *LSPosed* tool [18] as shown in Figure 7. *LSPosed* is a framework that enables dynamic code instrumentation of intended Android processes. The first two stages monitored Framework APIs that were used by host apps to load and instantiate plugin apps: Stage 1 used the APIs of `Intent` to identify plugin apps' packages; Stage 2 obtained the plugin app's PID through `Instrumentation` and extracted its code path from `LoadedApk` to verify its certificate. To support permission management, we hooked permission-checking interfaces of the AMS and interposed both the permission-dialog activity in PI to support permission requests. Because Android forbids user apps from directly making system calls to the kernel, user apps cannot bypass our permission-checking by using native code to make system calls to access system resources such as device location.

We used `Content Provider` to implement the PermSep's PID-lookup table that saves the hooking data from its first two stages of operation that are then used in the third stage. We also improved the efficiency of permission management by creating a new thread at the AMS to register an instance of `Content Observer` for the PID-lookup table. As such, the permission manager can be notified of changes and fetch new records into PermSep's memory in advance. We discuss the stages shown in Figure 7 below.

*4.4.1 Stage 1: Identify Plugin Apps.* Intent-wrapping is essential to the host app's launching of plugin apps. PermSep checks whether a component goes through the intent-extracting (`getComponent`) and the intent-saving (`setType`, `setData`, `putExtra`, and `putExtras`) operations in the expected order. If it does, PermSep obtains the plugin apps' package names through the wrapped component. VAHunt [26] pointed out that a small number of their test cases have multiple-intent operations that may produce false positives to such intent-wrapping patterns (i.e., intent deep copy and shortcut creation/deletion). PermSep excludes these cases at Stage 2 because they are not instantiated by the host app in new processes.

*4.4.2 Stage 2: Collect Distinguished Identities.* Having obtained the package names of the plugin apps in Stage 1, PermSep can use this information to associate with the plugin apps' PIDs at process instantiation.

**Recording plugin apps' PIDs.** PermSep monitors process instantiation through `newApplication`, which is a necessary step for every newly-created process. Because we only require plugin apps' PIDs, if PermSep only checks the package name of the `Application` object, the native app's processes will also be identified. As such, PermSep further checks the loaded package to distinguish the plugin from the native app's process. In app-virtualization scenarios, the host app acts as a placeholder to help plugin apps create processes, so the loaded package belongs to the host app. In native scenarios, on the other hand, the loaded package is identical to the package name of `Application` object.

**Verifying packages through certificates.** Repackaged apps are prohibited from executing on the Android OS. However, these apps can run successfully if users are deceived into loading repackaged APKs as plugin apps. To address this risk, PermSep ensures the consistency of the certificate between the plugin app and its native app. Therefore, before associating a plugin app's package name with PID, PermSep gets the plugin app's certificate from the APK file, and the native app's certificate from the Android OS. Specifically, it obtains the plugin app's file path by calling the `getPackageCodePath` on the instantiated application object and obtains the plugin app's certificate by calling `getPackageArchiveInfo` with the file path. The certificate of the native app, on the other hand, is retrieved from the Android OS by calling `getPackageInfo` with the plugin app's package name. If the two certificates do not match, PermSep recognizes that the app has been repackaged. It then blocks the app from being executed in the host app.

*4.4.3 Stage 3: Manage Permissions.* Then, in the third stage, permission requesting and checking can retrieve data from memory with lower latency. We discuss distinguishing the plugin apps from the host app for permission requesting and checking in the following.

**Permission-requesting.** PermSep identifies the source of permission requests and grants the permission to its requester. Here, the requester can be the plugin app or host app. We use PID as the identifier for plugin apps to convert the shared host app's package name to that of the requesting app. As the sharing is avoided, host apps can use their own package name as a distinguishable identifier. PermSep modifies two system components, i.e., AMS and PI. A schematic of this permission-requesting process is presented in Figure 6, in which the steps represent our modification.

