Don't Mess with Bro's Cheese! An Empirical Study of Resource Conflict in Android Multi-window

Chenkai Guo*†¶, Huimin Zhao*, Tianhong Wang‡, Naipeng Dong§, Qingqing Dong*, Jiarui Che‡,
Yaqiong Qiao*, Xiangyang Luo¶, and Zheli Liu*
*College of Cyber Science, Nankai University, China
†Haihe Lab of ITAI, China
‡College of Computer Science, Nankai University, China
§The University of Queensland, Australia
¶Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, China
‖State Key Laboratory of Mathematical Engineering and Advanced Computing, China
{guochenkai, liuzheli}@nankai.edu.cn, {huimilia, tianhongwang, 2120240716}@mail.nankai.edu.cn,
n.dong@uq.edu.au, qingqingdong0717@gmail.com, kitesmile@126.com, luoxy_ieu@sina.com

***Abstract*—The multi-window mode in Android has greatly improved productivity and usability by allowing multiple apps to run concurrently. However, alongside the advantages, such mode also introduces unforeseen risks in both functionality and security. In this work, we present the first systematic study to identify a previously unexplored class of issues, termed Multi-window Resource Conflicts (MRCs). Such conflicts occur when multiple app windows access the same system resource concurrently, potentially leading to crashes, functionality failures or unintended behaviors. To enhance the robustness and security of Android multi-window execution, we conduct a systematic and in-depth empirical study on the MRCs. We begin with a comprehensive root cause analysis, categorizing MRCs into three fundamental types based on their triggering patterns and affected resource states. To enable large-scale detection, we develop MRC-Detector, a static analysis framework that automatically identifies MRC issues in Android apps. Our manual verification confirms its high accuracy and effectiveness. We apply the MRC-Detector to the detection of over $150k$ real-world apps from `F-droid` and `Google Play`, uncovering the prevalence of MRC risks. Additionally, the distribution of MRC issues is analyzed in depth across multiple dimensions, including MRC type, APK size, app source and security classification. We further investigated the recognition and confirmation from developers and received 14 positive responses from vendors and project maintainers. Finally, comprehensive mitigation strategies are discussed. The materials of the study are available at: https://github.com/Huimilia/MRC.**

***Index Terms*—resource conflict, Android applications, multi-window, static detection.**

I. Introduction

Android applications (apps for short) have increasingly become the primary platform for work, study and entertainment, due to their convenience, openness and diversity. To meet the requirements of fast-paced lifestyles and the growing need for multitasking, Android has evolved beyond the traditional single-window execution mode to support various multi-window operation modes. Introduced in Android 7.0 and enhanced in later versions, the multi-window mode allows users to run multiple apps simultaneously on a single screen. In particular, Android 10.0 introduced the *multi-resume* mechanism, enabling `Activities` from different windows run concurrently at the `onResume` state, changing the way `Activity` lifecycles are managed. This mode-level innovation in both UI rendering and functional execution significantly enhances the usage experience in multitasking scenarios and opens avenues for new interface design and feature implementation.

In practice, the implementation of app functionalities often relies on access to underlying system resources. For example, scanning a `QR code` requires *camera* access, while map services depend on *location* data. In multi-window mode, multiple apps can potentially access the same resource concurrently at runtime. This raises the risk of resource access conflicts, which may lead to crashes, functional failures or undefined behavior—issues we refer to as *Multi-window Resource Conflicts* (*MRCs*) in this work. However, neither the Android system nor third-party app developers have shown sufficient awareness of such MRC issues. The Android developer documentation [1] does not enforce strict constraints or security protections for resource access under multi-window scenarios. It merely suggests that developers should "*implement appropriate resource release and reacquisition logic when dealing with focus or lifecycle changes in the multi-window*". In practice, this suggestion is often overlooked. Our investigation of 235 real-world apps reveals that most developers fail to handle the resource reacquisition or release in response to window focus or lifecycle changes, resulting in the prevalence of MRC issues (see §V-B).

Existing research has not systematically investigated the MRC issues. Prior studies on abnormal resource usage in Android have primarily focused on resource-permission relationships [2]–[4], overlooking the concurrent execution and resource access behaviors introduced by multi-window modes. Although some works have explored multi-window modes in mobile apps, their focus has primarily been on novel attack

Corresponding author: Yaqiong Qiao.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00178

vectors, such as UI deception [5], [6] and privilege escalation [7], [8]. These efforts fail to provide a systematic analysis from the perspective of resource access conflicts, leaving the MRC issue largely unexplored.

Conducting a comprehensive analysis of MRCs presents two primary challenges in practice. ❶ Confused root causes and trigger patterns: MRCs arise from complex interactions among multiple concurrently running windows, accessing shared or exclusive resources. These interactions often involve intricate temporal sequences and triggering conditions, making it difficult to generalize the underlying patterns of occurrence. Such variability significantly hampers the automation of MRC detection and fix. ❷ Numerous influencing variables: The manifestation and severity of MRCs depend on a wide range of variables, including resource type, access method, multi-window configuration and event source (e.g., user-initiated vs. system-triggered). This diversity makes it challenging to fully capture all conflict scenarios and their potential consequences.

To address these challenges and enhance the robustness for Android multi-window execution, this work is the first to formally clarify the concept of MRC and to conduct a systematic empirical study for their causes, impacts and prevalence. First, by analyzing the access behaviors of both exclusive and shared resources, we identify and characterize three essential types of MRCs along with their corresponding triggering patterns. Building on these behavioral patterns, we propose and implement MRC-Detector, an automated static analysis framework based on the `Soot` framework [9] and taint tracking techniques [10]. Strict manual verification is designed to confirm the accuracy and applicability of the MRC-Detector. Next, we use MRC-Detector to detect over $150k$ real-world Android apps, conducting a large-scale empirical analysis of MRCs' prevalence with multiple dimensions. The results demonstrate that MRCs are highly prevalent in the real-world. However, their severity varies significantly across different MRC types, `APK` sizes, security classifications and app sources. Then, to investigate the recognition and confirmation from developers, we submitted MRC issues to several prominent app vendors and open-source repositories. As to now, we have received 14 positive responses, including 3 risk confirmations from vendors and 9 positive recognitions from project maintainers. Finally, we discuss and provide practical mitigation strategies for MRCs.

In summary, this work makes the following contributions:

- We are the first to identify and formally clarify the problem of resource conflicts in Android multi-window modes, along with root cause analysis and categorization of their triggering patterns.
- We develop an automated detection framework capable of accurately identifying MRC issues in large-scale real-world Android apps, and conduct an in-depth analysis of their prevalence and distribution.
- We investigate the recognition of MRC issues from developers, as evidenced by official confirmations and positive feedback. Moreover, comprehensive mitigation strategies are discussed.

II. Motivation

We use two real-world examples to intuitively illustrate the triggering process and execution effects of MRC issues. Fig. 1 illustrates a typical *MRC* that occurs when the app `Duxiaoman` and the app `TikTok` alternately access the `Camera` resource in multi-window mode. A user first launches the `Duxiaoman` in full-screen mode to utilize its `QR code` scanning, where the camera is successfully accessed (Fig. 1(a)). Subsequently, the user opens `TikTok` in floating-window mode and activates its photo-taking functionality. At this stage, the `TikTok` runs under the multi-window mode, while the camera preview interface of `Duxiaoman` is suspended (Fig. 1(b)). However, after the user closes the floating window of `TikTok`, the camera preview of `Duxiaoman` fails to resume, and the interface becomes unresponsive. When the user attempts to click the *flashlight button*, `Duxiaoman` crashes and automatically restarts, displaying a bottom-aligned message indicating a *runtime exception* (Fig. 1(c)).

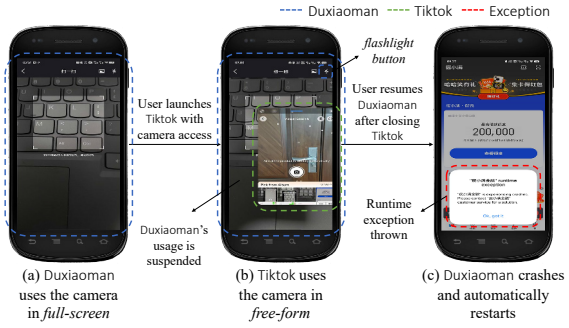

Fig. 1: Motivation 1—MRC on Camera Resource.

