WinSpy: Cross-window Side-channel Attacks on Android's Multi-window Mode

ZENG LI, Shandong University, Jinan, Shandong, China

CHUAN YAN, The University of Queensland, Brisbane, QLD, Australia

LIUHUO WAN, The University of Queensland, Brisbane, QLD, Australia

HUI ZHUANG, Shandong University, Jinan, Shandong, China

PENGFEI HU, Shandong University, Jinan, Shandong, China

GUANGDONG BAI, The University of Queensland, Brisbane, QLD, Australia

View all

Open Access Support provided by:

The University of Queensland

Shandong University

PDF Download
3680207.3723482.pdf
25 December 2025
Total Citations: 0
Total Downloads: 332

Published: 03 November 2025

Citation in BibTeX format

ACM MOBICOM '25: 31st Annual International Conference on Mobile Computing and Networking
November 4 - 8, 2025
Hong Kong, China

Conference Sponsors:
SIGMOBILE

WinSpy: Cross-window Side-channel Attacks on Android's Multi-window Mode

Zeng Li[1], Chuan Yan[2], Liuhuo Wan[2], Hui Zhuang[3], Pengfei Hu[3], Guangdong Bai[2], Yiran Shen[1*]

[1]School of Software, Shandong University, China

[2]School of Information Technology and Electrical Engineering, The University of Queensland, Australia

[3]School of Computer Science and Technology, Shandong University, China

Abstract

With the development of the Android system and increasing screen size, the use of *multi-window mode* has become prevalent among users. However, the security and privacy implications associated with this mode have not been thoroughly investigated. This paper uncovers severe and unique security vulnerabilities in Android's multi-window mode, revealing several high-risk side-channels that facilitate diverse cross-window attacks, leading to significant breaches of user privacy. In detail, our research introduces *WinSpy*, a framework leveraging a newly discovered resource contention side-channel in multi-window mode to fingerprint app launches, web pages, and in-app activities, all without violating Android's permission framework. Our extensive evaluations demonstrate that *WinSpy* achieves high accuracy (from 70 to 80% detecting website and app launches to over 97% recognizing critical in-app activities). Additionally, we reveal that due to Android's lenient permission management for this mode, window apps can also use Inertial Measurement Unit sensors to launch attacks, such as inferring the user's touch positions outside the window with high precision. Furthermore, we propose systematic mitigations against these vulnerabilities.

CCS Concepts

• **Security and privacy → Operating systems security**; **Domain-specific security and privacy architectures**.

*Corresponding author

Keywords

Android security and privacy, Multi-window mode, Side-channel attack

ACM Reference Format:
Zeng Li[1], Chuan Yan[2], Liuhuo Wan[2], Hui Zhuang[3],, Pengfei Hu[3], Guangdong Bai[2], Yiran Shen[1]. 2025. WinSpy: Cross-window Side-channel Attacks on Android's Multi-window Mode. In *The 31st Annual International Conference on Mobile Computing and Networking (ACM MobiCom '25), November 3–7, 2025, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3680207.3723482

1 Introduction

Android has become one of the most popular mobile operating systems globally, commanding a market share of 70% in smartphones [8]. This success is driven by its open ecosystem, high flexibility, and rapid iteration capabilities. The platform boasts a vibrant ecosystem with over 2.6 million third-party apps, significantly enhancing productivity and enriching users' daily experiences [15]. When users download and start these apps, Android schedules threads based on whether each app is in the foreground or background, ensuring that the foreground app remains responsive and interactive while optimizing system resource utilization [26–28]. However, as user scenarios become more complex and interactions diversify, the need for multitasking and simultaneous access to information has grown, making it insufficient to display a single program on a single screen.

In response to the increasing demand for multitasking and simultaneous access to information, Android introduced *multi-window mode* [12] in version 7. This feature allows users to run and interact with multiple apps on the same screen concurrently, significantly enhancing multitasking capabilities on Android devices. This model enhances convenience and efficiency when browsing information and performing cross-app operations.

To further facilitate flexible app switching in the multi-window mode, Android introduced a *multi-resume mechanism* [7] in version 10. This mechanism maintains all the multi-window apps in the resumed state, their most active lifecycle phase, with similar priorities and permissions as

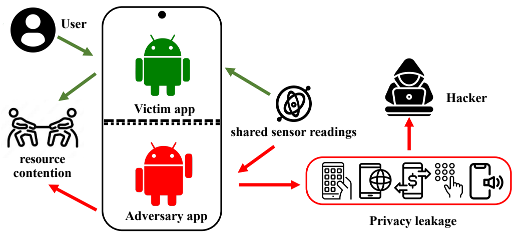


Figure 1: The adversary app of *WinSpy* launches cross-window side-channel attacks on the victim app via resource contention and shared sensor readings to infer the private information of the users.

foreground apps, allowing seamless switching with low delays.

Using Android smartphones in the multi-window mode has become a common daily scenario [19, 44]. For example, a user can watch Netflix and send messages on WhatsApp at the same time. Major Android manufacturers like Samsung, Motorola, and Google Pixel are developing devices with larger screens, including tablets and foldable phones, which are ideal for multi-window mode and greatly improve multitasking [24, 31, 32]. Consequently, supporting the multi-window mode for more efficient and convenient multitasking has become a key optimization objective when customizing operating systems and a major selling point in marketing these devices.

While multi-window mode offers users the convenience of multitasking, it also raises significant concerns regarding data exfiltration and information flow between concurrently running apps. On the one hand, malicious apps can exploit the Android GUI design, by generating fake or transparent floating windows to trick users into inputting sensitive content or passwords while interacting with a targeted app [5, 34, 54, 60]. A line of research has been dedicated to the detection of such problematic windows [20, 37, 40, 49]. On the other hand, a malicious app running in the background could potentially steal private information from foreground apps through side-channels that arise from fluctuations in resource consumption during user interactions [47, 51, 62, 63]. Android has mitigated these issues over time by tightening permissions management for background apps. However, no existing research has investigated the potential of app-level side-channel attacks enabled by the multi-window mode, leaving the crucial question of *whether side-channel attacks in multi-window mode can accurately steal private data* unresolved.

Our work. In this work, we propose *WinSpy*, a cross-window side-channel attack approach that leverages the multi-window mode to execute app-level attacks. To the best of our knowledge, *WinSpy* is the first comprehensive and systematic study on this topic. It explores the unique design of multi-window mode in the app priority management. In particular, app priorities in multi-window mode are set to be almost equal, rather than employing a more sophisticated adaptive priority management mechanism. This simplification makes multi-window apps behave like foreground apps, granting them high priorities to access side-channels with lenient permission management.

WinSpy explores two types of side-channels arising from this, as shown in Figure 1. First, in resource allocation for multi-window mode, each app competes for various runtime resources based on its needs. The system's lack of sophisticated security management in this contention leads to a neglect of safety in scheduling, allowing different contention patterns to reflect the underlying app behaviors. These patterns can be exploited to launch cross-app side-channel attacks. We are the first to discover that this side-channel is effective in launching cross-window attacks in the multi-window scenario at the app layer, while apps running in background mode cannot effectively launch such attacks due to their lower priority in resource contention. Second, in multi-window mode, all apps are granted permission levels almost similar to those in foreground mode. By exploiting this relaxation in permission management, an adversary app has full access to the embedded sensors and shares the same sensor readings with the victim app. These sensor readings can be utilized as side-channels to launch attacks, such as inferring tap positions [17, 56], classifying user's interactions with other apps [42, 45, 50] and even reconstructing sound played by other apps [18, 35]. However, due to Android's restrictions on background apps accessing sensors (unless permitted by the user), these attack threats are largely mitigated. Nonetheless, we find that window apps can still access sensors without permitted, thereby posing serious security issues.

According to the above insights about multi-window mode, we first construct the **Resource Contention Incursion** leveraging the problematic simplified resource allocation management. This technique infers various behaviors of the victim app by analyzing the characteristics of resource contention over CPU, memory, and disk usage. Specific attacks discussed in this paper include *App Launch Fingerprinting* to identify the start of an app, *Website Launch Fingerprinting* to recognize the website being browsed, and *In-app Activities Fingerprinting* to detect sensitive user activities within an app (Section 6). Additionally, based on issues arising from shared sensor readings from embedded sensors(e.g., accelerometer, and gyroscope), we devise **Motion Sensor Snooping**, where the adversary app reads sensor data while the user interacts with the victim app to identify actual user interactions with other apps (Section 7).