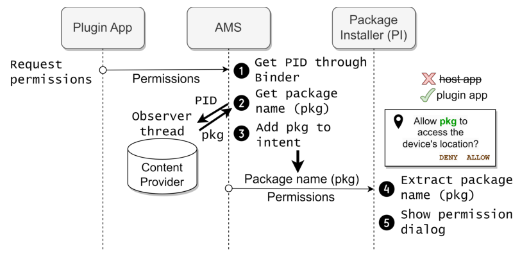


**Figure 8: Schematic of the permission-request process.**

**Permission-checking**. PermSep aims to prevent over-privilege and permit legal permission usage. It monitors `checkPermission` on the side of AMS and ensures that Android uses the distinguished identities but not the host apps' shared UID for permission **enforcement** and **self-checking**. Similar to the PID-package conversion at permission-requesting, the key to permission-checking here is to convert the shared UID to the plugin app's corresponding UID. PermSep obtains the package name from the observer thread and uses it to acquire the native app's UID. By doing so, we align the plugin apps' permissions successfully with the native apps, preventing over-privilege. However, we found that some host apps intercepted the plugin app's self-checking calls and launched new calls to the Android system on behalf of plugin apps, which is implemented with the IPC type ❷ shown in Figure 1. In such scenarios, PermSep cannot get the plugin app's PID through Binder, which always shows the host app's PID. To overcome this, we detect the host app's *intercept-and-relaunch* permission self-checking calls and use the plugin apps' identity. In a typical self-checking scenario, the conducting app is both the caller and the checking target. In *intercept-and-relaunch* scenarios, these two are different apps: the plugin app is the caller, while the host app is the checking target on behalf of the plugin apps.

## 4.5 Evaluation

We evaluate the PermSep prototype's 1) ability to prevent over-privilege and 2) embodiment of our desired properties. This evaluation was conducted on an ASUS Zenfone M2 (X01AD) running Android 9.0, with 8-core CPUs and 4GB RAM. To evaluate the ability to prevent over-privilege, we tested 17 host apps and plugin apps from Table 2. To evaluate the embodiment of our desired properties, we also crafted a plugin app — *PluginBench*, which is used to test permission-requesting and common Android API calls automatically. We manually executed the plugin apps in different host apps to conduct the evaluation and used the following methods for observation. 1) *permission context*: manually inspecting the app and permission names showing on the permission-request dialog; 2) *permission-granting statuses*: `dumpsys` tool filtering the permission of the observed package within the `adb` shell; 3) the outcomes of *permission-enforcement* and *permission-required operations*: the output log of PermLabel's monitoring modules.

**Preventing Over-privilege.** After deploying PermSep in real app virtualization environments, we evaluated if PermSep effectively separates the permissions of host apps from plugin apps, thus preventing over-privilege.

We ran the plugin apps in each host app on a PermSep-enabled system. After granting the host and plugin app's permission requests, we checked both their permission-granting statuses and found that PermSep successfully separated the list of permissions granted to host apps from plugin apps in all cases. We reproduced the over-privilege found by PermLabel and observed permission-granting statuses and the outcomes of permission-required operations. We manually executed the apps on a PermSep-enabled system along with PermLabel's monitoring modules. We checked the outcomes of plugin apps' permission-required operations concerning install-time permissions. For runtime permissions, after granting permissions upon the host app's requests, we checked both the permission-granting statuses and permission-required operations.

We tested PermSep against the over-privileged samples found by PermLabel in Section 3.2. PermSep blocked the over-privilege of both install-time and runtime permissions as the corresponding permission-required operations failed. Specifically, `H1` could no longer access the device's GPS location (`NR`), `P1` and `P10` could no longer request the undeclared permissions (`rND`), `P2` could no longer use the location provider's extension (`iND`), `P3` could no longer obtain the device's cell location (`rNDNR`), `P4` could no longer start a Wi-Fi scan (`iND`).

**Summary.** We discuss that PermSep achieved all four desired properties. Table 3 summarizes the comparison with related work, showing PermSep's advantage in addressing over-privilege.