Fig. 2 illustrates an MRC that occurs during touchscreen recording involving the app `Shein` and `Alipay` in multi-window mode. To record the product selection process in `Shein`, a user first activates the system's touchscreen recording and successfully records `Shein`'s interface in full-screen mode (Fig. 2(a)). Subsequently, the user logs in `Alipay` in free-form mode to pay for products bought in `Shein`. Both `Shein` and `Alipay` are correctly displayed in the screen recording (Fig. 2(b)). Then, the user clicks the password input field of `Alipay` to complete the online payment, which triggers its secure keyboard. To protect sensitive information, `Alipay` deliberately obscures its interface by rendering a black screen to block the screen recording. Ideally, this black screen should only cover the area occupied by `Alipay`, while the remaining visible portion of `Shein` should remain unaffected. However, in practice, the entire screen turns black—overlapping `Shein`'s interface and occupying its screen space.

III. Background

A. Multi-Resume and Multi-window

Starting from Android 7.0, the platform introduced multi-window mode [1], which allows users to open and operate multiple app windows simultaneously on the same screen.

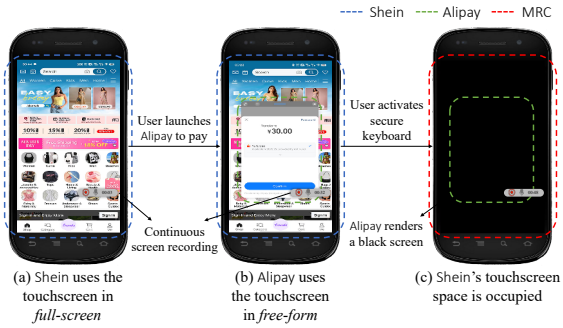

Fig. 2: Motivation 2—MRC on Touchscreen Resource.

However, early implementations (Android 7.0 to 9.0) employed a *single-resume* feature, where only the user-focused `Activity` remained in the `RESUMED` state, while other visible `Activities` are held in the `STARTED` state, resulting in a suboptimal user experience. This limitation was addressed in Android 10.0 (API level 29), which introduced the *multi-resume* mechanism, enabling all visible `Activities` on the screen to remain in the `RESUMED` state. Although this feature improved the responsiveness of user interactions in multi-window mode, it also introduced new potential defects related to resource conflicts——an issue this work investigates in depth. Based on the visual arrangement and functionality of application windows, current multi-window modes can generally be categorized into three types.

- **Picture-In-Picture (PIP)**: Predominantly used by video apps, PIP mode allows users to view videos in a small, resizable window typically anchored to a corner of the screen [11]. Notably, PIP mode does not support the `multi-resume` mechanism. When one focused app is in the `onResume()` state (focused), other apps remain in the `onPause()` and cannot access shared resource.
- **Free-Form (FF)**: FF mode allows users to display specific app interfaces as resizable, floating windows. Unlike PIP, which is primarily designed for video playback, FF mode typically retains the full native functionalities and supports the multi-resume mechanism, allowing multiple app windows to remain in the `RESUME` state simultaneously during runtime.
- **Split-Screen (SS)**: SS mode supports two app windows to be displayed side-by-side or stacked, each occupying a portion of the screen. Users can switch focus to prioritize interactions with one window. Note that within the multiple windows of SS mode, there is a designated primary window. When transitioning from SS mode to full-screen mode, only the primary window remains visible. Like FF mode, SS mode supports the multi-resume mechanism.

B. System Resources in Android

Android system resources refer to the various hardware and software capabilities provided by the operation system, including input/output devices, sensors, storage, network state and other system-level information. The system resources used by a target app are specified in the `<uses-feature>` element within the `AndroidManifest` file (the main configuration file). The features declared by an app include both *hardware* and *software* capabilities, corresponding to functionalities provided by the Android `PackageManager`. Hardware features, indicated in the `android:name` attribute with a prefix of `android.hardware`, generally refer to the physical characteristics of the device. These features utilize hardware components to provide data, services and functionalities to apps. Software features, on the other hand, are denoted by a prefix of `android.software` and represent modules or specific APIs. As OS-level abstractions over physical components, they may be contended in multi-threaded contexts but are effectively handled through conventional concurrency control mechanisms [12] [13]. By contrast, hardware features are protected in single-window mode yet become vulnerable in multi-window mode, revealing conflict patterns not captured by existing models. These patterns therefore constitute the central focus of our analysis. In total, the `<uses-feature>` element covers 18 hardware resources such as camera, audio and sensors, providing essential device functionalities like photography, audio playback or recording. These resources are recognized by the Android system as standard resources available to apps. These resources can be further classified as *exclusive resources* and *shared resources*, according to the access control strategies. *Exclusive resources* are system components that only one active app can access at a time. Attempting concurrent access to such resources typically results in denial of access or forced release by the current holder. In contrast, *Shared resources* allow simultaneous access by multiple apps or windows.

IV. Problem Definition

To ensure system security and app performance, all the app windows in multi-window mode should execute in isolation and avoid interfering with one another without explicit user intent. The root cause of the MRC lies in the *unintended triggering* of resource access by an app window without explicit user interaction. In practice, such unintended event can lead to unexpected anomalies affecting both the system and other apps. Based on this, we define MRCs using the following definitions.

A. Resource Access in Multi-window

In multi-window mode, the access of a windowed app to the hardware resources can be represented as a 5-tuple $\{\mathcal{R}, \mathcal{W}, \mathcal{E}, \rightarrow, \rightsquigarrow\}$, where $\mathcal{R} = \mathcal{R}_u \cup \mathcal{R}_s$ denotes the set of resources with $\mathcal{R}_u$ and $\mathcal{R}_s$ denoting the exclusive and shared resources, respectively; $\mathcal{W} = \mathcal{W}^n \cup \mathcal{W}^f$ denotes the set of app windows with $\mathcal{W}_n$ and $\mathcal{W}_f$ representing non-focused and focused windows, respectively; $\mathcal{E} = \mathcal{E}_u \cup \mathcal{E}_n$ denotes the set of triggering events with $\mathcal{E}_u$ and $\mathcal{E}_n$ referring to user-driven and non-user-driven events, respectively; $\xrightarrow{\mathcal{E}} = \mathcal{W}^u \times \mathcal{W}^f$ represents the state transitions of window focus acquisition triggered by event set $\mathcal{E}$; $\overset{\mathcal{E}}{\rightsquigarrow} = \mathcal{W} \times \mathcal{R}$ represents the access relationship from window set $\mathcal{W}$ to resource set $\mathcal{R}$ triggered by $\mathcal{E}$. Thus, the access from a window $w \in \mathcal{W}$ in the multi-window mode

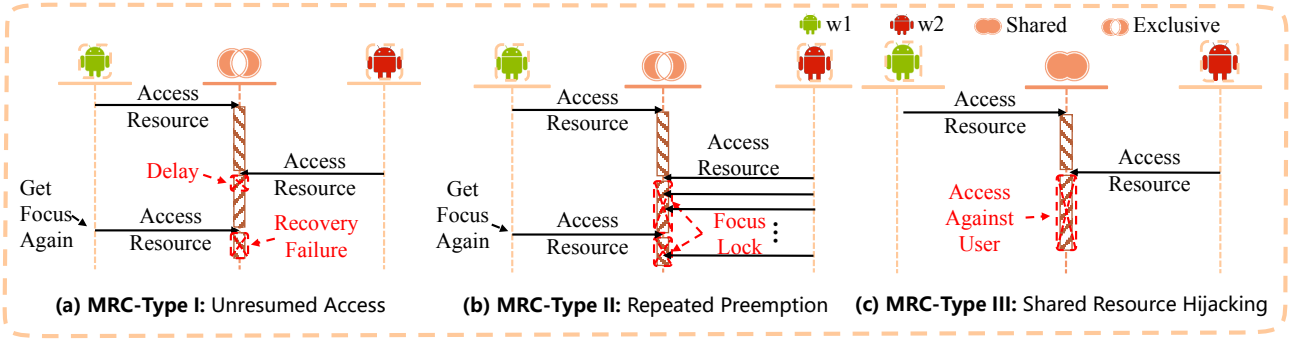

Fig. 3: Execution Sequences for Different Types of MRCs.

to a resource $r \in \mathcal{R}$ can be divided into two types based on access restrictions:

- **Exclusive Access.** This occurs when a non-focused window acquires focus and subsequently accesses an exclusive resource, which can be formally expressed as

$$a_e = w^n \xrightarrow{e_1} w^f \overset{e_2}{\rightsquigarrow} r_u, \tag{1}$$

  where $w^n \in \mathcal{W}^n$, $w^f \in \mathcal{W}^f$, $e_1, e_2 \in \mathcal{E}$ and $r_u \in \mathcal{R}_u$.
- **Shared Access.** This involves direct access to a shared resource from either a focused or non-focused window, which is formally expressed as

$$a_s = w \overset{e_3}{\rightsquigarrow} r_s, \tag{2}$$

  where $w \in \mathcal{W}$, $e_3 \in \mathcal{E}$ and $r_s \in \mathcal{R}_s$.