Through extensive evaluations on large-scale datasets collected from 21 popular apps and 21 mainstream websites on multiple Android devices, *WinSpy* achieves the recognition accuracy of 70.1% and 80.3%, on apps and website launching fingerprinting respectively. For in-app activities, the accuracy reaches over 97.6% for detecting critical activities. In the motion sensor snooping scenario, *WinSpy* accurately localizes the user's touch position, thereby inferring user inputs on a numeric keypad. Our study reveals significant security and privacy vulnerabilities in the current Android multi-window mode. These findings highlight the urgent need for enhanced security measures to protect user data across multiple concurrently running apps.

Contributions. The main contributions of this work are as follows.

- **Understanding the resource allocation mechanism in Android's multi-window mode and the security exposures it presents.** Contrary to the traditional foreground-background mode, we outline the differences in resource allocation and identify typical use scenarios in the current multi-window mode, highlighting three related security exposures.
- **Constructing a comprehensive side-channel attack model for multi-window mode.** Based on the attack model, we design *WinSpy*, a cross-window side-channel attack framework that exploits the security vulnerabilities of multi-window mode. Through extensive evaluations on various specific attack tasks, including fingerprinting of app launch, website launch, and in-app activities, and sensor-based tap localization inference, we demonstrate the efficacy of *WinSpy*.
- **Revealing security vulnerabilities in the Android resource scheduling in multi-window mode.** Our findings highlight the vulnerability of multi-window mode to side-channel attacks, revealing the current inefficiencies and insecurity in Android resource scheduling. Simultaneously, we provide possible mitigations to maintain a secure and robust Android ecosystem.

Ethics and Disclosure. We conducted all experiments within a private workspace, and all the apps tested were installed and tested on controlled devices. All discovered attacks have been ethically disclosed to Google.

Availability. The source code of *WinSpy* and additional evaluation results mentioned in the paper can be accessed at an open source repository [2].

2 Related Work

Privacy issue of Multi-window mode. Several studies have investigated the security implications of multi-window mode on mobile devices. AlJarrah et al. [5] investigate how floating windows can overlay applications and mislead users into providing sensitive information, potentially leading to unintended data disclosure. Building on this, Wang et al. [54] demonstrate that such overlays can also hijack biometric inputs like fingerprints. Ying et al. [60] examine the potential for UI spoofing attacks, where the generation of numerous windows can lead to a Denial of Service (DoS) attack on the phone, ultimately causing it to crash. To address these security issues, Ren et al. [49] developed WindowGuard, a system that evaluates the properties and activities of windows on mobile devices. It aims to block unauthorized overlays and prevent interaction hijacking by regulating the interactions among different window layers. In contrast to these studies which focus primarily on fraud in the UI and flaws in interactions, our work delves deeper into the underlying mechanisms of multi-window operation which causes privacy leakage under side-channel attacks.

Side-channel attacks by analyzing system statistics. There are existing studies on side-channel attacks utilizing the runtime statistics provided by Android APIs. However, due to restrictions imposed by Android, these APIs are no longer feasible. For example, Dipta et al. [30] exploited an Android 8.0 vulnerability, allowing background apps to infer user activities in foreground apps by analyzing CPU frequency patterns. Other studies used CPU statistics for malware detection [25, 43, 46, 63], such as analyzing background apps' CPU and memory usage to identify malicious software. There is also some work that utilizes timing side-channel, for example, Palfinger et al. [47] found that the execution times of function calls could indicate system status, that is, by analyzing the execution delays from file access functions, one could deduce the presence of specific files. Similar work on PC platforms involved behavior fingerprinting based on indirect information [4, 55]. For example, on Intel platforms, attackers used Port Contention on CPU functional units and inferred contending processes' behaviors from delay patterns. Additionally, studies [22, 23, 36, 51, 58, 59, 62, 64, 65] leveraged software architecture information, APIs, system open data, and hardware features to provide time-series data for conducting fingerprinting attacks on user behavior. However, most of the existing attacks rely on specific system APIs (often disabled in newer system versions) or are designed for non-Android platforms unsuitable for direct adaptation. For instance, CPU port contention-based attacks [4] depend on underlying hardware characteristics, which are obscured in Android applications through heavily encapsulated APIs and virtualized runtime environments. In contrast, our proposed contention-based side-channel attack solely relies on inter-app contention caused by the scheduling mechanism under Android's multi-window mode.

Side-channel attacks based on motion sensors. Studies on side-channel for motion sensors has a long history. As early as 2012, numerous studies focused on side-channel attacks involving motion sensors such as accelerometers on mobile phones. Some works [17, 56] classified the 10 numerical keys on a digital keypad. Other works [42, 45] implemented screen segmentation into multiple rows and columns, classifying user tap regions, with the latter also testing classification on a 26-letter virtual keyboard. Later, Singh et al.[50] classified genders based on subtle differences in motion habits when men and women used their phones. Matyunin et al.[41] found that different operational states of a mobile phone produced distinct electromagnetic signals that affected the magnetometer, which were used to classify the apps and web pages being used. Other works utilized the accelerometer to implement practical functionalities. Some studies [38, 48] used the accelerometer and gyroscope to recognize 3D motion trajectories of the phone, facilitating gesture recognition. Several works [38, 39] utilized a wristwatch's motion sensor to classify the positions of keys pressed on a physical QWERTY keyboard. With the advancement of deep learning technologies, some studies discovered that accelerometers were sensitive enough to capture vibration characteristics when the phone played sound out loud [18, 35]. The corresponding techniques above can be utilized to undertake side-channel attacks on Android's multi-window mode by exploiting the sensor readings of the smartphones.

3 Introduction to Multi-window Mode

Multi-window mode, initially an experimental feature in Android 6.0, was formally implemented in Android 7.0 [9]. It has been refined in later versions and encompasses three main scenarios (as depicted in Figure 2):

- **Floating-window (left):** This mode enables users to interact with apps in movable, resizable windows that float above other applications.
- **Picture-in-picture (middle):** introduced in Android 8.0, this mode is typically used for video playback. It can also be employed for regular interactions without video content.
- **Split-screen (right):** This mode divides the screen into two parts, either horizontally or vertically.

Process scheduling and resource allocation in multi-window mode. In the Android system, each app operates as a separate Linux process with its own sandbox for security and resource isolation [13]. Android employs the Linux scheduler to manage multiple apps running simultaneously. This includes applying fair scheduling for equitable allocation of CPU, memory, and so on, and preemptive rules enabling processes to autonomously secure necessary resources [14]. Furthermore, Android's scheduling is tailored to phone characteristics, prioritizing processes based on factors such as foreground-background status, multi-window status, interactivity, visibility, and so on, ensuring that higher-priority processes receive more resources during contention.

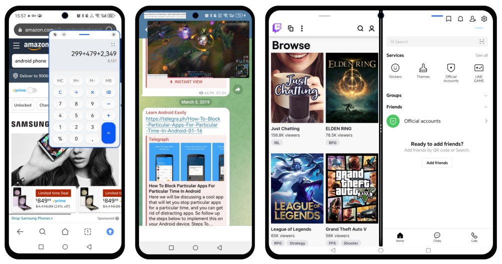


Figure 2: Three common multi-window mode scenarios: Left, a shopping website occupies the full screen with a calculator in a floating-window. Middle, an instant messaging app is in full screen while a video app is in picture-in-picture mode. Right, a live streaming app and a messaging app run in split-screen mode.

4 Exposure Definition

Unlike traditional foreground-background mode, multi-window mode focuses more on enhancing multitasking capabilities, which consequently introduces unique privacy exposures. We summarize three potential exposures of it according to its unique management mechanism on priorities, permissions, and processes to guide the design of subsequent attack framework.