**DP1 and DP2:** We validated DP1 for the host and plugin apps by demonstrating that PermSep did not affect permission-requesting and API calls. We utilized *PluginBench* to test permission-required API calls identified by PermLabel. This plugin app requests permissions and then triggers each API sequentially. We found that the host and plugin app could use legal permissions and successfully call permission-required APIs. In contrast, previous work that aborted normal execution when the app virtualization environment was detected [1, 9, 19, 25, 27]. In addition, we validated that PermSep achieves DP2. We confirmed that the app and permission name shown on the permission dialog are the same as the ones shown on the native app.

**DP3:** The design of PermSep does not involve the modification of either host apps or plugin apps. Thus, we validated DP3 by checking whether these apps could function normally while it was deployed. We tested plugin apps in Table 2 in host apps and found that all functioned normally without encountering unexpected crashes while preventing over-privilege, showing that PermSep can seamlessly integrate with the current app virtualization framework.

**DP4:** To ensure that PermSep's overhead would be acceptable to users, we ran *PluginBench* on Dual Space [10] to test a set of microbenchmarks that evaluate the latency of operations involved in each of PermSep's three stages and Android API calls. We tested *PluginBench* with and without PermSep and calculated the latency incurred by PermSep. The results presented in Figure 9 are the averaged latency of 230 runs. The latency PermSep introduced in Stage 1 is negligible. The latency in Stage 2 is relatively higher than the other stages because more complex operations are involved for logging identities and verifying the certificate of plugin apps. Note that this latency occurs only during the process instantiation of plugin apps, which is a one-time cost before plugin apps execute.

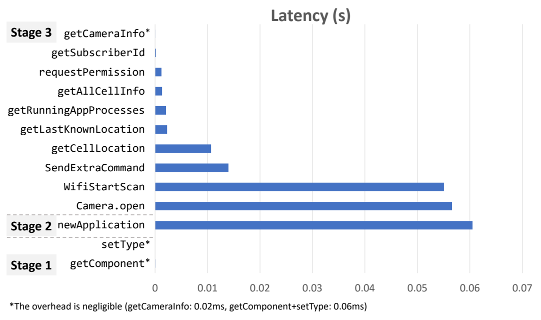


**Figure 9: Microbenchmarks.**

**Table 3: Comparing PermSep with related work.**

| Defense Method | Preventing Over-privilege | DP1 | DP2 | DP3 | DP4 |
|---|---|---|---|---|---|
| Default host apps | ✗ | ✓ | ✗ | ✓ | ✓ |
| VAHunt [26] | ✗ | ✓ | ✗ | ✓ | –* |
| NJAS [7] | ▲ | ▲* | ✗ | ✗ | ✓ |
| Boxify [6] | ▲ | ▲* | ✗ | ✓ | ✓ |
| Fingerprinting [1, 9, 19, 25, 27] | – | ✗ | ✗ | ✗ | ✓ |
| PluginPermCheck [13] | ▲ | ✓ | ✗ | ✓ | ✓ |
| PermSep (**This work**) | ✓ | ✓ | ✓ | ✓ | ✓ |

▲: Partially achieved, i.e. `rNDNR` and `NR` unsupported.
▲*: NJAS needs to re-generate the host app for different plugin apps; Boxify adopts the fail-safe defaults and thus limits the functionalities of plugin apps.
–: Fingerprinting aborts normal execution when the app virtualization environment is detected.
–*: VAHunt uses a static detection method that does not affect apps' execution at runtime.

Stage 3 in operations that include multi-staged IPC operations or initializing hardware, like opening the camera or scanning Wi-Fi.

## 5 DISCUSSION

**Integrating PermSep into the Android Platform.** Although we currently prototyped PermSep on Android 9, the core design mechanisms, such as identifying plugin apps through PIDs, do not depend on version-specific features. We believe PermSep can be ported to newer Android releases with modest efforts.