B. Types of MRCs

As formalized in Section IV-A, resource access in multi-window modes is jointly determined by three core factors: window focus state (e.g., $\mathcal{W}^n$ and $\mathcal{W}^f$), resource type (e.g., $\mathcal{R}_u$ and $\mathcal{R}_s$), and triggering event (e.g., $\mathcal{E}_u$ and $\mathcal{E}_n$). When the triggering events originate from non-user, i.e., $e_1, e_2, e_3 \in \mathcal{E}_n$ in Eq. (1) and (2), the access process is referred as *unintended access triggering*, raising the *root cause* of the MRCs. Under the unintended access triggering, the remaining two factors induce four possible scenarios ($\mathcal{W} \times \mathcal{R}$). We then perform execution-sequence reasoning and experimental validation across these scenarios (V-A) and, on this basis, identify three representative categories of MRCs that capture the prevalent and characteristic conflict patterns in multi-window environments. Note that these categories are not exhaustive, and subtle MRC variants may exist (see VI).

- **MRC-Type I (Unresumed Access):** This type occurs when a non-focused window attempts to obtain focus triggered by a non-user-driven event ($e_1 \in \mathcal{E}_n$), which can potentially lead to unstable or undefined behavior. A typical process of *unresumed access* is illustrated in Fig. 3(a). In multi-window mode, the system focus initially held by window w_1 is preempted by w_2. After w_2 completes its operation and exits, w_1 regains focus. However, since this focus reacquisition by w_1 is triggered by a non-user-driven event, the resource access logic in w_1 may fail to resume properly, resulting in an unresumed access conflict.
- **MRC-Type II (Repeated Preemption):** This type of MRC occurs when a window repeatedly and unintentionally acquires focus through non-user-driven event ($e_2 \in \mathcal{E}_n$), continuously preempting other active windows. As shown in Fig. 3(b), the window w_2 preempts the operation focus of w_1 through repeatedly accessing the target exclusive resource. This behavior disrupts resource access or UI interaction of other apps, degrading both usage experience and system stability.
- **MRC-Type III (Shared Hijacking):** This type of MRC arises when a non-focused or background window accesses a shared resource without the user's awareness or consent. As illustrated in Fig. 3(c), a non-user-driven event ($e_3 \in \mathcal{E}_n$) triggers window w_2 to hijack a shared resource previously accessed by w_1. This results in w_2 performing unintended operations (e.g., reading clipboard data or hijacking sensor information), potentially causing privacy leakage or functionality failure.

V. Systematic Study of MRCs

Building on the root cause of MRCs identified in the problem definition (§IV), this section presents a systematic study of MRCs from multiple analytical perspectives, guided by the following four key research questions:

- **RQ1: What types of system resources are susceptible to MRC issues?**
- **RQ2: Can MRC issues be automatically detected, and how accurate is such automated analysis?**
- **RQ3: How prevalent are MRC issues in large-scale, real-world Android apps?**
- **RQ4: How do app developers recognize and confirm MRC issues?**

A. RQ1: MRCs in Different Types of Resources

1) Resource Classification: As outlined in the Problem Definition (§IV), diverse system resources can give rise to different types of MRCs depending on their access characteristics in the multi-window mode. Therefore, accurately identifying whether a given resource exhibits *exclusive* or *shared* behavior is a necessary prerequisite for effective conflict detection.

To this end, we first conducted a manual study of all the 18 hardware resources indicated in the `<uses-feature>` element in the Android document [14], aiming to classify them based on their access characteristics (*resource classification*).

HAL Support. Our preliminary investigation revealed that some hardware resources do not directly interact with the physical hardware via the *Hardware Abstraction Layer* (`HAL`). Instead, their functionalities are simulated at the framework level through logical abstractions, without involving low-level hardware access. As these resources do not engage in direct hardware communication, they are not subject to genuine MRCs and are excluded from further analysis. Specifically, we eliminate resources that lack clearly defined `HAL` interfaces in `AOSP`, including `Device UI`, `Gamepad`, `OpenGL ES`, `Infrared`, `Screen`, and `Vulkan`. Our study thus focuses on the remaining 12 *HAL-supported* hardware resources, analyzing their behavior in practical multi-window scenarios.

Stub Structure. To empirically determine whether a resource exhibits exclusive or shared access behavior, we manually tested each resource through real multi-window execution. To ensure fairness and consistency, we developed a dedicated testing app for each hardware resource, with all test apps following a uniform *stub structure* (as shown in Fig. 4) and a standardized implementation. Each app consists of only two *activities*: `MainActivity` and `MRCActivity`. The `MainActivity` serves as the entry point of the app and connects to `MRCActivity` through a dedicated `button`. Within `MRCActivity`, we override the `onWindowFocusChanged(boolean hasFocus)` method to monitor the window's focus status. Based on the `hasFocus` value, the *activity* conditionally invokes corresponding methods to *access* (via `start()`) or *release* (via `stop()`) the target resource. Both `start()` and `stop()` methods are implemented strictly using standard Android APIs provided by the official `SDK` [15], with no custom optimizations, delay mechanisms, or reuse strategies. This uniform design ensures consistent resources access behavior across all test apps, minimizing potential bias introduced by implementation differences.

Determining by Execution. To explicitly support execution in multi-window environments, we set the `android:resizeableActivity` attribute to `true` for each test app. Subsequently, for each hardware resource, we deployed two identical instances of the corresponding test app on `Android 14`, and executed them under different multi-window mode combinations, namely `FS+FF`, `FS+PIP` and `SS+SS`. In each configuration, both instances simultaneously requested and attempted to use the same hardware resource. If both apps succeeded in utilizing the resource concurrently without conflict, the resource was determined as *shared*. Conversely, if only one instance was able to successfully access the resource, it was classified as *exclusive*. Using this strategy, we determined that 7 out of the 12 `HAL-backed` hardware resources exhibit *exclusive* behavior, while the remaining 7 demonstrate *shared* access characteristics. Notably, the `Audio` and `Telephone` resources can exhibit both exclusive and shared behaviors depending on the trigger context. The detailed classification is summarized in Table. I.

```
1 public void onWindowFocusChanged(boolean hasFocus)
2    ...
3    if(hasFocus){
4        resource.start();  // remove -> MRC Type I
5    }else{
6        resource.stop();   // remove -> MRC Type III
7    }
```

(a) Stub for Classification and MRC-Type Ⅰ/Ⅲ Detection

```
1 protected void onResume(){
2    ...
3    Runnable checkRunnable = new Runnable(){
4       public void run(){
5          ...
6          if(!resource.isRunning()){
7             resource.start();  // repeatedly invoke
8          }
```

(b) Stub for MRC-Type Ⅱ Detection

Fig. 4: Stub Structure for Resource Classification and MRC-Type Identification.

2) MRCs in Different Resources: Based on the identified resource classifications, the MRC issues triggered by hardware resources can then be systematically studied. For each resource, we constructed corresponding variant code implementation tailored to different MRC types, building upon the standarized *stub structure* described earlier.

- **MRC-Type I (Unresumed Access):** This type arises when an exclusive resource loses and later regains the window focus, but the app fails to resume resource access. This behavior corresponds to the `MRC-Type I` code block in Fig. 4, in which the implementation omits logic for reacquiring the resource upon focus recovery. For example, in multi-window mode, an app that previously lost access to the `Microphone` resource may regain system focus, but fail to restore microphone input if no explicit resource reinitialization is triggered. Consequently, the *microphone input capture cannot resume*.
- **MRC-Type II (Repeated Preemption):** This type occurs when an app repeatedly accesses an exclusive resource within its lifecycle. The `MRC-Type II` code block in Fig. 4 captures this behaviour through handler-posted `Runnable loop` that continuously invokes the resource access routine. For example, an app that persistently accesses the `NFC` resource in multi-window mode may monopolize it, thereby preventing other apps from initiating NFC-based operations, causing `NFC unavailability` in those competing apps.
- **MRC-Type III (Shared Hijacking):** This type is triggered when an app fails to release the shared resource after losing window focus, resulting in resource leakage or interference. This is captured by the `MRC-Type III` code block in Fig. 4, which omits logic for proper resource release upon focus loss. For example, when an app receives an `AUDIOFOCUS_LOSS` event, but does not pause or stop ongoing audio playback, it may continue

TABLE I: HAL Resource Types.