Exposure 1: The Multi-Resume Mechanism. In early versions of Android, only one app could actively run in the foreground. Switching apps would pause or stop the initial one, triggering its `onPause()` or `onStop()` functions and relegating it to background operations like services. Android 10 revolutionized this by introducing the *Multi-Resume Mechanism* [7], enabling multiple apps to remain active and run concurrently in multi-window mode, thus boosting multitasking efficiency.

Since mobile apps typically require substantial resources like CPU during launching and performing specific functionalities, we find there is a contention for resources when multiple apps are active under the *multi-resume mechanism*. For example, when app α monopolizes limited resources, it forces app β to contend for them. In this scenario, α can observe consistent changes in system resource usage every time β performs a specific task, allowing α to infer β's operational behaviors based on their resource contention fingerprints.

Exposure 2: Continuous Monitoring. Table 1, referenced in [33], details the system's likelihood of terminating processes based on their activity states. Apps in the *Resumed* state, typically active in the foreground or multi-window modes, are least likely to be killed due to higher priority.

Conversely, background apps in the *Stopped* state face a higher killing likelihood as they have lower priority.

Table 1: Likelihood of process being killed based on State.

Likelihood of killing	Activity state	Process state
Lowest	Resumed	Foreground, Multi-window
Low	Started/Paused	Visible (no focus)
Higher	Stopped	Background (invisible)
Highest	Destroyed	Empty

Specifically, after a user finishes using an app and leaves it in the background, subsequently opening many other apps will progressively lower the priority of the background apps. Eventually, the system will freeze and terminate these background apps. In contrast, multi-window mode gives apps running in this state nearly the same priority as foreground apps, which remains unchanged even as more apps are launched. This management not only lowers the chances of an app being terminated but also opens doors for malicious monitoring between apps. For instance, a compromised window app can perform extensive monitoring of other window apps without system termination, a capability that background-running malicious apps cannot achieve. In our subsequent experiments, we find the adversary app running in multi-window mode consistently remains active, which provides a significant advantage for the adversary.

Exposure 3: Shared Sensors Readings. Recent off-the-shelf Android phones are equipped with various low-cost sensors to support multiple apps and services, such as Inertial Measurement Units (IMUs), proximity sensors, and lighting sensors. Foreground apps can access these sensors without user permission and at the maximum sampling rates (Maximum rate reading permission is a normal permission that doesn't require a user grant [6]). To prevent potential misuse and protect user data, starting from Android 9, background apps are prohibited from accessing these sensors at even the minimum rate, unless the user grants explicit permission [29].

However, this restriction does not apply to multi-window mode, where apps can always access these sensors at maximum rate, meaning the readings of these sensors are shared across apps. For example, the accelerometer is often used to monitor user activities, providing time-series data across the three axes at typically 300-500Hz. An app can use the shared IMU sensor readings to infer tap positions [39, 42, 56]. We adapt these attacks for window apps, allowing them to infer tap positions outside of the window.

5 Attack Overview

5.1 Attack Scenario

In our usage scenario, a user operates two apps concurrently in multi-window mode on an Android device for work or entertainment. For instance, one app may be streaming video content while the other is used for social networking. In multi-window mode, both apps run in parallel and can access the full system resources and sensors without the user's explicit permission.

However, one of these apps, referred to as the adversary app, has been compromised and is under the control of remote hackers. The adversary app is designed to conduct side-channel attacks on the other app, the victim. Specifically, by exploiting the multi-resume mechanism (Exposure 1) in multi-window mode, the adversary app can identify the victim app and monitor its in-app activities. This capability allows the adversary to capture a wide range of user activities, such as detecting app launches, monitoring web browsing, and observing transactional interactions within a banking app. Considering the high operation priority (Exposure 2), the adversary app in multi-window mode is highly unlikely to be terminated by the system. This ensures that the adversary can continuously monitor the activities of the other app for an extended period. Finally, the shared sensor readings (Exposure 3) allow the adversary app to access IMU sensors without the user's permission. These readings can be analyzed to deduce sensitive user interactions with the victim app. For example, by accessing IMU sensor readings, the adversary can infer typing inputs within the victim [39, 42, 56].

5.2 Adversarial Capabilities and Attacks Classification

Based on the definition of exposures introduced by Android's multi-window mode discussed in Section 4, we consider two major categories of side-channel attacks as the capabilities of *WinSpy*:

Resource Contention Incursion. Based on exposures 1 and 2 (The Multi-Resume Mechanism, Continuous Monitoring), the adversary app runs concurrently with the victim app, contending for limited resources like CPU at nearly equal priority levels. As the resource contention between the two apps significantly affects their own retained resources, the adversary can identify the characteristics of the victim by tracking changes in its own resource usage. The specific attacks derived from this point:

- **App Launch Fingerprinting (Section 6.3).** By identifying and monitoring the resource consumption characteristics during the launch of the victim app, adversary can determine its identity.

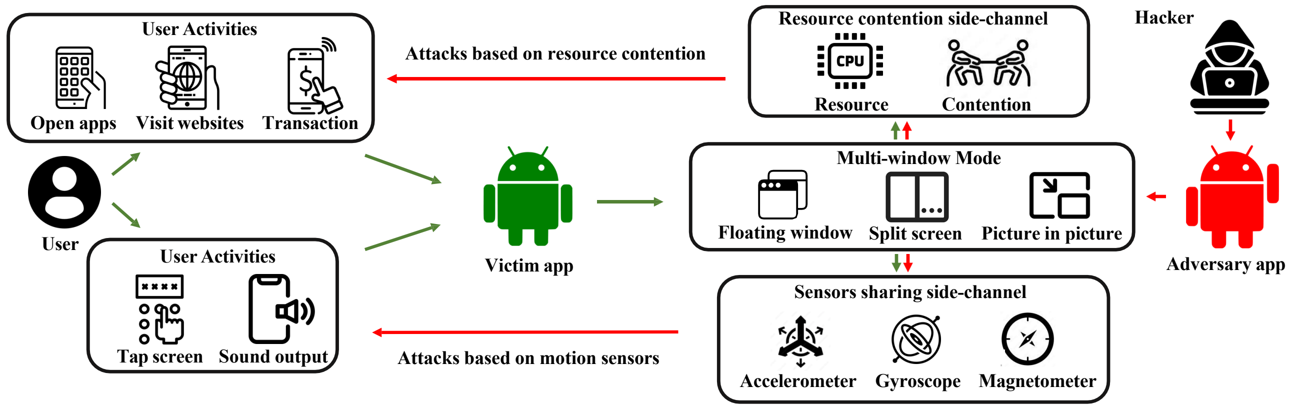


Figure 3: The pipeline of *WinSpy* for conducting a number of different side-channel attacks in multi-window mode.

- **Website Launch Fingerprinting (Section 6.4)**. When the victim app is a web browser, the adversary app can identify the webpage the user is browsing.
- **In-app Activities Fingerprinting (Section 6.5).** User activities within victim can cause observable changes, allowing the identification of sensitive activities.

Motion Sensor Snooping. Based on exposures 2 and 3 (Continuous Monitoring, Shared Sensors Readings), the attacker can read IMUs data at the maximum frequencies without requiring explicit user permissions, enabling them to infer the user's interactions with victim apps, termed as **IMU-based Tap location Inference** (Section 7.2). This attack relies on the finding that different screen tap locations generate distinct patterns in IMUs time-series, which can be analyzed to deduce where a user has touched on the screen, thus inferring the interactions between the user and the victim app, such as the input content.

6 Design and Evaluation on Attacks based on Resource Contention

In this section, we first introduce the design of the adversary app for resource contention incursions. We then evaluate and discuss the three specific types of attacks conducted by the adversary app, i.e., the app launch fingerprinting, website launch fingerprinting and in-app activities fingerprinting attacks.

