PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android

Xianbo Wang, Shangcheng Shi, Yikang Chen, Wing Cheong Lau
The Chinese University of Hong Kong
{xianbo, ss016, ykchen, wclau}@ie.cuhk.edu.hk

Abstract—**Nowadays, most mobile devices are equipped with various hardware interfaces such as touchscreen, fingerprint scanner, camera and microphone to capture inputs from the user. Many mobile apps use these physical interfaces to receive user-input for authentication/authorization operations including one-click login, fingerprint-based payment approval, and face/voice unlocking. In this paper, we investigate the so-called PHYjacking attack where a victim is misled by a zero-permission malicious app to feed physical inputs to different hardware interfaces on a mobile device to result in unintended authorization. We analyze the protection mechanisms in Android for different types of physical input interfaces and introduce new techniques to bypass them. Specifically, we identify weaknesses in the existing protection schemes for the related system APIs and observe common pitfalls when apps implement physical-input-based authorization. Worse still, we discover a race-condition bug in Android that can be exploited even when app-based mitigations are properly implemented. Based on these findings, we introduce fingerprint-jacking and facejacking techniques and demonstrate their impact on real apps. We also discuss the feasibility of launching similar attacks against NFC and microphone inputs, as well as effective tapjacking attacks against Single Sign-On apps. We have designed a static analyzer to examine 3000+ real-world apps and find 44% of them contain PHYjacking-related implementation flaws. We demonstrate the practicality and potential impact of PHYjacking via proof-of-concept implementations which enable unauthorized money transfer on a payment app with over 800 million users, user-privacy leak from a social media app with over 400 million users and escalating app permissions in Android 11.**

I. Introduction

Many authorization transactions in mobile devices require confirmation with human inputs via some hardware interface. With the proliferation of mobile devices equipped with a rich set of hardware sensors, there is a widespread adoption of authorization services based on different physical inputs, *e.g.*, screen-tapping, fingerprint scanning and face recognition. In this paper, we investigate authorization hijacking attacks when different physical inputs are involved. A central issue of such hijacking attacks is the clarity of context during authorization: users need explicit notification about *which app* is requesting the authorization for *what data or action* before making the decision. Any confusion can potentially create security threats during the authorization process. Thus, attackers may deliberately create confusion and make the victim authorize something unwanted to realize authorization hijacking. As users tend to retrieve the authorization context based on visual contents, User Interface (UI) attack is an important, but not the sole, component for authorization hijacking. By combining new found attack vectors on physical input APIs with the manipulation of on-screen content to hide or obscure the context, it is possible to lure the victim to feed physical inputs to different hardware interfaces on mobile devices and result in unintended authorization. We coin the term PHYjacking for such attacks. In this work, we study and demonstrate the feasibility of PHYjacking on Android devices by analyzing and answering the following research questions:

Q1 Are there practical techniques for attackers to hide or manipulate the authorization context?
Q2 Do mobile operating systems (OS) continue to accept the target physical input during authorization even when the context information is not visible to the user?
Q3 Do real-world apps perform checking during authorization to ensure proper context information is delivered to the user?

There are many possible situations for the user to miss the context information during authorization, including the cases when the OS or app does not provide enough information, or the user ignores the information, or the context information is covered by some other window. In the last case, as shown in Fig. 1, the OS or app should immediately block the physical input to avoid unintentional authorization. Otherwise, a malicious app can create a delusive covering above the benign authentication window to hijack the physical input. One realistic attack scenario: a malicious app displays a fake unlock screen to trick the user to habitually do a fingerprint scanning without knowing that the fingerprint is actually used by a target app to approve an unintended fund transfer.

After analyzing Android system API and legitimate apps regarding their protection mechanisms on various physical inputs, we discover various system/app-level weaknesses which can be exploited to realize the aforementioned authorization hijacking attacks. Based on these findings, we propose a general PHYjacking framework targeting various physical authorization inputs, including screen touch, fingerprint scanning, face recognition, NFC tag reading, and voice-based unlock. Fig. 2 summarizes the existing as well as the new approaches

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24097
www.ndss-symposium.org

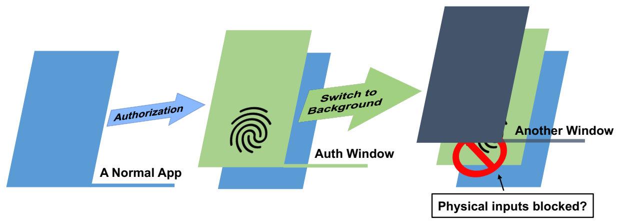


Fig. 1: Possibility of PHYjacking: is the user-input detected by the physical sensors received by its intended application?