Attribute	Hardware Resources											
	Aud	**Blu**	**Cam**	**Fin**	**Loc**	**Mic**	**NFC**	**Sen**	**Tel**	**Tou**	**USB**	**WiF**
Resource Classification	◐	○	●	●	○	●	●	○	◐	○	●	○
Multi-window Mode	*FS+PIP*	*SS+SS*	*FS+FF*	*SS+SS*	*FS+FF*	*FS+FF*	*SS+SS*	*FS+FF*	*FS+FF*	*FS+FF*	*SS+SS*	*FS+PIP*

Abbreviations: Aud (Audio), **Blu** (Bluetooth), **Cam** (Camera), **Fin** (Fingerprint), **Loc** (Location), **Mic** (Microphone), **NFC** (NFC), **Sen** (Sensor),**Tel** (Telephony), **Tou** (Touchscreen), **USB** (USB), **WiF** (Wi-Fi). **Legend:** ● Exclusive, ○ Shared, ◐ Both Shared and Exclusive.

outputting audio concurrently with other apps, resulting in overlapping sound i.e., *audio output hijacking*.

Subsequently, each instrumented test app was paired with its original (unmodified) counterpart to form the multi-window execution combinations. In each test combination, we swapped the roles and window modes between the two apps to ensure that each app was executed once in every configuration. We then conducted manual trigger operations for each of the three MRC types: ❶ For `MRC-Type I`, the focus is alternated between the two windows to simulate sequential access to the exclusive resource. ❷ For `MRC-Type II`, the window is maintained to repeatedly access to the exclusive resource. ❸ For `MRC-Type III`, the window accessing to the shared resource is intentionally triggered to lose its focus without releasing the resource. During each test, we monitored for abnormal behaviors such as crashes, black screens, functional failures and other non-user-intended effects. If any such anomaly occurred, it was recorded as an MRC instance. If any window of the execution combinations triggered at least one MRC, we classified it as *MRC-prone resource*. Finally, we identified 7 MRC-prone resources with 5 exclusive resources and 2 shared resources, as demonstrated in the Table II.

Finding 1: Among the 12 HAL-supported hardware resources, 7 are identified as MRC-prone, including 5 exclusive resources and 2 shared resources, demonstrating the high prevalence of MRCs across different types of resources.

B. RQ2: Accuracy of Automatic Detection.

1) MRC-Detector: To enable automated detection of MRC issues, we design and implement a static analysis tool named MRC-Detector, built upon the `Soot` framework [9]. Given an Android `APK` as input, MRC-Detector automatically analyzes the *code structure* and *call graph* to identify potential MRC issues. Fig. 5 illustrates the overall workflow of MRC-Detector, which is composed of three main stages:

Preprocessing. MRC-Detector first decompiles the target `APK` and convert the bytecode into its intermediate representation by `Soot`. After that, the permission declarations and `<uses-feature>` tags from the `AndroidManifest` file are extracted by a common-used app parser `APKParser` [16], where the hardware resources utilized by the app can be identified. The identified resources are then fed into our developed *resource analyzer* for two parts of configuration extraction based on the resource's classification: ❶ Multi-window focus-awareness callbacks (e.g., `onWindowFocusChanged` for focus monitoring); ❷ Corresponding resource management APIs (e.g., `android.media.AudioRecord.startRecording()` for audio recording).

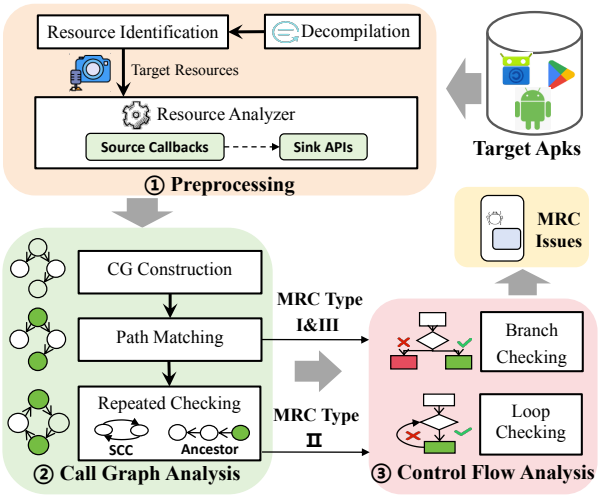

Fig. 5: Workflow of MRC-Detector.

Call Graph Analysis. In the Android multi-window execution environment, MRCs originate from the mismatch between the Android lifecycle and app resource management, which becomes particularly problematic due to the absence of system-level enforcement. To model this, MRC-Detector regards the previously configured multi-window focus-aware callbacks as the *sources*, since they indicate key transitions in an app's visibility and interaction state. Meanwhile, the resource management APIs are treated as the *sinks*, as they directly govern the acquisition and release of system resources. The complete details of the sources and sinks are demonstrated in Table III. We then examine whether there exists a feasible path from the *source* to the *sink*. Due to the lack of system-level enforcement in multi-window mode, issues occur by default if resource acquisition and release are not explicitly handled by apps. Therefore, MRC-Detector checks whether the app acquires resources upon gaining focus (`MRC-Type I`) and releases resources upon losing focus (`MRC-Types II & III`). If no such path is found, it indicates that the corresponding resource is not properly managed under multi-window execution, thereby revealing a potential MRC.

To reduces redundant path matching, MRC-Detector adopts a bottom-up taint tracking approach in practice. First, it builds a complete *Call Graph* (`CG`) of the target app using `Soot`. Then starting from each *sink API*, a reverse traversal is performed on the `CG` to collect all caller methods and corresponding `Activities`. The MRC-Detector then checks

TABLE II: MRCs in HAL Resources.

Attribute	Exclusive							Shared						
	Aud	Cam	Fin	Mic	NFC	Tel	USB	Aud	Blu	Loc	Sen	Tel	Tou	WiF
MRC-Type	I & II	I & II	I & II	I & II	II	✗	✗	III	✗	✗	✗	✗	III	✗
Consequence	①	①④	②③	①	①③	*N/A*	*N/A*	②	*N/A*	*N/A*	*N/A*	*N/A*	②⑤	*N/A*

Note: ① Functional failure, **② Non-user-intended effects**, **③ Data Leakage**, **④ App crash**, **⑤ Black screen**.

whether the traversal path can be matched to a *source callback* that contains the correct resource access logic. If no such match is found, the case is identified as a potential MRC issue. During this process, different types of MRCs are handled in a differentiated manner based on their specific triggering patterns and analysis requirements. For the detection of `MRC-Type I` (Unresumed Access) and `MRC-Type II` (Repeated Preemption), the *sink APIs* should be set as *resource start APIs*; while for the detection of `MRC-Type III` (Shared Hijacking), they should be set as *resource release APIs*. In particular, the `MRC-Type II` requires additional checking to determine whether the *sink APIs* may be repeatedly invoked, which is implemented either inter-procedurally or intra-procedurally: ❶ For inter-procedural repetition, MRC-Detector utilizes the `LoopFinder` tool [17] to identify *strongly connected components* in the `CG` that include the target API as a node, thereby detecting recursive or indirect cyclic invocations. ❷ For intra-procedural repetition, MRC-Detector records the ancestor methods (i.e., the callers in the invocation path) of the target API and further confirms them through *Control Flow Graph* (`CFG`) analysis in the next stage.

TABLE III: Sources and Sinks for MRC Detection.