6.1 Design of Adversary App

To introduce the contention between the adversary app and the victim app over the limited resource (e.g., CPU), we have configured the adversary app with multiple CPU resource-consuming threads. Additionally, the adversary also maintains a statistics thread that monitors the statistics related to the performance of the CPU resource-consuming threads, i.e., the total number of tasks accomplished. The detailed designs are as follows:

- **CPU Threads:** the CPU threads run intensive integer and floating-point computation tasks to occupy the computational resources of the mobile CPUs, continuously performing basic mathematical calculations such as ln(), exp(), sin(), etc.
- **Statistics Thread:** the statistics thread is responsible for counting the number of tasks completed by each CPU resource-consuming thread within a specific period, which is 10ms in this paper. The total number of tasks completed by all of the CPU threads is then saved along with the system time, forming a contention-tuple of two integers: $< T_s, N_{\mathrm{CPU}} >$ where, T_s is the timestamp, and N_{CPU} represent the total number of tasks completed by the CPU.

It is worth noting that we also considered contention over other resources, such as memory and disk usage. However, our evaluation reveals that a CPU-only setting achieves the best performance. This is because CPU threads handle basic computations, allowing a significant number of CPU tasks to be completed, which provides a wide value range. This increased task completion potential enables the observation of fine-grained characteristics during competition. Additionally, we found that the number of CPU threads in the adversary app plays a crucial role. Excessive threads can overload system resources and destabilize the app, while too few threads fail to create effective resource contention. An optimal number of threads is essential for maintaining stability and effective monitoring. Our evaluation shows that 10 CPU threads work well for app launch fingerprinting attacks, while 5 CPU threads are effective for website launch and in-app activity fingerprinting attacks. The detailed results for the type and number of contention threads are omitted due to page limits but can be found in the open-source repository.

6.2 Hardware and Evaluation Setup

The experiments are primarily conducted on the *Honor Magic V2*, a foldable smartphone running Android 14 (the latest Android version as of submission). This device is particularly convenient for use in multi-window mode, especially in split-screen mode due to its large screen. We ensure that the battery level remains above 50% to maintain stable device performance. To verify the generalizability of our experiments, we also employed additional devices including the P30 (released in 2019, Android 10), and Xiaomi Pad 6 (a tablet released in 2019, Android 13).

To evaluate each specific attack, we employ a simple one-dimensional Convolutional Neural Network (1D-CNN) as the classifier. We have tested other classifiers like LSTM and SVM as well, yet we ultimately selected the 1D CNN for its robustness. This network combinates seven convolutional layers for extracting features and four deconvolution layers for aggregating them, followed by two fully-connected layers and a softmax classifier. [1]. We apply a common 80%-20% split for training and testing, averaging multiple runs for our final results.

6.3 App Launch Fingerprinting

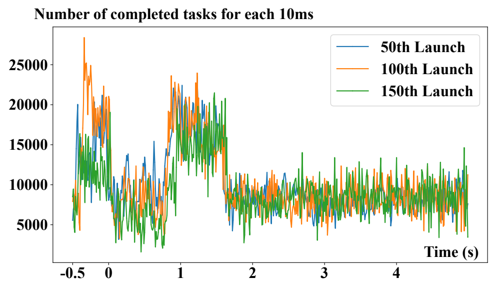
In this section, we conduct experiments to evaluate the accuracy of *WinSpy* in inferring the start of different apps through app launch fingerprinting.

6.3.1 **Experimental Procedure.** We select the top 21 popular apps from APKPure's [1] hotlist as the candidate victim apps including Telegram, Instagram, Pixiv, Tinder, Gmail, Google Translator, Facebook, among others. These 21 apps comprehensively cover the various primary activities and functions users engage in on mobile devices, including social networks, messaging, content streaming, and more. This broader selection allows us to evaluate the robustness and generalizability of the app launch fingerprinting attacks across a diverse set of applications (the complete list is shown in Figure 5).

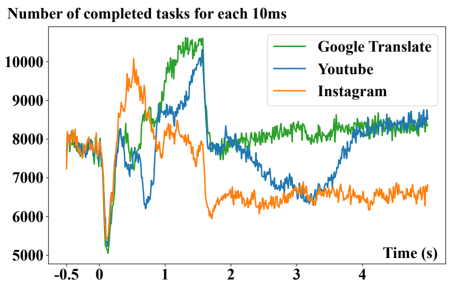
The experimental procedure is as follows: first, we open the adversary app in a floating window, which occupies 10 CPU resource-consuming threads to introduce intensive contention with one of the victim candidates. Then, we randomly tap to launch one of the 21 victim apps. After waiting for 5 seconds, we close the victim app. During the launch of each candidate victim app, the statistics thread of the adversary app monitors the contention and stores approximately 5 seconds of time-series data of the contention-tuple $< T_s, N_{CPU} >$. Since the statistics thread has a sampling period of 10ms, 500 tuples are recorded with each launch. For each victim app, we collect 200 launch samples.

[1]The detailed design of the network is available from GitHub [2].

6.3.2 **Time-Series Analysis of App Launches.** The identification of different victim app launches is based on distinct and reproducible time-series characteristics of completed tasks every 10ms during contention. For example, Figure 4(a) shows the time series for the 50th, 100th, and 150th Facebook launch attempts, illustrating that, despite these being separate launch attempts, these attempts have very consistent time-series. In contrast, Figure 4(b) presents the average time-series of the number of completed CPU tasks for three different popular apps, namely YouTube, Google Translate, and Instagram. They show significant distinctive characteristics, making them identifiable through app launch fingerprinting attacks.



(a) Facebook launches



(b) Different apps launches

Figure 4: The characteristics of time-series of number of completed tasks in contention; (a) shows three launch attempts on Facebook and (b) shows launch attempts from three popular apps.

6.3.3 **Classification Accuracy.** We collect 200 launches per victim app in multi-window mode. To verify whether our newly discovered resource contention side-channel is effective in background apps, we also collect data of the same scale in foreground-background mode, i.e., the adversary runs in background and the victim runs in foreground.

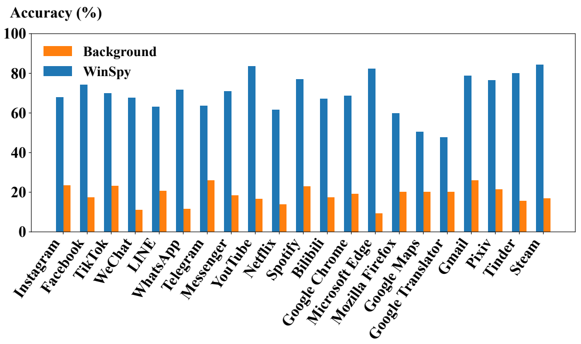


Figure 5: The comparison between the accuracy of *WinSpy* and the foreground-background mode for app launch fingerprinting attacks.

The classification accuracy for each of the 21 apps is presented in Figure 5. We can observe that *WinSpy* in multi-window mode achieves an average accuracy of 70.5%, while the accuracy of the adversary app in background mode is only 20.1%. Moreover, *WinSpy* is consistently and significantly more accurate in multi-window mode compared to adversary in foreground-background mode.

6.4 Website Launch Fingerprinting

In this section, we will conduct experiments and evaluations on the accuracy of *WinSpy* in website launch fingerprinting attacks.

We choose 21 popular websites from the similarweb's [3] hotlist, which provides a list of global website rankings based on the traffic and visitation data of various websites, the complete list is shown in Figure 6.

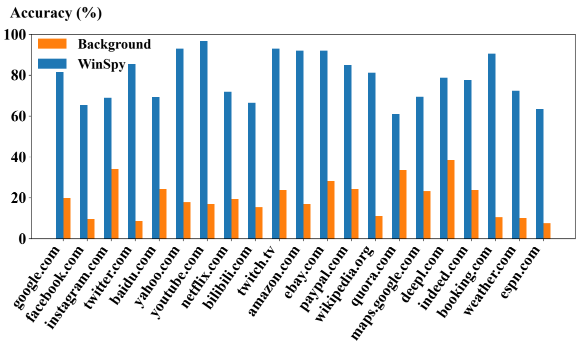
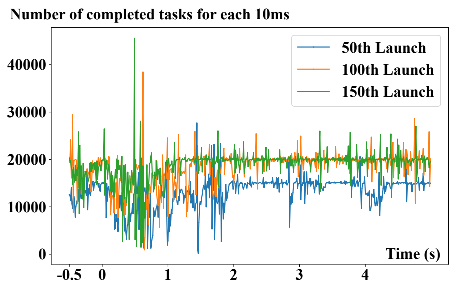


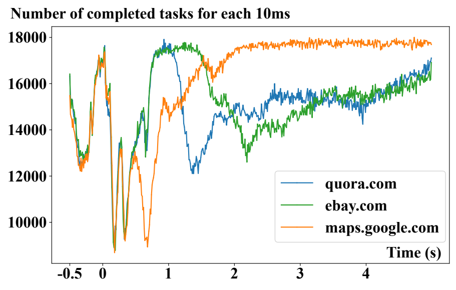
Figure 6: The comparison between the accuracy of *WinSpy* and the foreground-background mode for website launch fingerprinting.