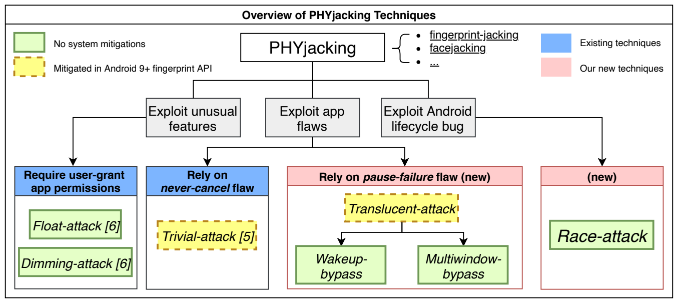


Fig. 2: Overview and taxonomy of PHYjacking techniques.

we proposed to enable PHYjacking. In particular, we introduce new techniques for creating declusive visual coverings, manipulating Android Activity Lifecycle and combining them with other attacks like tapjacking. Our research also reveals several security bugs in the Android system, including i) a flaw in the latest fingerprint hijacking mitigation mechanism, ii) a permission hijacking vulnerability in a security essential system setting, and iii) a race-condition bug in the Android Activity Lifecycle. Our proposed PHYjacking attacks are practical and effective because: 1) all of them can be launched from a zero-permission malicious app ; 2) they can bypass various system-level countermeasures introduced by recent versions of Android and 3) they are able to exploit widespread app implementation flaws when using physical-input APIs. We demonstrate the feasibility and potential impact of PHYjacking by implementing proof-of-concept (PoC) attacks [1] for popular real-world apps across various Android versions. We show that with a zero-permission malicious app, the attacker can cause damage in form of financial loss, user privacy leaks, and app privilege escalation. To summarize, this paper has made the following technical contributions:

- We propose and investigate PHYjacking, a new class of authorization hijacking attacks which mis-directs security sensitive user-inputs from different kinds of physical sensors in a smartphone or tablet.
- We identify new security issues in Android, including a mitigation weakness, a race-condition attack that can break the Android Activity life cycle model, and a crucial permission setting that is vulnerable to hijacking.
- We demonstrate the practicality and critical impact of PHYjacking via proof-of-concept attacks against prominent mobile apps.
- We design a static code analyzer and use it to discover that a significant portion of Android apps in the wild contain implementation flaws that are vulnerable to PHYjacking.

Roadmap. We first provide background knowledge in II and then propose the overall PHYjacking framework in III. As two key parts of the framework, IV introduces new confusion setup techniques, and V analyzes flaws in existing protection schemes. VI details concrete attacks for different physical inputs, while VII presents empirical results of our measurement study. We then discuss possible mitigations (VIII) and related work (IX) before concluding the paper (X).

[1]Sample code and video demo of attacks can be found at: https://mobitec.ie.cuhk.edu.hk/phyjacking

II. Background

A. Authorization in Android

There are countless scenarios where authorization is involved in Android. Our study focuses on in-app authorization like payment confirmation or permission grant. We abstract the following concepts from general authorization scenarios to facilitate discussions:

- *Objective*: Every authorization request has an objective, be it acting on behalf of the user (*e.g.*, to confirm fund transfer) or retrieving information (*e.g.*, get the user's profile).
- *Physical input*: Apps rely on various physical inputs from the user for confirming the authorization via button tapping, fingerprint scanning, face recognition or voice unlocking, *etc.*, where different hardware sensors/ interfaces are involved.
- *Entity*: The multiple interacting parties engaged in the authorization process, namely, the *requester*, *provider*, and *user*. The requester requests permission. Once the user approves, the provider can disclose the information or proceed with the action.

With the above definitions, Table I depicts different authorization scenarios and the physical input interfaces involved.