Res Type	**Source** (Focus-awareness Callback)

All(◐)	`onWindowFocusChanged()`
	`onTopResumedActivityChanged()`
	`hasWindowFocus()`
	`OnFocusChangeListener.onFocusChange()`
	`onAudioFocusChange()` *(only for Audio)*

Res Type	**Sink** (Resource Management API)

Aud(●)	`AudioTrack.play()`
	`MediaPlayer.start()`
	`SoundPool.play()`
	`ExoPlayer.play() & setPlayWhenReady()`

Cam(●)	`Camera.open()` `CameraManager.openCamera()`

Fin(●)	`android.BiometricPrompt.authenticate()`
	`androidx.BiometricPrompt.authenticate()`
	`FingerprintManager.authenticate()`

Mic(●)	`AudioRecord.startRecording()` `MediaRecorder.start()`
NFC(●)	`NfcAdapter.enableForegroundDispatch()`

Aud(○)	`AudioManager.abandonAudioFocusRequest()`
	`MediaPlayer.pause() & release()`
	`AudioTrack.pause() & stop() & release()`
	`SoundPool.pause() & release()`
	`ExoPlayer.pause() & stop() & release()`

Tou(○)	`InputMethodManager.hideSoftInputWindow()` `InputMethodService.onDestroy()`

Legend: ● Exclusive, ○ Shared, ◐ Both Shared and Exclusive.

Control Flow Analysis. After obtaining feasible call path, MRC-Detector continues to perform code logic analysis of the *source callback* to further verify its trigger behavior. ❶ For `MRC-Type I & III`, MRC-Detector constructs the `CFG` for the *source callback*, and verifies whether the target *sink API* resides in the correct conditional branch: `hasFocus` branch for `MRC-Type I` and `!hasFocus` branch for `MRC-Type III`. If the *sink API* violates the expected conditional branch, it is identified as an MRC issue. ❷ For `MRC-Type II`, MRC-Detector builds `CFG` of the ancestor method to determine whether the *sink API* is located inside a *loop*, which indicates a repeated access. If so, the corresponding `Activity` is identified with an MRC issue.

2) Evaluation: The effectiveness of MRC-Detector is evaluated through detection and verification on real-world apps.

Dataset. To evaluate the effectiveness of MRC-Detector in detecting MRC issues, we collected a dataset comprising 200 top-ranked apps from the `TOP_FREE` list of `Google Play`, and 238 recently updated apps (in March 2025) from the `F-Droid` open-source platform. Guided by the permission declarations and resource-related descriptions on the app introduction pages, we filtered 235 resource-related apps covering all the tested 7 types of resources (see Table IV), including 166 from `Google Play` and 69 from `F-Droid`.

TABLE IV: Distribution of Resource-related Apps.

Source	Aud	Cam	Fin	Mic	NFC	Tou
F-Droid	38	48	26	31	2	2
Google Play	30	161	93	49	41	1

Metrics. We evaluate the effectiveness of MRC-Detector using standard quantitative metrics, namely $precision$, $recall$, and $F1$-$score$. Let N_{MRC} denote the set of apps that truly have MRC issues (*ground truth*), and $\hat{N}_{MRC}$ denote the set of apps detected as MRC-prone by the MRC-Detector. Then

$$Precision = \frac{|N_{MRC} \cap \hat{N}_{MRC}|}{\hat{N}_{MRC}} \tag{3}$$

$$Recall = \frac{|N_{MRC} \cap \hat{N}_{MRC}|}{N_{MRC}} \tag{4}$$

$$F1 - score = \frac{2(Precision \times Recall)}{Precision + Recall} \tag{5}$$

Manual Verification. Since no existing *ground truth* MRC-prone apps (N_{MRC}), we construct it by manual verification to support the effectiveness evaluation of MRC-Detector. However, manually triggering the target MRCs involves several preconditions, including launching the multi-window mode, invoking the resource-related functionality, interacting between windows, etc, which may introduce severe human bias. To mitigate such bias, we carefully designed the manual verification process as follows: ❶ We first recruited three undergraduate student volunteers to serve as evaluators. Each

TABLE V: Detection Results of MRC-Detector.

Attribute	Exclusive Resource									Shared Resource		Total
	Audio		Camera		Fingerprint		Microphone		NFC	Audio	Touchscreen	
Type	I	II	I	II	I	II	I	II	II	III	III	
Apk	86.57%(58/67)	1.49%(1/67)	83.09%(172/207)	7.25%(15/207)	63.87%(76/119)	5.88%(7/119)	76.00%(38/50)	18.00%(9/50)	6.98%(3/43)	73.85%(48/65)	66.67%(2/3)	82.55%(194/235)
Precision	96.55%(56/58)		94.77%(163/172)		93.42%(71/76)		94.73%(36/38)		66.67%(2/3)	97.92%(47/48)	50%(1/2)	94.71%(376/397)
Recall	98.25%(56/57)		96.45%(163/169)		95.95%(71/74)		87.80%(36/41)		66.67%(2/3)	100%(47/47)	50.00%(1/2)	94.95%(376/396)
F1-score	97.40%		95.60%		94.67%		91.13%		66.67%	98.95%	50.00%	94.83%

evaluator received detailed training on the MRC causes, consequences and triggering procedures. We then provided them with standardized trigger guides, summarizing step-by-step procedures for each resource type in a pattern-based format. Using these guides, the evaluators individually verified all apps in our dataset. In cases of disagreement, the majority vote was used to determine the final result. ❷ Each target app was paired with our standard test app under multi-window mode to trigger resource access. The multi-window mode (e.g., `FS+FF`, `FS+PIP`) was randomly chosen by the evaluators, and the target app was consistently placed in the smaller window (e.g., `FF`, `PIP`). ❸ Given that the resource-related functionalities vary across apps, evaluators identified the entry points of such functionalities by discussion on the app's description. In total, the manual verification involves 564 test runs, with each evaluator spending over 28 hours on average.

Results. According to the results illustrated in Table V, MRC-Detector achieved an overall detection *f1-score* of 94.83%, where both the *precision* and *recall* are over 94.5%, demonstrating high detection reliability. The lowest detection precision was observed for `NFC` and `TouchScreen` resources, though these cases were rare and thus not statistically representative. The reasons for false negatives are two-fold: ❶ Third-party library encapsulation: Some apps employ external libraries (e.g., `KingZxing` for camera-based *QR scanning* or `Superpowered` for audio processing), which often restructure API calls during packaging and optimization. This makes it difficult for MRC-Detector to accurately reconstruct the complete resource access paths. ❷ Asynchronous invocation paths: Scenarios involving concurrent task scheduling (e.g., via `ThreadPool`) or lifecycle-based callbacks introduce significant non-determinism, which pose challenges for static analysis to precisely capture the actual resource access paths. Correspondingly, the false positives were caused by apps employing advanced techniques such as *pluginization*, *hot-patching* or *packing protection*. In such cases, only a stub or shell component is registered in the `AndroidManifest.xml`, while the actual permission declaration, `<uses-feature>` tags and resource-related logic, are dynamically loaded at runtime using mechanisms such as *reflection* and *custom class loaders* (e.g., `DexClassLoader`). Since such dynamically loaded information is invisible during static analysis, MRC-Detector is unable to accurately identify actual access behaviors of the resources, leading to missed detection of relevant access paths. Moreover, as shown in Table V, 82.55% of the analyzed `APKs` were detected to have MRC issues, highlighting the prevalence and severity of the problem, which is further substantiated in a subsequent large-scale study. In addition, the number of `MRC-Type II` instances is significantly lower than the other two types. The underlying reasons for this discrepancy will be discussed in detail in the following RQ3 (§V-C).

Finding 2: MRC-Detector achieved an *f1-score* close to 95%, demonstrating its excellent detection accuracy and practical effectiveness. The false positives and false negatives were mainly caused by the use of specialized third-party libraries and app protection techniques. Among all types, `MRC-Type II` exhibited the lowest occurrence and the weakest detection performance.

C. RQ3: Prevalence of Large-scale Real-world Apps.

We then investigate the prevalence of MRCs in the real-world through automated detection on a large-scale set of real-world apps.

Datasets. The large-scale dataset is collected from both `F-Droid` and `Google Play`. For `F-Droid`, we developed automated scripts to crawl all categorized apps with a total number of 3,438. For the `Google Play` store, we initially collected a dataset of 310,695 apps that had been downloaded more than $10k$ times since 2013. Next, we applied resource-based configuration filters to identify apps that involve MRC-prone resources, yielding a refined subset. In addition, to further support security-relevant analysis, we used labels from the `AndroZoo` dataset [18] to classify this subset into benign and malicious apps (i.e., flagged by at least one `VirusTotal` engine [19]). Table VI summarizes the detailed statistics of this large-scale dataset.