The accuracy of website launch fingerprinting attacks of *WinSpy* and its contending method, i.e., adversary in background, on 21 different websites are shown in Figure 6. From the evaluation results, we can clearly observe the large accuracy gap between our proposed method and its counterpart for 21 websites. *WinSpy* achieves an average accuracy on website launch fingerprinting attacks about 80.3% while when the adversary runs in background, its average attack accuracy is only about 22.7%.

In the evaluations on the accuracy of website launch attacks, we collect website launch data samples from the adversary app when the victim app, Microsoft Edge, launches different websites. To demonstrate the superior performance of *WinSpy* in website launch fingerprinting attacks, the two apps run in either multi-window or foreground-background modes during data collection experiments. For each mode, we collect 200 data samples for each website.



(a) amazon.com launches



(b) Different websites launches

Figure 7: The characteristics of time-series of number of completed tasks in contention; (a) shows three launch attempts on amazon.com and (b) shows launch attempts from three popular websites.

To provide an intuitive explanation of why *WinSpy* achieves promising accuracy in website fingerprinting attacks, Figure 7 presents examples of time-series from monitoring the resources contention through *WinSpy* to show the characteristics caused by launching different websites in the victim app. By comparing the time-series from the 50th, 100th, and 150th launch attempts on *Amazon.com* (Figure 7(a)), we can observe significant consistency among these launch attempts, indicating that the characteristics of launching the same website are reproducible. On the contrary, the average time-series generated from launching three different websites, i.e., *quora.com*, *ebay.com*, and *maps.google.com*, in Figure 7(b) demonstrate significantly distinctive characteristics.

6.5 In-app Activities Fingerprinting

Finally, we conduct extensive experiments and evaluations to investigate the potential of side-channel attacks based on resource contention for inferring more fine-grained and sensitive in-app activities, result in significant privacy leakage for some specific apps.

6.5.1 **Apps and In-app Activities Selection.** For in-app activities fingerprinting attacks, we choose *WhatsApp*, a widely-used instant messaging and calling app with over 2 billion monthly active users [53], and *Alipay*, an online payment platform with over 650 million monthly active users [52], as two representative apps. These apps involve sensitive interactions with users and have access to private data. Therefore, successful attacks on the in-app activities of these apps pose significant privacy and financial risks.

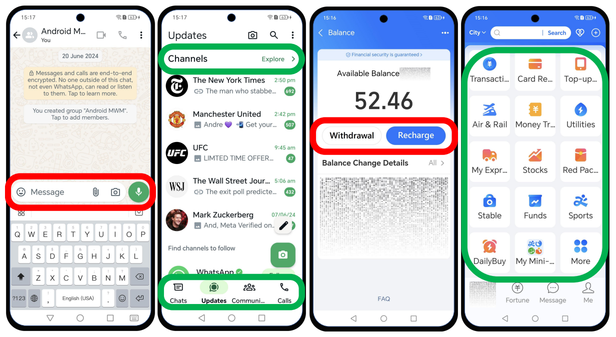


Figure 8: The critical and non-critical in-app activities in *WhatsApp* (left) and *Alipay* (right) apps.

Specifically, for *WhatsApp*, as shown in the region within the red rectangle on the leftmost side of Figure 8, the attacks focus on three critical and frequent in-app activities: first, **Tap Keyboard** is the user taps the input box to bring up the on-screen keyboard for sending text messages. Second, **Tap Microphone** is the user taps the microphone button to start recording a voice message. Third, **Tap Camera** is the user taps the camera button to trigger the camera or access the photo library for sharing a photo.

Alongside these critical activities, multiple non-critical actions are also included in the data collection to form a comprehensive dataset for evaluating the accuracy of identifying one of the critical in-app activities. As shown in the regions of the green rectangles in the left side of Figure 8, these non-critical actions include navigating between different tabs (the green rectangle at the bottom) and viewing channels content (the green rectangle at the top).

For *Alipay*, the considered critical in-app activities are shown within the red rectangle in the right side of Figure 8. These activities are financial-related actions, specifically recharging and withdrawing funds: first, **Tap Recharge** is the user is going to add some funds from his/her bank account to his/her Alipay account by tapping the recharge button. Second, **Tap Withdraw** is the user is going to withdraw money from his/her Alipay account to his/her bank account.

The above in-app activities are critical as they are often followed by password input, which can be inferred from the IMU-based tap location attacks launched by *WinSpy* (described in the next section). Similar to the case for *WhatsApp*, a number of non-critical in-app activities are also included in the data collection experiment, such as checking the account balance, viewing transaction details, etc., by clicking the corresponding buttons within the region framed by the green rectangle in the rightmost part of Figure 8.

6.5.2 **Evaluation Results.** To evaluate the accuracy of the in-app activities fingerprinting attacks, we collect an in-app activities dataset when the pair of adversary and victim apps run in either multi-window mode (i.e., *WinSpy*) or foreground-background mode. For each critical in-app activity, 1,000 tapping actions are performed. For the non-critical activities, a total of 5,000 tapping actions are collected. During data collection, the adversary app, either running in multi-window mode or background mode, records the time series of the resource contention.

Table 2: Accuracy of in-app activities fingerprinting attacks on *WhatsApp* and *Alipay* launched by *WinSpy* and adversary runs in background.

Activities in WhatApp	WinSpy	Background
Tap Keyboard	97.4%	71.9%
Tap Microphone	98.2%	73.4%
Tap Camera	96.6%	68.6%
Average	**97.4%**	**71.3%**

Activities in Alipay	WinSpy	Background
Tap Withdraw	97.2%	78.2%
Tap Recharge	98.1%	77.3%
Average	**97.7%**	**77.8%**

Table 2 presents the evaluation results on inferring the critical in-app activities of *WhatsApp* and *Alipay* via either *WinSpy* or the adversary app running in background mode. By comparing the accuracy of different methods, we observe that *WinSpy* achieves high accuracy in inferring all the critical activities. This indicates that the newly discovered resource contention side-channel is highly effective, posing significant risks of privacy leakage concerning users' private and sensitive interactions with victim apps in multi-window mode and even in foreground-background mode.

6.6 Extended Discussion

6.6.1 **Generalization.** The evaluations above primarily focus on a large-scale dataset collected from a foldable smartphone. To demonstrate the generalization of *WinSpy* in side-channel attacks via resource contention, we collected another dataset using three Android devices: the Honor Magic V2 (foldable), Huawei P30, and a Xiaomi Pad 6 tablet. During data collection, we considered all three multi-window modes—floating-window, split-screen, and picture-in-picture in addition to a background mode. This dataset centers on website launch fingerprinting attacks targeting seven popular and representative websites from Similarweb's [3] hotlist, including *google.com*, *instagram.com*, *facebook.com*, *twitter.com*, *baidu.com*, *youtube.com*, and *netflix.com*. For the adversary app running on the three devices in four different modes, we collected 200 samples for each website.

Table 3: Website launch fingerprinting attacks on different smartphones and tablet in different modes.

Device	Floating	Split	PiP	Background
Magic V2	94.8%	95.5%	90.3%	42.3%
P30	92.7%	91.2%	90.1%	31.1%
Xiaomi Pad 6	89.9%	87.4%	88.0%	22.3%

As the evaluation results shown in Table 3, *WinSpy* consistently achieves high accuracies on different devices and in different multi-window modes. The accuracy is at least over 88%. Moreover, comparing the results in different modes reveals significant accuracy gaps between *WinSpy* and the adversary running in the background, with differences reaching up to 60%.