B. Android Security Mechanisms

The security of the authorization process depends on multiple security provisions in Android, at both the software and hardware level. The following discussions focus on the mechanisms which are essential to our study.

a) Permissions. Android framework defines three permission levels for apps, namely, *normal*, *signature* and *dangerous*. Dangerous permissions need to be explicitly granted by the user at runtime, while normal level permissions are granted automatically at app installation time. All permissions need to be declared statically in the Android Manifest file. To access the fingerprint sensor, an app needs to request the `USE_FINGERPRINT` permission, which is a normal level auto-grant permission. Therefore, every app that integrates fingerprint functionalities will declare this permission in its

TABLE I: Examples of Authorization Scenarios on Android

	Objective	Physical input	Input device	Requester	Provider
Fingerprint payment	money transfer	finger scanning	fingerprint sensor	shopping app (*e.g.*, eBay)	mobile wallet (*e.g.*, PayPal)
Single Sign-On	login	button tapping	touchscreen	third-party app (*e.g.*, Airbnb)	social platform (*e.g.*, Facebook)
Permission granting	get permission	switch enabling	touchscreen	Android app	Android system
Face unlock	unlock device	face recognition	camera	Android system	Android system
NFC card payment	money transfer	NFC scanning	NFC sensor	shopping app	mobile wallet

Manifest file. By default, apps have no permission to draw persistent overlays on other apps, except for apps having the `SYSTEM_ALERT_WINDOW` permission. This permission is under the dangerous level and needs explicit granting by the user.

b) Hardware. The permission model for apps can limit their capabilities and restrict access to system resources. However, for apps running in an insecure environment, *e.g.*, rooted devices, the manufacturers need to develop hardware-level protection mechanisms to avoid data leakage. For instance, to prevent attackers with root privileges from accessing sensitive data, manufacturers nowadays ship the chips with compartmentalized hardware modules (*e.g.*, TrustZone for ARM) that can execute code in an isolated domain. With TrustZone, the raw fingerprint data is encrypted and stored securely even when the Android system is fully compromised.

C. Android Activity Lifecycle

Android Activity Lifecycle is one of the core designs of the Android UI model. It forms the foundation of our discussion on new confusion attacks in Section IV. Activities are pages with widgets that are shown to users and are basic components of Android apps, analogous to windows in PC systems. An Activity, whose lifecycle is modeled and managed by the Android framework, can be created, paused, and destroyed. Fig. 3 illustrates the simplified Activity Lifecycle model.

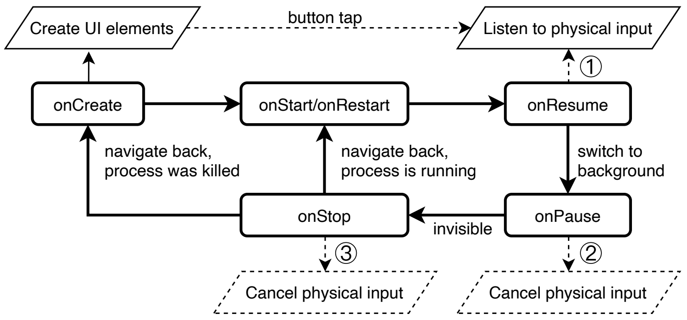


Fig. 3: Simplified diagram of Android Activity Lifecycle and common implementation patterns of the authorization process.

When a user enters a new Activity, the `onCreate` event will be triggered first, followed by the `onStart` and `onResume` events. In usual cases, when an Activity is switched to the background, `onPause` and `onStop` will be triggered sequentially. However, if the background Activity is still visible, *e.g.*, when covered by a translucent Activity, it stays in the paused state instead of being stopped. Finally, when the user brings the background Activity to foreground, the `onRestart` and `onStart` (only if it was stopped), as well as `onResume` events are triggered sequentially.

The Activity lifecycle model is intertwined with the implementation of the different authentication processes. A common practice is to start listening to the physical input (*e.g.*, fingerprint scanning) upon the `onResume` event of the associated Activity and cancel the listener upon the `onPause` or `onStop` event, as shown by ①, ②, ③ respectively in Fig. 3. In this way, the app will start listening to the sensor once the Activity appears and close the sensor once the Activity disappears. However, some apps only start listening to physical input after a button tapping and never cancel the listener.