TABLE VI: Statistics of Large-scale Real-world Apps.

App	Total	F-Droid	Google Play	Benign	Malicious	0–1M	1–5M	5–20M	>20M
Aud	31,909	220	31,689	31,552	357	468	1,388	13,460	16,593
Cam	126,748	464	126,284	125,247	1,501	3,968	10,618	50,400	61,762
Fin	38,982	179	38,803	38,531	451	1,145	3,460	12,240	22,137
Mic	12,975	247	12,728	12,806	169	369	1,273	4,986	6,347
NFC	10,603	98	10,505	10,492	111	592	503	3,508	6,001
Tou	1,822	10	1,812	1,805	17	548	481	644	149
Total	160,369	677	159,692	158,475	1,894	4,615	14,828	64,447	76,479

Results. MRC-Detector presented a high usability during the detection, successfully analyzing 158,587 out of 160,369 apps (approximately 98.89%). This is primarily attributed to the fact that MRC-Detector focuses on the methods belonging to the `Android SDK` or framework-level APIs, ensuring robust resilience against code obfuscation. Even after being obfuscated with `ProGuard` [20] or `R8` [21], the target app can be effectively recognized and detected by the MRC-Detector. We then study the MRC issues from different perspectives in the real-world large-scale dataset, summarized in Fig. 6. On average, 72.64% of the apps are detected as MRC issues, shown in Fig. 6(a), highlighting the overall prevalence of MRCs in real-world apps. Among them, `Microphone` stands out as the most MRC-prone resource with the highest *MRC percentage* (the proportion of apps with MRCs among all tested apps) with 88.44%. This can be primarily attributed

to the default behavior in earlier versions of Android, where apps were allowed to continuously occupy the microphone without enforced focus arbitration [22]. For the MRC type, `MRC-Type I` occurs most frequently, while `MRC-Type II` is the least. This is because `MRC-Type II` typically requires developers to intentionally exploit multi-window mode to perform focus preemption, which is uncommon among legitimate apps. In contrast, `MRC-Type I` often arises due to a lack of awareness of the complex behavior of multi-window execution, resulting in missing logic for reacquiring resources after focus is regained. Since this type of logic lies outside the core functionality, it is easily overlooked by developers.

Additionally, we comprehensively study the MRC performance in dimension of data source, security and `APK` size. Fig. 6(b) compares the prevalence of MRC issues between open-source `F-Droid` and commercial `Google Play`, where `Google Play` apps show a relatively higher *MRC percentage* (exceeding 75%). The reason lies in that commercial apps tend to incorporate more complex resource access functionalities, which increases the likelihood of MRC occurrences. Fig. 6(c) further explores the *MRC percentage* from a security perspective by comparing benign and malicious apps. From the result, malicious apps tend to exhibit more MRC issues, which aligns with the intuition that malware is more likely to engage in data hijacking behaviors, particularly those associated with `MRC-Type II & III`. Notably, the `Camera` and `Microphone` are the most frequently targeted resources for MRCs in malicious apps, as they are directly linked to sensitive environmental and personal data. Fig. 6(d) illustrates the relationship between MRC occurrence and the distribution of `APK` sizes. Most apps smaller than 1MB come from `F-Droid` and are designed as lightweight tools targeting specific hardware resources. Constituting less than 5% of the total, they surprisingly exhibit the highest MRC percentage (up to 83% on average). This is mainly because small apps are mostly maintained by individual developers or small teams. Due to short development cycles and limited testing resources, these apps often overlook the management of resource access under multi-window mode.

Finding 3: Over 70% of real-world apps were detected to exhibit MRC issues, highlighting the prevalence of the problem. Comparatively, MRCs are more prevalent in malicious, commercial, and small-sized apps. `Camera` and `Microphone` are identified as the most MRC-prone resources.

D. RQ4: Confirmation of Developers.

To further investigate how MRC issues are recognized by the developers, we submitted the identified MRC instances to the vendors of top-ranked apps as well as to the maintainers of open-source app repositories.

1) Confirmation of Vendors SRCs and Authorities: We first investigate the real-world response of commercial app vendors to MRC issues. To this end, we have submitted the MRC reports to the *Security Response Center* (*SRCs*) of 8 top-ranked Chinese app vendors. Each submission includes *issue description*, *triggering process*, *screen recording as evidence* and *mitigation suggestion*. Up to now, 5 *SRCs* have positively responded to the reported MRC issues, as shown in Table VII. Among them, the *SRCs* of `Baidu`, `AliPay` and `JingDong` confirmed the reported MRC issues as *medium-risk* and *low-risk* vulnerabilities, and awarded corresponding bounties. Furthermore, all three vendors, i.e., `Baidu`, `AliPay` and `JingDong`, have adopted the mitigation strategies provided in the submissions to complete the issue fixes, and released updated versions with the issue resolved. The remaining 3 vendors, i.e., `Cainiao`, `VulBox` and `Alibaba Cloud`, also recognized the existence of MRC issues, but the potential impact of the MRC issue on their app is still under evaluation and verification. In addition, the submission was recognized by `Google Bug Hunters` with severity *S2* and officially registered in the `China National Vulnerability Database (CNVD)` under `CNVD-2025-13057`.

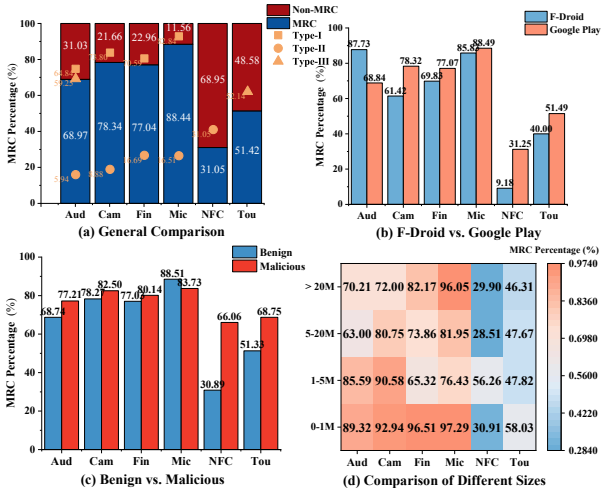

Fig. 6: Results on Large-scale Real-world Apps.

TABLE VII: Confirmation from Top-ranked Apps.

SRC	Consequence	Confirmation	Status
Baidu SRC	Client App DoS	Medium risk	Fixed
Alipay SRC	Local DoS	Low risk	Fixed
Du Xiaoman SRC	Design Defect / Logical Error	Positive Response	Fixing
JingDong SRC	Mobile and IoT - Android	Medium risk	Fixed
Kwai SRC	Logic Error	Positive Response	Fixing
Google Bug Hunters	Android & Devices VRP	Severity S2	Infeasible
CNVD	Common	Medium risk with `CNVD-2025-13057`	Undisclosed

2) Recognition of Open-source Repositories: Subsequently, we explore the awareness and recognition of MRC issues in the open-source community. To this end, we submitted MRC issue reports to 10 representative `GitHub` repositories, including *issue description*, *expected behavior* and *suggested solution*. As of now, 9 `Github` repositories have responded *positively* to our suggestions, as shown in Table VIII.

Finding 4: Among the submitted MRCs, the majority of developers responded positively with confirming and even rewarding the submissions. This includes 8 commercial app vendors, 2 external authorities and 9 open-source project maintainers, indicating that MRC issues are highly recognized by the developers.

TABLE VIII: Confirmation from Open-source Repositories.

Name	Category	Version	Size	Stars	Issue ID
Jitsi Meet	Phone & SMS	24.5.0	20MB	25.6k	15475
SnapSaver	Multimedia	0.7.0	23.3MB	29	4
Cavity	Multimedia	1.8.0	13MB	9	61
SBW	Money	2.5.9	36MB	247	201
GeoNotes	Multimedia	1.7.1	6.4MB	69	134
Session	Phone & SMS	1.23.2	128MB	269	870
Invoice Ninja	Phone & SMS	5.0.173	49MB	558	704
Trail-Sense	Navigation	6.9.0	8.7MB	1.5k	2769
BinaryEye	Multimedia	1.66.0	5MB	1.7k	547

E. Mitigation Discussion.

Based on issue analysis and manual validation, we summarize the feasible mitigation strategies into three categories.