**Limitations of PermLabel.** PermLabel inherits two limitations of dynamic testing: (1) testing time and (2) program coverage issues. Based on the methodology, the average activity coverage for tested apps is about 10%. The set of permissions that PermLabel discovered were bounded by program execution paths recorded within the testing time frame and capability of the event generator. We chose DroidBot because it is lightweight and does not require system modification or app instrumentation. To improve the coverage, we developed a custom script to automate several predefined operations, e.g., approving normal permission dialogs. The limitations can be potentially addressed by leveraging an optimized event generator that provides better test coverage more efficiently.

**Limitations of PermSep.** PermSep aligns the plugin app's identity to its native app counterpart to avoid reinventing a permission model. It requires a given plugin app to co-exist with a respective native app and both apps share the same set of permissions. The first limitation has little impact in practice as users commonly execute cloned copies of native apps in the host apps as plugins. The second limitation can be addressed by assigning a new UID to plugin apps that separates the apps' permissions. This is an area of future work.

## 6 RELATED WORK

*Measurement of Over-privilege.* Previous work [9, 31, 25] only presented that the UID sharing of app virtualization leads to over-privilege problems with proof-of-concept examples but did not discuss the different types, causes, and consequences of over-privilege. In contrast, our work considers the over-privilege of install-time and runtime permissions. It provides a comprehensive classification of over-privilege, including undeclared and unrequested permissions. Moreover, our work thoroughly study real-world apps that may exploit over-privilege.

*Defense against Over-privilege.* At the time of writing, the only work focusing on preventing over-privilege is PluginPermCheck [13]. It aims to prevent the over-privilege by removing permissions undeclared from the plugin apps' manifest file. PermSep addresses the PluginPermCheck's limitations as follows. First, beside preventing over-privilege resulting from "undeclared permissions", PermSep additionally prevents "unrequested permissions". Second, PermSep supports runtime permission checks to provide accurate context for users during runtime.

*Defense against Other App-virtualization Attacks.* One line of research focused on protecting benign apps from being repackaged and executed in a host app for malicious purposes. These work [1, 9, 19, 25, 27] proposed approaches for plugin apps to detect if they execute in an app virtualization environment and stop the execution if true. Another line of research proposed defense using malicious indicators. For instance, Zhang et al. [31] identified malware by certificate differences between the host app and plugin apps. However, the approach could erroneously identify benign apps as malware. These solutions are inapplicable to addressing over-privilege.

## 7 CONCLUSION

This paper presented a comprehensive measurement study and defense solution for over-privilege during app virtualization. We have designed PermLabel, a novel tool to identify over-privileged permissions in apps and classify them with the defined taxonomy. We found that known apps running in commodity host apps exhibited over-privilege problems. Thus, we proposed PermSep to prevent discovered threats in the measurement study while preserving the normal execution of app virtualization. We evaluated PermSep against 17 commonly used host apps and showed that it effectively prevents over-privilege while preserving performance.

## ACKNOWLEDGEMENT

This research was supported in part by the National Science and Technology Council of Taiwan under grants NSTC 112-2223-E-002-010-MY4, NSTC 112-2634-F-002-002-MBK, and National Taiwan University under grant NTU-113L7871.

## REFERENCES

[1] Marco Alecci, Riccardo Cestaro, Mauro Conti, Ketan Kanishka, and Eleonora Losiouk. "Mascara: A Novel Attack Leveraging Android Virtualization". In: *arXiv preprint arXiv:2010.10639* (2020).

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "AndroZoo: Collecting Millions of Android Apps for the Research Community". In: *International Conference on Mining Software Repositories.* MSR '16. Austin, Texas: ACM, 2016, pp. 468–471.

[3] *Android Developers.* 2023. url: https://developer.android.com/reference/.

[4] Anshul Arora, Sateesh K Peddoju, and Mauro Conti. "Permpair: Android malware detection using permission pairs". In: *IEEE Transactions on Information Forensics and Security* 15 (2019), pp. 1968–1982.

[5] asLody. *VirtualApp.* 2022. url: https://github.com/asLody/VirtualApp.

[6] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. "Boxify: Full-fledged app sandboxing for stock android". In: *24th USENIX Security Symposium (USENIX Security 15).* 2015, pp. 691–706.