D. Android API for Dedicated Sensors

While some physical inputs used for authorization are from general sensors, like touchscreen and microphone, some sensors are dedicated to authentication. One typical example of such a sensor on mobile devices is the fingerprint scanner. Although mobile phones are equipped with different vendor-specific fingerprint sensors, Android defines a Hardware Abstraction Layer (HAL) and provides a unified fingerprint API. Android released the first official fingerprint API in Android 6.0 (2015), wrapped in the `FingerprintManager` class. This API was the only official API to support vendor-independent fingerprint scanner access by apps until the rollout of the new `biometricPrompt` API [1] in Android 9. One major difference of the latter API is that it provides a unified UI when prompting users to input their fingerprints. Apart from relieving developers from implementing their own UI, it also reduces the risks of problematic implementation by app developers. However, the `biometricPrompt` API is not backward compatible with devices before Android 9. According to the official Android website [2], until June 2021, more than 50% of devices are still running versions older than Android 9. Additionally, the `biometricPrompt` API is not supported on all vendors' devices. Some devices, *e.g.*, specific Samsung devices, can only use the old API [3]. Therefore, a significant portion of apps in the wild are still using the `FingerprintManager` API. To further mitigate risks and guarantee *What You See is What You Sign* (WYSIWYS), Android 9+ also patched the code of `FingerprintManager` API to add checking-logic to ensure the app occupying the fingerprint scanner is the one running in the foreground.

The vision of `biometricPrompt` API is to unify the acquisition of various biometric modalities, including face, fingerprint, and iris. However, due to the security requirement for hardware, most devices do not have dedicated 3D face sensors to support this API for face recognition. Many third-party SDKs support face recognition using general-purpose camera APIs in Android, which include a deprecated Camera API and a new Camera2 API with more features. There are different ways for apps to fetch camera images. One common approach is to get streaming frames from live preview using the `onPreviewFrame()` callback. Another way is to take a single photo with an API call like `takePicture()`. There are also apps relying on the system camera app to take the photo.

III. The PHYjacking Attack Framework

In this section, we discuss the threat model, settings, and general procedure of PHYjacking attacks.

A. Attacker Capabilities

The physical input hijacking attacks discussed in this paper follow the **malicious app attacker** model. We assume the attacker can install a malicious app on the victim's device, *e.g.*, by uploading the malicious app to either Google Play or some other unofficial app markets and waiting for victims to install it. Note that our malicious app **requires no permission** to be granted by the user explicitly. Its capabilities are the same as other regular apps, which are limited by the Android framework. We also assume that the victim will launch the malicious app at least once. Unless otherwise stated, the attacks we discuss in this paper follow this threat model by default. Our attack does not require system privilege escalation and the victim device need not be rooted.

B. Targets and Scenarios of the Attack

In common physical input hijacking scenarios, there are four primary entities: the victim user, the mobile device, the target app, and the malicious app. The goal of the physical input hijacking is to lure the victim into feeding physical input to his mobile device and unknowingly authorize sensitive actions in the target app. The role of the malicious app in the attack process is only to invoke the authorization process in the target app and layout the delusive interface. It does not try to capture or steal the physical input. The physical input data itself is not what interests attackers. Their ultimate goal is to steal sensitive information or permission, which are usually controlled and owned by the legitimate target app, and can only be accessed or exported after proper authorization. For example, mobile wallets host balance accounts for users. A fingerprint touch received by the malicious app is useless. However, if the attacker can lure the victim into touching the fingerprint sensor when the mobile wallet app is waiting for fingerprint authorization of money transfer, then the attacker can steal money from the victim account.

Depending on the physical input required for authorization, the hijacking target can be different. In this work, we study the possibility of launching hijacking attacks against the following types of physical inputs:

- *Fingerprint scanning.* Most modern mobile devices are equipped with fingerprint sensors, which can be used as a biometric authenticator. Apps may utilize these sensors for content unlock or payment confirmation. Android system provides dedicated APIs for apps to access the fingerprint scanner.