User-side Mitigation. At the user side, apps can introduce specific exception handling structure, e.g., `try-catch`, around the MRC. When an MRC-related exception is caught, the app may invoke a warning dialog to inform the user that the requested resource may be preempted or hijacked in the current multi-window context, advising the caution in further use. For example, `Baidu Browser` displays a prompt for warning when the `Camera` permission request fails under multi-window mode. Although such warnings cannot fundamentally eliminate MRC issues, they can significantly reduce user misjudgment, helping users recognize abnormal MRC behavior and avoid unintended consequences.

System-side Mitigation. A more thorough solution to MRCs lies in establishing system-level resource awareness and recovery mechanisms. For instance, `Sensor` resources (e.g., accelerometers) do not require runtime permissions, allowing background or non-focused apps under multi-window mode to silently monitor and collect data from other windows. Such MRCs stemming from OS-level defects are difficult to detect or prevent at the user or developer level. If Android were to introduce lifecycle-aware enforcement and permission constraints for resource access under multi-window execution, e.g., by bounding the `Sensor` access to the window focus.

Developer-side Mitigation. To mitigate the impacts of MRC issues, app developers should override the appropriate *source callbacks* (as shown in Table III) based on the type of resource being accessed and monitor window focus changes to adjust resource behavior accordingly. ❶ For `MRC-Type I`, when an app window regains focus, the developer should explicitly check resource's availability and reinitialize the resource request to prevent unexpected crashes caused by system-level release. For instance, in the case of the `Camera` resource access (Fig. 1 in §II), developers should reinitialize the camera upon focus regain under the multi-window mode, where the representative code logic can be designed as: `if(camera==null){camera=Camera.open(); camera.setPreviewDisplay(surfaceHolder); camera.startPreview();}`. ❷ For `MRC-Type II`, developers should explicitly manage the lifecycle of resource usage, ensuring that resources are properly released when the window loses focus, in order to avoid repeated resource holding. For example, when accessing the `Audio` resource, developers should invoke `abandonAudioFocusRequest()` upon focus loss to release the `Audio` focus; as for the `Camera` access, the *release()* method should be called within `onPause()` or `onStop()` to ensure the *Camera* resource is properly released. ❸ For `MRC-Type III`, the app should release resources or restrict related functionalities when the window loses focus. For example, if the *Audio* is being played via `MediaPlayer`, the playback should be stopped and the `MediaPlayer` object should be released when the `Audio` focus is lost. Thus, the overlapping outputs under multi-window mode can be avoided. Similarly, in black-screen protection cases involving the `FLAG_SECURE` flag (Fig. 2 in §II), developers should precisely target sensitive content areas using `window.getDecorView()` when the window loses focus, instead of indiscriminately applying the flag to the entire window. Thus, unintended disruption of rendering in other visible windows can be prevented.

VI. Threats to Validity

Static Analysis Limitations. MRC-Detector is built on static analysis using the `Soot` framework, which may suffer from over-approximation and incomplete modeling of dynamic behaviors. Some control-flow and lifecycle transitions, especially those influenced by reflection, dynamic code loading, or third-party `SDKs`, may not be fully captured, potentially leading to false negatives during MRC detection. To address this, we plan to integrate hybrid analysis techniques by combining static analysis with lightweight runtime instrumentation to improve the detection.

Coverage of MRC Types. Our study focuses on three representative types of MRCs, derived from typical focus transitions and resource usage patterns in multi-window mode, which covers the vast majority of non-user-driven resource-access scenarios and to provide an actionable rule basis for automatic detection. However, these three types do not necessarily encompass all real-world MRCs: edge cases or complex interaction patterns, such as cascading focus shifts, background service interference, and multi-resource dependencies, may also lead to subtle MRC variants that are not captured by our current types. Future improvement can extend coverage by mining additional conflict patterns through large-scale runtime logging and anomaly clustering, thereby expanding the type system and detection rules to improve the practicality.

Runtime Validation. While we manually confirmed the MRC instances and real-world consequences for hundreds of representative cases, we did not perform runtime validation across all $150k+$ analyzed apps. Therefore, although our study provides strong statistical evidence of MRC risks, some reported issues may not occur under specific runtime conditions. For example, if two apps respectively use the front-facing and rear-facing cameras, they share the same API surface but actually access different physical devices. Since static analysis cannot reliably distinguish lens facing and other fine-grained resource configurations, the detector may flag a false conflict. As part of future work, we aim to develop a dynamic validation

module to simulate focus transitions and window interactions, enabling automated and scalable confirmation of suspected MRC instances.

VII. Related Work

Android Muti-window Analysis. While enabling user the multitasking capabilities [1], Android's multi-window mode introduces functional and security risks [23], [24]. Prior works have revealed various multi-window risks, including black screen abnormalities [25], transparent overlay deception [26], pop-up interference [27], energy inefficiencies [28] and window hijacking [6]. In response, recent efforts have developed automated detection strategies, targeting at the malicious overlays [29] and abnormal window interactions [30]. *In this study, we specifically focus on an underexplored conflict issue related to the resource access under Android multi-window mode.*

Resource Access Conflicts. The resource access conflicts [31] occur when multiple entities compete for shared or exclusive resources, which can potentially cause functional failures [32] or inconsistent executions [33]. Existing efforts on this topic predominantly focus on *permission abuse*, *over-privileged behavior* and *permission enforcement*. Permission abuse explores malicious resource hijacking through deceptive permission exploitation [34]–[36]. Over-privileged behavior examines unintended access risks induced by redundant permissions [37]–[39]. Permission enforcement involves designing fine-grained runtime control to mitigate conflicts [40]–[42]. *In contrast to these studies, we aim to investigate resource access conflicts in multi-window mode, revealing the corresponding impacts in novel usage scenario.*

Static Anomaly Detection. Static anomaly detection for Android focuses on identifying potential code defects and enabling scalable analysis for large-scale apps [43] [44]. Some existing efforts target at concurrency defects such as race conditions and synchronization issues in multi-threaded or event-driven code [44]–[46]. Other approaches analyze resource mismanagement and API misuse, addressing problems such as unreleased resources and improper sequences of API calls [47]–[49]. Inter-component communication analysis detects risks such as data leakage, privilege escalation and intent spoofing across app components [50]–[52]. Additionally, permission and policy violation detection investigates issues such as missing runtime permission check, over-privileged declaration and inadequate context-aware access control [53]–[55]. *While most existing static detection techniques focus on intra-app anomalies, our approach targets at concurrent resource conflicts across windowed apps.*

VIII. Conclusion

This work presents the first systematic study of Multi-window Resource Conflicts (MRCs) on Android. We formally define the concept of MRC, analyze its root causes and categorize it into three representative types based on triggering patterns and behavior sequences. To detect MRCs at scale, we develop MRC-Detector, a static analysis tool built on `Soot`, and validate its effectiveness through strict manual confirmation. The detection results on large-scale real-world apps reveal that MRC issues are prevalent and overlooked by both Android system and app developers. We further examine the impacts of MRCs and propose mitigation strategies, some of which have been confirmed and recognized by predominant app vendors. For future work, we plan to extend MRC-Detector with dynamic analysis capabilities and integrate runtime monitoring to detect and mitigate MRCs in real time.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (No. 62572256, 62032012, 62002177, 62172435), the National Key Research and Development Program of China (No.2022YFB3102900), the Natural Science Foundation of Tianjin of China (No. 23JCYBJC00320, 21JCZDJC00740), the Tianjin Key Science and Technology Project, (No. 24HHXCSS00004), and the Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation (No. CSSAE-2023-001).

References

[1] A. Developers, "Multi-window support," 2023. [Online]. Available: https://developer.android.google.cn/guide/topics/large-screens/multi-window-support

[2] R. Li, W. Diao, S. Yang, X. Liu, S. Guo, and K. Zhang, "Lost in conversion: exploit data structure conversion with attribute loss to break android systems," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5503–5520.

[3] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *2021 IEEE Symposium on security and privacy (SP)*. IEEE, 2021, pp. 70–86.

[4] X. Zhang, Z. Yu, X. Li, C. Zhang, C. Sun, N. Zhang, and R. H. Deng, "Understanding the bad development practices of android custom permissions in the wild," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[5] L. Ying, Y. Cheng, Y. Lu, Y. Gu, P. Su, and D. Feng, "Attacks and defence on android free floating windows," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 759–770.