6.6.2 **Multi-window Mode v.s. Background Mode for Resource Contention.** The evaluations in this whole section demonstrate that *WinSpy* consistently achieves high accuracy in each attack, leveraging our newly discovered resource contention side-channel in multi-window mode. However, the performance is poor when this side-channel is used with background apps. This can be attributed to several factors:

First and foremost, multi-window and background modes differ significantly in their operating mechanisms, resulting in varied resource allocations to the adversary app. In multi-window mode, the adversary app receives sufficient resources to complete over 20,000 CPU tasks within 10ms. In contrast, in background mode, it is limited to only 1,000 to 4,000 CPU tasks. Figure 9 presents the time-series of resource contention generated by the 100th website launch on *twitter.com* in multi-window mode and background mode, respectively, highlighting the reduced contention in background mode. This restriction hampers the background app's ability to effectively contend with foreground apps, making it challenging to extract unique and consistent time-series characteristics from the resource contention side-channel.

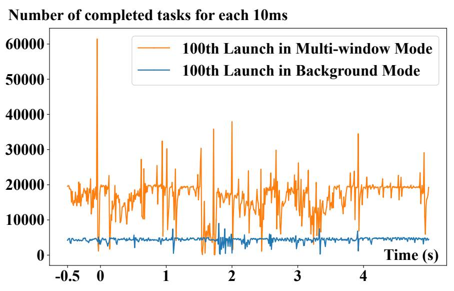


Figure 9: The 100th launch attempt on twitter in multi-window mode and background mode.

Second, as discussed earlier, apps in background mode are assigned significantly lower priority than those in multi-window mode. Consequently, the stability of the adversary app in the background cannot be guaranteed, and its processes are more likely to be terminated by the system when resources are insufficient. This instability makes it difficult to reproduce the characteristics in the time-series.

7 Design and Evaluation on Attacks based on Shared Sensor Readings

In this section, we investigate side-channel attacks conducted by *WinSpy* using shared sensor readings, specifically IMU time-series, when the adversary and victim apps run concurrently in multi-window mode. We will first introduce the design of the corresponding adversary app. Then, we will conduct experiments and extensive evaluations on IMU-based tap location inference attack. These evaluations will demonstrate the accuracy of IMU-based side-channel attacks in multi-window mode.

7.1 Design of Adversary App

The IMU-based tap inference attacks aim to localize the tap location of the user outside the adversary app to infer sensitive information, such as the content input into the victim app via the touch keyboard. To achieve this goal, the adversary app registers with the system to access IMU sensors using `SENSOR_DELAY_FASTEST` [10], capturing data from the accelerometer and gyroscope. In multi-window mode, this is considered a normal permission [6], allowing the app to read IMU data at maximum rates without user consent, while background apps cannot access IMU data at all without explicit permission request. The highest sampling rates vary for different smartphones, for example, 425Hz on the Honor device and 500Hz on the P30, with each cycle yielding values from three axes.

7.2 IMU-based Tap Position Inference

7.2.1 **Experiment Design and Data Collection.** To investigate the IMU-based tap inference attacks conducted by *WinSpy*, we design two series of data collection experiments. The first experiment captures the IMU time-series data when the user taps on various locations across the screen, aiming to assess the overall localization accuracy. The second experiment records the IMU time-series data while the user types on a numeric keypad, with the goal of classifying the digits 0-9 based on the IMU time-series data induced by each tap.

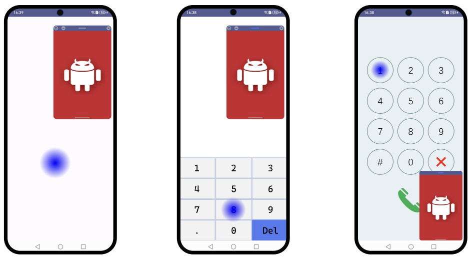


Figure 10: The experiment settings for IMU-based tap inference. *Left:* the tapping localization. *Middle:* the digits classification with a keypad at the bottom. *Right:* the digits classification with a keypad in the middle.

In the experiments, two Android devices are used: the Honor Magic V2 and Huawei P30. The adversary app runs as a floating window in multi-window mode alongside the victim app. For the tap localization data collection, a blue dot appears randomly on the screen for three seconds to guide the user to press on it, as shown in the leftmost part of Figure 10. This ensures that the ground truth locations are recorded along with the corresponding IMU time-series data captured by the adversary, supporting subsequent evaluations. For the digits classification data collection, we consider two types of numeric keypad layouts, as shown in the middle and right parts of Figure 10: the numeric keypad located at the bottom or in the middle of the screen. During data collection, a blue dot appears randomly in the middle of a digit area for three seconds to guide the user to press it, allowing for convenient acquisition of ground truth for evaluations.

We collected datasets using different devices in various folding statuses (Magic V2 in both folded and unfolded modes). In each scenario, approximately 3,000 samples of time-series data caused by tapping were collected. A 1D-CNN is adopted as the backbone for tap localization and digit classification. The difference lies in the usage: the 1D-CNN is used for regression in tap localization, while it functions as a classifier for digit classification.

7.2.2 **Evaluation Results of Tap Localization.** Table 4 presents the pixel mean absolute error (PMAE) of the tap localization in Euclidean distance ($\sqrt{dx^2 + dy^2}$), and its projection on the horizontal ($|dx|$) and vertical ($|dy|$) directions, marked in bold font. To make the results more intuitive, the table also includes the width, height, and diagonal length in pixels for reference. To emphasize the necessity of using the fast sampling rate (i.e., in `SENSOR_DELAY_FASTEST` mode), we also collected a subset of data at a medium sampling rate (`SENSOR_DELAY_GAME`) [11], which is approx. 50Hz.

Table 4: PMAE for tap localization.

PMAE (pixels)	Magic V2 (folded)	Magic V2 (unfolded)	P30
$\lvert dx\rvert$	**118.7**/1060	**122.2**/2156	**77.1**/1080
$\lvert dy\rvert$	**179.3**/2376	**133.3**/2344	**119.7**/2340
$\sqrt{dx^2 + dy^2}$	**233.9**/2602	**197.0**/3185	**154.0**/2577
$\sqrt{dx^2 + dy^2}$ (50hz)	**534.6**/2602	**492.2**/3185	**451.1**/2577

From the results, we observe that *WinSpy* achieves reasonable tap localization accuracy with the fast sampling rate. For example, by comparing the PMAEs to the length of each dimension, the relative errors (PMAE against the width, height, or diagonal length) range from 5% to 11.1%. In contrast, the localization accuracy at the 50Hz sampling rate is significantly worse, with errors more than twice as large as those at the fast sampling rate.

To gain more insights into the accuracy of tap localization attacks, we present the distributions of PMAE values and their corresponding spatial distribution on the screen in Figure 11, using results from the Huawei P30 as an example. From the left part of the figure, we observe that 80% of the localization results have PMAEs within 166.03 pixels, which accounts for 6.4% of the diagonal length or about 1.2 cm according to the screen size of the P30.

The right part of the figure shows that tap localization attacks achieve slightly smaller localization error (lower

PMAEs) in the area along the secondary diagonal of the screen compared to the peripheral areas. This is likely because when users hold the phone, there is less hand support in this area, making it more susceptible to acceleration.

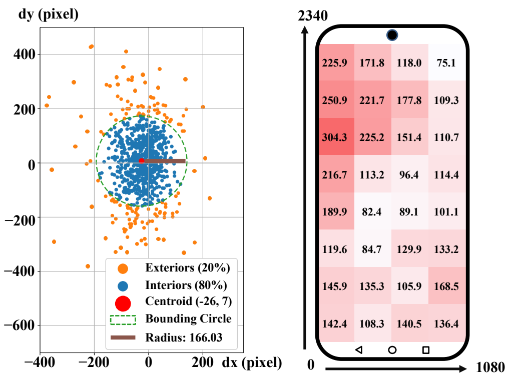


Figure 11: Left: The distribution of relative coordinates (dx, dy) between predicted and actual positions in P30. Right: The average relative distance in each area.