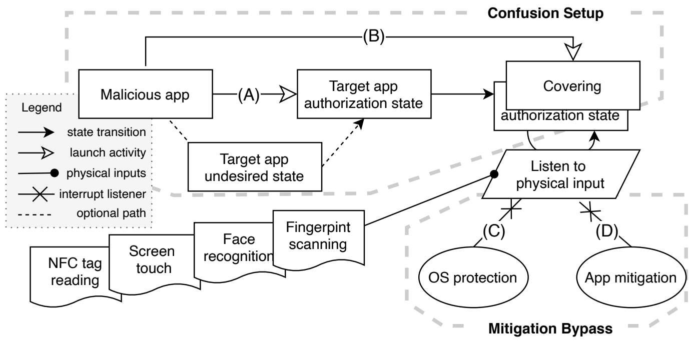


Fig. 4: The general framework of PHYjacking attack. (A) Driving target app to the desired state. (B) Creating malicious covering. Bypassing system (C) and app (D) level protections. (C) and (D) are key challenges to enable the attack.

- *Face recognition.* Apart from a few mobile devices with dedicated face recognition hardware, most devices utilize the general-purpose camera for face recognition. Apps can use either system-provided API or third-party SDK (software development kit) for face recognition. Meanwhile, some popular apps support face recognition for payment authorization.
- *Screen tap.* Screen tap is the most common physical input on mobile devices for confirmation action. For example, a single button tap is all it needs to authorize Single Sign-On.
- *NFC tag scanning.* Mobile phones with embedded NFC chips can be used either as an emulated card or a reader. Towards this end, some apps use such functions to read tags for authorization in different scenarios like payment. The user needs to put a tag near the mobile device to proceed.
- *Audio recording.* Though less common, some apps request for owners' voice or encoded ultrasound for payment or login.

C. The Required Steps of PHYjacking

To launch PHYjacking, the following requirements need to be satisfied:

- *RQ1: Enforce the target state.* The target app needs to be driven to the state where it is listening for physical inputs.
- *RQ2: Cover without interruption.* The malicious app can hide the authorization context in the target app by, *i.e.*, creating delusive visual covering while not interrupting the physical input listener in the target app.
- *RQ3: Zero permission.* For effectiveness consideration, the malicious app should not require any special permission which may raise suspicion from the victim or security vetting systems.

To achieve *RQ1*, the malicious app needs to call Android API like `startActivity` to invoke the authorization Activity in the target app, as shown by Step (A) in Fig. 4. Although Activities in Android disable direct invocation by default, some target apps which offer services like payment or Single Sign-On to third-party applications tend to enable it to enhance usability. Even if the target Activity is not directly invocable or requires button taps during authorization, the malicious app can still launch another Activity in the target app that can reach the authorization state with a few taps. From there, the malicious app can employ tapjacking to mislead the victim to navigate to the desired state. This approach is illustrated in Fig. 4 as the "optional path". An alternative approach is to put the malicious app in the background and passively wait for the desired state of the target app. However, in modern Android systems, both background running and inferring other apps states are mitigated. Therefore, the proactive approach is more practical and effective.

The hurdles for *RQ2* depend on the specific type of physical input to be hijacked. When the malicious app covers the target app after Step (B), either the target app or the system may interrupt the authorization process to protect the user from UI hijacking attacks, as shown by (C) and (D) in Fig. 4. Android systems and apps may take different mitigation strategies to handle the potential attacks. For example, recent Android versions provide an optional solution to prevent tapjacking, while the mitigation to prevent fingerprint hijacking is enforced. Apps can also implement their own foreground detection logic and stop accepting physical inputs in a timely manner.

The high-level framework of PHYjacking is depicted in Fig. 4. Depending on the implementation of the target app and the type of physical inputs, the attacker needs to choose different routes and apply various techniques. Details of each attack scenario and the corresponding techniques will be discussed in the following sections.

IV. New Techniques for Confusion Setup

To set up the confusion scene for PHYjacking, we need several UI attack techniques as the building blocks. As shown by Step (B) in Fig. 4, we need to first hide the authorization context. This can usually be achieved by creating some delusive top window to cover the actual authorization window. Second, the malicious app needs to drive the target app to the desired state to prepare for the attack. When direct invocation (A) is not viable, we can employ tapjacking technique to guide the user to navigate the target app to the desired state. For effectiveness consideration, we should achieve these steps without requiring special permissions for the malicious app.