[6] C. Guo, T. Wang, Q. Wang, N. Dong, X. Luo, and Z. Liu, "Fratricide! hijacking in android multi-window," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[7] H. Zhou, S. Wu, C. Qian, X. Luo, H. Cai, and C. Zhang, "Beyond the surface: Uncovering the unprotected components of android against overlay attack," in *The 31st Network and Distributed System Security Symposium*, 2024.

[8] J. Liu, D. Wu, and J. Xue, "Tdroid: Exposing app switching attacks in android with control flow specialization," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 236–247.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," ser. CASCON '99. IBM Press, 1999, p. 13.

[10] J. Kim, T. Kim, and E. G. Im, "Survey of dynamic taint analysis," in *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*, 2014, pp. 269–272.

[11] "Support for picture-in-picture," 2023. [Online]. Available: https://developer.android.google.cn/guide/topics/ui/picture-in-picture

[12] K. Yu, C. Wang, Y. Cai, X. Luo, and Z. Yang, "Detecting concurrency vulnerabilities based on partial orders of memory and thread events," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 280–291.

[13] B. Cui, M. Wang, C. Zhang, J. Yan, J. Yan, and J. Zhang, "Detection of java basic thread misuses based on static event analysis," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1049–1060.

[14] "Android uses-feature documentation," https://developer.android.com/guide/topics/manifest/uses-feature-element.

[15] "Android sdk documentation," https://developer.android.com/reference.

[16] "Apkparser: Android apk parsing library," https://github.com/hsiafan/apk-parser.

[17] "Loopfinder: A tool for detecting critical loops in android apps," https://github.com/brookcub/LoopFinder.

[18] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[19] "Virustotal," https://www.virustotal.com.

[20] "Proguard:java and android optimizer," https://github.com/Guardsquare/proguard.

[21] "R8:enable app optimization," https://developer.android.com/topic/performance/app-optimization/enable-app-optimization.

[22] "Sharing audio input," 2023. [Online]. Available: https://developer.android.google.cn/media/platform/sharing-audio-input

[23] N. KIMATA and A. TAKEMURA, "Development of multi-window style supporting system for recognition of societal problem based on ism," *Doboku Gakkai Ronbunshu*, vol. 1992, no. 449, pp. 203–212, 1992.

[24] C. Kumar, K. Naik *et al.*, "A study on user interaction with multi-window screen support framework for multitasking operations on smartphones," in *2017 14th IEEE India Council International Conference (INDICON)*. IEEE, 2017, pp. 1–6.

[25] L. Zhu, P. Wang, O. Jaiteh, Q. Wang, and C. Guo, "An empirical understanding of black screen abnormality in android multi-view," in *2023 IEEE 6th International Conference on Computer and Communication Engineering Technology (CCET)*, 2023, pp. 107–111.

[26] L. Wu, C. Wang, T. Liu, Y. Zhao, and H. Wang, "From assistants to adversaries: Exploring the security risks of mobile llm agents," 2025. [Online]. Available: https://arxiv.org/abs/2505.12981

[27] A. AlJarrah and M. Shehab, "Maintaining user interface integrity on android," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2016, pp. 449–458.

[28] G. Singh, M. K. Rohit, C. Kumar, and K. Naik, "Multidroid: An energy optimization technique for multi-window operations on oled smartphones," *IEEE Access*, vol. 6, pp. 31 983–31 995, 2018.

[29] L. Gong, Z. Li, H. Wang, H. Lin, X. Ma, and Y. Liu, "Overlay-based android malware detection at market scales: Systematically adapting to the new technological landscape," *IEEE Transactions on Mobile Computing*, vol. 21, no. 12, pp. 4488–4501, 2021.

[30] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, "Understanding and detecting overlay-based android malware at market scales," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 168–179.

[31] J. Fu, Y. Wang, Y. Zhou, and X. Wang, "How resource utilization influences ui responsiveness of android software," *Information and Software Technology*, vol. 141, p. 106728, 2022.

[32] Y. Zheng, C. Li, Y. Xiong, W. Liu, C. Ji, Z. Zhu, and L. Yu, "iaware: Interaction aware task scheduling for reducing resource contention in mobile systems," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–24, 2023.

[33] O. Riganelli, D. Micucci, and L. Mariani, "Controlling interactions with libraries in android apps through runtime enforcement," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 14, no. 2, pp. 1–29, 2019.

[34] T. Osman, M. Mannan, U. Hengartner, and A. Youssef, "Securing applications against side-channel attacks through resource access veto," *Digital Threats: Research and Practice*, vol. 1, no. 4, pp. 1–29, 2020.

[35] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang, "Uncovering cross-context inconsistent access control enforcement in android," in *The 2022 Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[36] X. Wang, Y. Zhang, X. Wang, Y. Jia, and L. Xing, "Union under duress: Understanding hazards of duplicate resource mismediation in android software supply chain," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3403–3420.

[37] H. Gao, C. Guo, G. Bai, D. Huang, Z. He, Y. Wu, and J. Xu, "Sharing runtime permission issues for developers based on similar-app review mining," *Journal of Systems and Software*, vol. 184, p. 111118, 2022.

[38] Y. Wang, Y. Wang, S. Wang, Y. Liu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Runtime permission issues in android apps: Taxonomy, practices, and ways forward," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 185–210, 2022.

[39] J. Xiao, S. Chen, Q. He, Z. Feng, and X. Xue, "An android application risk evaluation framework based on minimum permission set identification," *Journal of Systems and Software*, vol. 163, p. 110533, 2020.

[40] X. Liu and K. Liu, "A permission-carrying security policy and static enforcement for information flows in android programs," *Computers & Security*, vol. 126, p. 103090, 2023.

[41] W. Enck, "Analysis of access control enforcement in android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, 2020, pp. 117–118.

[42] A. Sayyadabdi, "A framework for formal specification and verification of security properties of the android permissions system," *arXiv preprint arXiv:2203.09857*, 2022.

[43] J. Senanayake, H. Kalutarage, M. O. Al-Kadri, A. Petrovski, and L. Piras, "Android source code vulnerability detection: a systematic literature review," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–37, 2023.

[44] G. Hu, B. Zhang, X. Xiao, W. Zhang, L. Liao, Y. Zhou, and X. Yan, "Samldroid: a static taint analysis and machine learning combined high-accuracy method for identifying android apps with location privacy leakage risks," *Entropy*, vol. 23, no. 11, p. 1489, 2021.

[45] S. Malakar, T. B. Haider, and R. Shahriar, "Racefixer–an automated data race fixer," *arXiv preprint arXiv:2401.04221*, 2024.

[46] A. Costea, A. Tiwari, S. Chianasta, A. Roychoudhury, and I. Sergey, "Hippodrome: Data race repair using static analysis summaries," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–33, 2023.

[47] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou, "Boosting static resource leak detection via llm-based resource-oriented intention inference," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 668–668.

[48] R. B. Pereira, J. F. Ferreira, A. Mendes, and R. Abreu, "Extending ecoandroid with automated detection of resource leaks," in *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2022, pp. 17–27.

[49] J. G. Lima, R. Giusti, and A. C. Dias-Neto, "Leakpred: An approach for identifying components with resource leaks in android mobile applications," *Computers*, vol. 13, no. 6, p. 140, 2024.

[50] Y. Hu, Z. Jin, W. Li, Y. Xiang, and J. Zhang, "Siat: A systematic inter-component communication analysis technology for detecting threats on android," *arXiv preprint arXiv:2006.12831*, 2020.

[51] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1398–1409.

[52] A. Nirumand, B. Zamani, and B. T. Ladani, "A comprehensive framework for inter-app icc security analysis of android apps," *Automated Software Engineering*, vol. 31, no. 2, p. 45, 2024.

[53] S. Wang, Y. Wang, X. Zhan, Y. Wang, Y. Liu, X. Luo, and S.-C. Cheung, "Aper: Evolution-aware runtime permission misuse detection for android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 125–137.

[54] G. Xu, S. Xu, C. Gao, B. Wang, and G. Xu, "Perhelper: Helping developers make better decisions on permission uses in android apps," *Applied Sciences*, vol. 9, no. 18, p. 3699, 2019.

[55] A. Sayyadabdi, B. T. Ladani, and B. Zamani, "Towards a formal approach for detection of vulnerabilities in the android permissions system," *arXiv preprint arXiv:2208.11062*, 2022.