[7] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. "Njas: Sandboxing unmodified applications in non-rooted devices running stock android". In: *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices.* 2015, pp. 27–38.

[8] David Curry. *Android Statistics (2022).* url: https://www.businessofapps.com/data/android-statistics/.

[9] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin. "Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android". In: *ACM Symposium on Access Control Models and Technologies.* 2020, pp. 25–32.

[10] *Dual Space.* url: http://www.dualspace.com/.

[11] Google. *Declare permissions for your app.* url: https://support.google.com/googleplay/android-developer/answer/9214102.

[12] Google. *Permissions and APIs that Access Sensitive Information.* url: https://support.google.com/googleplay/android-developer/answer/9888170.

[13] Shou-Ching Hsiao and Hsu-Chun Hsiao. "POSTER: PLUGINPERMCHECK: Preventing Permission Escalation in App Virtualization". In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy.* 2022.

[14] Boya Li, Guojun Wang, Haroon Elahi, and Guihua Duan. "A light-weight framework for pre-submission vetting of android applications in app stores". In: *International Conference on Dependability in Sensor, Cloud, and Big Data Systems and Applications.* Springer. 2019, pp. 356–368.

[15] Li Li, Tegawendé F Bissyandé, and Jacques Klein. "Rebooting research on detecting repackaged android apps: Literature review and benchmark". In: *IEEE Transactions on Software Engineering* 47.4 (2019), pp. 676–693.

[16] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. "Understanding android app piggybacking: A systematic study of malicious code grafting". In: *IEEE Transactions on Information Forensics and Security* 12.6 (2017), pp. 1269–1284.

[17] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. "Droidbot: a light-weight ui-guided test input generator for android". In: *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C).* IEEE. 2017, pp. 23–26.

[18] LSPosed. *LSPosed.* url: https://github.com/LSPosed/LSPosed.

[19] Tongbo Luo, Cong Zheng, Zhi Xu, and Xin Ouyang. "Anti-plugin: Don't let your app play as an Android plugin". In: *Proceedings of Blackhat Asia* (2017).

[20] *Maven Repository.* 2023. url: https://mvnrepository.com/.

[21] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. "You shall not repackage! demystifying anti-repackaging on android". In: *Computers & Security* 103 (2021), p. 102181.

[22] *Parallel Space.* url: http://parallel-app.com/.

[23] Qihoo360. *DroidPlugin.* 2022. url: https://github.com/DroidPluginTeam/DroidPlugin.

[24] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. "FLEXDROID: Enforcing In-App Privilege Separation in Android." In: *NDSS.* 2016.

[25] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps". In: *Annual International Conference on Mobile Systems, Applications, and Services.* 2019, pp. 222–235.

[26] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. "VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 2020, pp. 535–549.

[27] Yifang Wu, Jianjun Huang, Bin Liang, and Wenchang Shi. "Do not jail my app: Detecting the Android plugin environments by time lag contradiction". In: *Journal of Computer Security* 28.2 (2020), pp. 269–293.

[28] Wei Xu, Fangfang Zhang, and Sencun Zhu. "Permlyzer: Analyzing permission usage in android applications". In: *IEEE International Symposium on Software Reliability Engineering.* IEEE. 2013, pp. 400–410.

[29] Win Zaw Zarni Aung. "Permission-based android malware detection". In: *International Journal of Scientific & Technology Research* 2.3 (2013), pp. 228–234.

[30] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. "Research on third-party libraries in android apps: A taxonomy and systematic literature review". In: *IEEE Transactions on Software Engineering* (2021).

[31] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. "App in the middle: Demystify application virtualization in Android and its security threats". In: *ACM on Measurement and Analysis of Computing Systems* 3.1 (2019), pp. 1–24.

[32] Xiaonan Zhu, Jinku Li, Yajin Zhou, and Jianfeng Ma. "AdCapsule: Practical confinement of advertisements in android applications". In: *IEEE Transactions on Dependable and Secure Computing* 17.3 (2018), pp. 479–492.