A. Why Existing Techniques are Not Enough?

A handful of Android UI hijacking techniques already exist, which may potentially be used as building blocks for PHYjacking. In what follows, we review these relevant techniques and explain why they alone cannot meet our requirements.

a) Tapjacking. As a well-known UI attack, various tapjacking techniques have been discussed in literature [4], [5] and exploited by malicious apps in the wild [6]. Existing tapjacking on Android mainly rely on two features: 1) "draw over apps" window and 2) manipulated toast window. The former requires special permission, which contradicts *RQ3*. The latter has been patched in 2017.

b) Task Hijacking. Researchers in [7] discovered that, by manipulating the Android Activity stack model, adversaries can hijack the task stack and always put a malicious Activity above another Activity. One popular attack vector is setting the `taskAffinity` attribute to match the target app. Malicious apps mainly use this approach for phishing attacks. When applying task hijacking to our attack scenario, we found that it could not meet *RQ2* because the target app underneath became inactive, and the physical input listener would be interrupted.

B. New UI confusion techniques for PHYjacking

The aforementioned UI attacks either require special permission or cause interruption of the authorization process. To satisfy our requirements, we propose the following three UI confusion techniques as building blocks for PHYjacking.

a) Translucent Covering: Zero-Permission Overlay. Existing malicious overlay attacks usually try to achieve a persistent floating layer that will not be dismissed during app switching. However, this feature usually relies on special permissions or unfixed bugs. As we target one-time hijacking, persistent overlay is not required, and it is possible to find more effective ways of creating a covering layer without interrupting the underneath Activity. One suitable technique we discovered is to launch a covering Activity with the `Translucent` property. Android pauses an Activity if it is not in the foreground but only stops or destroys it when it becomes invisible to the user. Setting the `Translucent` (`android:windowIsTranslucent`) property for the covering Activity makes the system deem the underneath Activity as visible, which leaves the physical input listener staying alive in some cases. In actual attacks, the transparency of the covering can be configured so that it is visually opaque. By design, an Activity can set the `singleInstance` launch mode to ensure it is created in a separate task and no other Activities will be put into the same task. An example of such is the "System Setting" app, which is supposed to forbid any cover layer. However, if we launch the translucent covering with flags `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK`, it can be displayed on top of the target Activity regardless of its launch mode.

b) Zero-Permission Tapjacking. When the desired state of the target app is not directly reachable, the attacker can employ tapjacking to lure the user to navigate through the underneath app to reach the state required for the attack. Instead of using existing tapjacking techniques with permission requirements like the "draw over apps" window, we propose an alternative zero-permission tapjacking technique that uses the *translucent covering*. This is achieved through three key techniques:

1) *tap-through*: The covering should pass the tap event to the underneath window. This can be done by setting the `FLAG_NOT_TOUCH_MODAL` and `FLAG_NOT_TOUCHABLE` flags, as shown in [4].

2) *persistency*: Unlike a window with the "draw over apps" permission, our zero-permission covering will be switched to the background if the underneath app launches a new window. However, by relaunching itself at its `onPause` event, it can always revive on top. Also, by setting the `FLAG_ACTIVITY_NO_ANIMATION` flag, the transition can become unnoticeable to the victim.
3) *state inferring*: For a more convincing multi-step attack, the malicious covering needs to change its visual content adaptively. Although the covering cannot get the touch events, each trigger of its pause event indicates an Activity transition in the underneath app. It can also query the availability of the target hardware sensor to infer if the underneath app is in the desired input listening state.

c) Corrupting Activity Lifecycle with Race Condition. Although the translucent covering will not stop the underneath Activity, it still turns it into the paused state, which can interrupt the input listener in some cases. For instance, some apps explicitly close the listener upon `onPause` event. Also, some system APIs, like the fingerprint API, enforce foreground usage and automatically close the sensor when Activity state changes. To search for possible circumvention of these lifecycle-based mitigations, we test combinations of Activity launching APIs and flags to look for unexpected Activity Lifecycle states and discover the following race condition bug which has serious consequences: if one launches a translucent Activity within 100 msec after starting a normal Activity, the Activity Lifecycle can be driven into a limbo state where both Activities resume, contradicting the design principle of the Lifecycle model that only one Activity can be in the resumed state at any time.