Table 5: The accuracy of digits classification via IMU-based tap inference attacks.

Scenario	Layout type	Accuracy
Magic V2 (folded)	keypad-bottom	40.1%
Magic V2 (folded)	keypad-middle	81.7%
Magic V2 (unfolded)	keypad-bottom	74.7%
Magic V2 (unfolded)	keypad-middle	94.4%
P30	keypad-bottom	80.0%
P30	keypad-middle	91.8%

7.2.3 **Evaluation Results of Digits Classification.** Table 5 presents the accuracy of digit classification via IMU-based tap inference attacks launched by *WinSpy* on different devices, in various folding states (folded or unfolded), and with different numeric keypad layouts (bottom or middle). The results indicate that IMU-based tap inference can achieve relatively high accuracy in certain scenarios. Specifically, tapping on a numeric keypad located in the middle of the screen poses a higher risk compared to scenarios where the keypad is at the bottom. The average attack accuracy for keypads in the middle is 89.3%, while for keypads at the bottom, it is only 64.9%, representing a 24.4% improvement. These deviations may be caused by the fact that the IMUs are located in the middle of the screen, making the sensors more sensitive to tapping in the middle area. Therefore, the location of the sensors significantly impacts the effectiveness of IMU-based attacks.

It is worth noting that, since users inevitably interact with the attack app through tap interactions , the attacker app can always collect some training ground truth data that is outside of the training set, which ultimately benefits the attacker. Moreover, there is a certain degree of consistency in input features across different users, as many papers have experimentally verified [16, 17, 21, 61]. For instance, as shown in [16], when targeting 4-digit PIN passwords, the top-5 accuracy before and after cross-user adaptation reached 43% and 30% respectively.

8 Discussions on Mitigation and Limitation

8.1 Expanding the Discovery

The resource contention side-channel essentially involves equal contenders inadvertently revealing information about themselves while competing for limited resources. Beyond the attacks discussed and evaluated in this paper, this side-channel can also be applied to other contention scenarios in Android, such as competing for network bandwidth or GPU processing power. It is also applicable to similar contention scenarios in other systems like iOS, Windows, and Linux.

The shared sensor reading side-channel arises from the lax permission management for windowed apps on Android, which allows them to read IMU data at the maximum rate. This side-channel can provide even more detailed information. By applying the method described in [35], we found that in multi-window mode, an adversary app could recover the sound output of another app via the IMU, achieving an average Mel-Cepstral Distortion (MCD) value of 6.85 dB (an MCD value less than 8 is considered effective [57]).

We have omitted the detailed evaluation results of these extended experiments due to space constraints, but they are available in the corresponding open-source repository.

8.2 Potential Mitigations

Our IMU-based attack primarily exploits the system's lax control over sensor access in multi-window modes. An adaptive sensor access management approach could be implemented to counteract side-channel attacks based on shared sensor readings. For instance, in multi-window mode, apps not in focus could be restricted to accessing IMU sensors at a lower sampling rate. This would allow basic functionalities reliant on IMU sensor readings to continue working while significantly degrading the accuracy of attacks.

As for the proposed contention side-channel, it primarily exploits the simplistic priority management in multi-window modes, where apps operate at similar priority levels to enhance the multitasking user experience. Below, we provide mitigations at the system and app levels.

8.2.1 System-Level Mitigations.

At the system level, mitigations can be made in areas such as scheduling mechanisms, detection methods, and resource allocation.

Adaptive Priority Management Mechanism. The resource allocation priority of multi-window apps can be dynamically adjusted based on user interaction focus. By assigning higher scheduling weights to the focused apps, the resource contention intensity of non-focused apps can be reduced, thereby weakening the ability of attackers to capture resource usage fingerprints.

Randomized Resource Allocation Strategy. Controlled random perturbations can be introduced during scheduling, such as minor fluctuations in CPU time slice allocation, to destabilize the temporal patterns of resource contention.

Coarse-Grained Resource Allocation. Non-uniform, coarse-grained resource allocation, such as batch CPU time slice allocation in 50ms intervals, can be adopted to minimize the leakage of fine-grained resource usage characteristics.

Behavior-Based Attack Detection. The resource usage patterns of multi-window apps, such as CPU utilization and memory access frequency, can be monitored. Machine learning models can then be deployed to identify abnormal contention behaviors, such as persistent high resource consumption without valid business logic, and proactively isolate or restrict resource access for suspicious apps.

Resource Isolation and Pre-allocation: before an app executes high-risk tasks, the system can pre-allocate necessary resources, such as CPU, memory, and etc..

8.2.2 Application-Level Mitigations.

At the app level, mitigations require the app to redesign the execution process of key tasks.

Dynamic Noise Injection: injecting lightweight random computational tasks (e.g., simple CPU calculations or memory access) during critical operations of victim apps can obscure execution signatures of important tasks.

Resource-Agnostic Task Design: sensitive operation workflows can be redesigned to decouple execution time from CPU resource allocation. By implementing constant-time algorithms, such as the timing-attack-resistant design in AES encryption, the leakage of operational details through resource competition-induced temporal variations can be prevented at a more fundamental level.

8.3 Limitations

To the best of our knowledge, we are the first to work on implementing app-level side-channel attacks to multi-window mode. While promising accuracies have been achieved for several side-channel attacks investigated in this paper, our work still has some limitations.

Multi-window with multi-apps. Although the introduction of multi-window mode is primarily intended to enable two apps to run efficiently simultaneously, such as watching a video in a floating window while browsing the web or using split-screen mode to simultaneously edit a document and check emails, it also supports the parallel operation of more than two apps. This introduces new challenges in fingerprinting based on resource contention, which we leave as a future direction of exploration.

Performance and stealth. Our attack involves significant CPU utilization, which may lead to performance issues on mobile devices, such as slowed applications, overheating, or rapid battery drain—symptoms that could potentially alert users. However, in most off-the-shelf smartphones, the CPU's energy consumption is relatively low, whereas components such as the display and baseband modules contribute significantly to overall power usage. Therefore, the attack could still be covert on such devices.

Variable positioning of interactive elements. From the evaluation results on IMU-based digit classification, we find that the simple 1D-CNN classifier cannot accurately classify digits when the numerical keypad is located at the bottom of the phone, compared to when the keypad is in the middle. However, for most apps, the keypad is normally located at the bottom. This issue could be addressed by designing more sophisticated signal processing and classification methods that better capture the characteristics within the typing-induced IMU time-series than a simple 1D-CNN, which is not the focus of this paper.

9 CONCLUSION

In this paper, we investigate significant security and privacy risks introduced by multi-window mode and find several vulnerabilities in this mode, revealing high-risk side channels, leading to potential breaches of user privacy. We develop *WinSpy*, a cross-window side-channel attack framework exploiting resource contention and shared sensor readings in multi-window environments. Extensive evaluations demonstrated that *WinSpy* effectively performs app launch, website launch, and in-app activity fingerprinting. Additionally, it leverages IMU sensors to accurately detect and localize user tap events, allowing for the inference of sensitive interactions within victim apps by tracking subtle motion patterns. Our findings reveal that *WinSpy* achieves higher accuracy in side-channel attacks within multi-window mode, primarily due to simplified resource contention management and shared permissions among apps in this mode. These results underscore the urgent need for enhanced security measures in multi-window mode, including more sophisticated priority management and stricter permission controls.

References

[1] APKPure.com 2024. *APKPure*. APKPure.com. https://apkpure.com/app-24h Accessed on 2024-09-05.

[2] 2024. The source code of WinSpy and additional evaluation results. https://github.com/multiwindowmode/WinSpy.

[3] 2024. *Top Websites - SimilarWeb*. https://www.similarweb.com/top-websites/ Accessed on 2024-09-05.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[5] Abeer AlJarrah and Mohamed Shehab. 2016. Maintaining user interface integrity on Android. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 449–458.

[6] Android. [n. d.]. Android Normal Permission. https://developer.android.com/guide/topics/permissions/overview#normal. Accessed: 2024-09-05.

[7] Android. 2024. *Android Multi-Resume*. Android. https://source.android.com/docs/core/display/multi_display/multi-resume

[8] Android. 2024. *Market share of mobile operating systems worldwide from 2009 to 2024, by quarter*. Android. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[9] Android. 2024. *Multi-window support*. Android. https://developer.android.com/guide/topics/large-screens/multi-window-support

[10] Android. 2024. Sensors Rate Limiting. Android Developers. https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview#sensors-rate-limiting Accessed: 09-03, 2024.

[11] Android. 2024. Sensors Rate Limiting Parameter. Android Developers. https://developer.android.com/reference/android/hardware/SensorManager#SENSOR_DELAY_GAME Accessed: 09-05, 2024.

[12] Android Developers. 2024. Android 7.0 for Developers. https://developer.android.com/about/versions/nougat/android-7.0 Accessed: 2024-09-05.

[13] Android Developers. 2024. App Sandbox. Available online. https://source.android.com/docs/security/app-sandbox Accessed: 2024-09-05.

[14] Android Developers. 2024. Processes and Threads. Available online. https://developer.android.com/guide/components/processes-and-threads Accessed: 2024-09-05.

[15] AppBrain. 2023. The most common Android OS versions currently installed on Android devices (phones and tablets) used by AppBrain SDK users. https://www.appbrain.com/stats/top-android-sdk-versions

[16] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. 2012. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th annual computer security applications conference*. 41–50.

[17] Adam J Aviv and Jonathan M Smith. 2012. *Side channels enabled by smartphone interaction*. Ph. D. Dissertation. Ph. D. dissertation, Pennsylvania State University.

[18] Zhongjie Ba, Tianhang Zheng, Xinyu Zhang, Zhan Qin, Baochun Li, Xue Liu, and Kui Ren. 2020. Learning-based Practical Smartphone Eavesdropping with Built-in Accelerometer.. In *NDSS*, Vol. 2020. 1–18.

[19] Daniel Bader. 2018. *Multi-Window on phones is the best Android feature you're probably not using*. https://www.androidcentral.com/multi-window-phones-best-android-features Accessed: 2024-09-05.

[20] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 931–948.

[21] Liang Cai and Hao Chen. 2011. {TouchLogger}: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*.

[22] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2014. Peeking into your app without actually seeing it:{UI} state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*. 1037–1052.

[23] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. 2017. POWERFUL: Mobile app fingerprinting via power analysis. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[24] Codiant. 2023. *Foldable Devices and App Development: Glimpse of Future 2023*. https://codiant.com/blog/foldable-devices-and-app-development-in-future/ Accessed: 2024-09-05.

[25] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An exploration of ARM system-level cache and GPU side channels. In *Proceedings of the 37th Annual Computer Security Applications Conference*. 784–795.

[26] Android Developers. 2024. Performance Overview. https://developer.android.com/topic/performance/overview. Accessed: 2024-09-05.

[27] Android Developers. 2024. Performance Threads. https://developer.android.com/topic/performance/threads. Accessed: 2024-09-05.

[28] Android Developers. 2024. Processes and Threads. https://developer.android.com/guide/components/processes-and-threads. Accessed: 2024-09-05.

[29] Android Developers. 2024. Sensors Overview. https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview#only-gather-sensor-data-in-the-foreground. Accessed: 2024-09-05.

[30] Debopriya Roy Dipta and Berk Gulmezoglu. 2022. Df-sca: Dynamic frequency side channel attacks are practical. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 841–853.

[31] Rubens Eishima. 2022. *Multi-tasking on Android: How to use split-screen mode*. https://www.nextpit.com/how-to-split-screen-android-devices Accessed: 2024-09-05.

[32] Marvin Gabriel. 2023. *HUAWEI Mate X3: The Best Foldable Smartphone There Is*. https://techbroll.com/2023/05/huawei-mate-x3-the-best-foldable-smartphone-there-is.html Accessed: 2024-09-05.

[33] Google. 2024. *Android Developers - Activity Lifecycle*. https://developer.android.com/guide/components/activities/process-lifecycle

[34] Chenkai Guo, Tianhong Wang, Qianlu Wang, Naipeng Dong, Xiangyang Luo, and Zheli Liu. 2025. Fratricide! Hijacking in Android Multi-window. *IEEE Transactions on Dependable and Secure Computing* (2025).

[35] Pengfei Hu, Hui Zhuang, Panneer Selvam Santhalingam, Riccardo Spolaor, Parth Pathak, Guoming Zhang, and Xiuzhen Cheng. 2022. Accear: Accelerometer acoustic eavesdropping with unconstrained vocabulary. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1757–1773.

[36] Alexander S La Cour, Khurram K Afridi, and G Edward Suh. 2021. Wireless charging power side-channel attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 651–665.

[37] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 398–409.

[38] Anindya Maiti, Oscar Armbruster, Murtuza Jadliwala, and Jibo He. 2016. Smartwatch-based keystroke inference attacks and context-aware protection mechanisms. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 795–806.

[39] Anindya Maiti, Murtuza Jadliwala, Jibo He, and Igor Bilogrevic. 2018. Side-channel inference attacks on mobile keypads using smartwatches. *IEEE Transactions on Mobile Computing* 17, 9 (2018), 2180–2194.

[40] Björn Mathis, Vitalii Avdiienko, Ezekiel O Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 263–273.

[41] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. 2019. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*. 135–149.

[42] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 323–336.

[43] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2139–2153.

[44] Gadgets Now. 2020. *How to use the multi-window feature on Android smartphones*. https://www.gadgetsnow.com/how-to/how-to-use-the-multi-window-feature-on-android-smartphones/articleshow/77200949.cms Accessed: 2024-09-05.

[45] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. Accessory: password inference using accelerometers on smartphones. In *proceedings of the twelfth workshop on mobile computing systems & applications*. 1–6.

[46] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. 645–662.

[47] Gerald Palfinger, Bernd Prünster, and Dominik Julian Ziegler. 2020. Androtime: Identifying timing side channels in the android api. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1849–1856.

[48] Tse-Yu Pan, Chih-Hsuan Kuo, Hou-Tim Liu, and Min-Chun Hu. 2018. Handwriting trajectory reconstruction using low-cost imu. *IEEE Transactions on Emerging Topics in Computational Intelligence* 3, 3 (2018), 261–270.

[49] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android.. In *NDSS*.

[50] Shirish Singh, Devu Manikantan Shila, and Gail Kaiser. 2019. Side channel attack on smartphone sensors to infer gender of the user. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 436–437.

[51] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 749–763.

[52] statistics. 2024. *Monthly active users of Alipay*. statistics. https://www.statista.com/statistics/1395691/china-alipay-monthly-active-users/

[53] statistics. 2024. *Most popular global mobile messenger apps*. statistics. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

[54] Xianbo Wang, Yikang Chen, Ronghai Yang, Shangcheng Shi, and Wing Cheong Lau. 2020. Fingerprint-jacking: Practical fingerprint authorization hijacking in Android apps. *Blackhat, Europe, Tech. Rep. Blackhat* 2020 (2020).

[55] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 125–137.

[56] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. 113–124.

[57] Chen Yan, Guoming Zhang, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. 2019. The feasibility of injecting inaudible voice commands to voice assistants. *IEEE Transactions on Dependable and Secure Computing* 18, 3 (2019), 1108–1124.

[58] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. 2015. A study on power side channels on mobile devices. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. 30–38.

[59] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S Balagani. 2016. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Transactions on Information Forensics and Security* 12, 5 (2016), 1056–1066.

[60] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and defence on android free floating windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 759–770.

[61] Cheng Zhang, Anhong Guo, Dingtian Zhang, Yang Li, Caleb Southern, Rosa I Arriaga, and Gregory D Abowd. 2016. Beyond the touchscreen: an exploration of extending interactions on commodity smartphones. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 6, 2 (2016), 1–23.

[62] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems.. In *USENIX Security Symposium*, Vol. 20. 23.

[63] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. 2015. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 915–930.

[64] Yicheng Zhang, Carter Slocum, Jiasi Chen, and Nael Abu-Ghazaleh. 2023. It's all in your head (set): Side-channel attacks on {AR/VR} systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3979–3996.

[65] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. 2013. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1017–1028.