To exploit this race condition bug, a malicious app can first launch the target Activity and then the covering within 100 msec, depicted as (A) and (B) in Fig. 4, to result in two simultaneously running (resumed) Activities. As shown in Listing 1, in normal cases, the underneath *VictimActivity* receives the `onPause` event as expected (line 5). When the race condition is triggered, the underneath *VictimActivity* stays in the resumed state (line 12). Leveraging this critical finding, we manage to bypass some system-level and app-level protection mechanisms. Refer to VI-D for details.

```
1  # Failed to trigger race condition:
2  20:38:55.065 lifecycleLog: Activity.onCreate: <VictimActivity>
3  20:38:55.085 lifecycleLog: Activity.onStart: <VictimActivity>
4  20:38:55.085 lifecycleLog: Activity.onResume: <VictimActivity>
5  20:38:55.155 lifecycleLog: Activity.onPause: <VictimActivity>
6  20:38:55.189 lifecycleLog: Activity.onStart: <MaliciousActivity>
7  20:38:55.190 lifecycleLog: Activity.onResume: <MaliciousActivity>
8
9  # Race condition triggered:
10 20:39:25.314 lifecycleLog: Activity.onCreate: <VictimActivity>
11 20:39:25.378 lifecycleLog: Activity.onStart: <VictimActivity>
12 20:39:25.379 lifecycleLog: Activity.onResume: <VictimActivity>
13 20:39:25.454 lifecycleLog: Activity.onCreate: <MaliciousActivity>
14 20:39:25.477 lifecycleLog: Activity.onStart: <MaliciousActivity>
15 20:39:25.480 lifecycleLog: Activity.onResume: <MaliciousActivity>
```

Listing 1: Activity events log when race condition failed and succeeded.

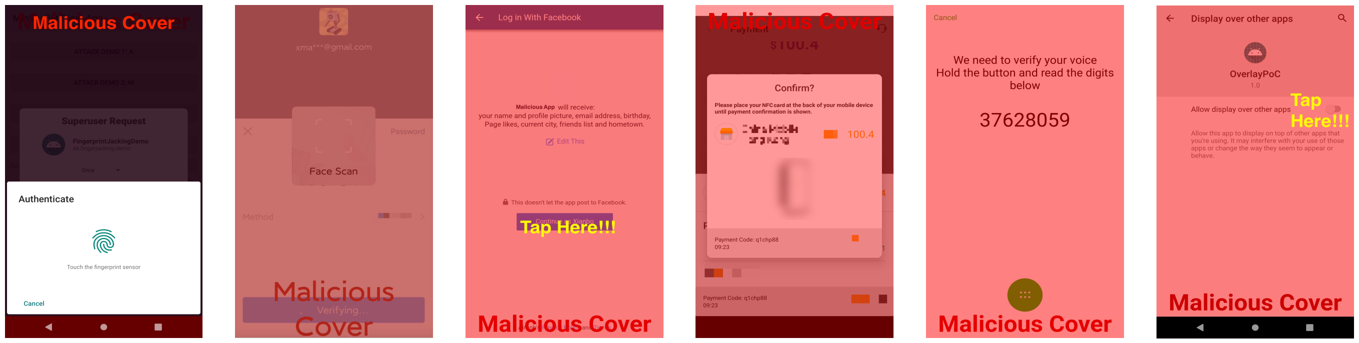
In Android, there are mainly two APIs for launching Activities. One is `startActivity` and another is `startActivities`. The latter can launch a sequence of Activities in a single call. Mechanisms of these two APIs have differences and vulnerabilities that only apply to the latter have been reported in the past [8]. PHYjacking involves launching at least two Activities, the target and the covering. Our experiments show that by making two `startActivity` calls within 100 msec or using `startActivities` can consistently trigger the race condition.

V. Analysis of System and App Protections

With the new confusion setup techniques we proposed in the previous section, we can achieve Step (A) and Step (B) in Fig. 4 and put the target app in the desired state waiting for some physical input before authorizing some action. However, it is still unclear whether the physical input can be hijacked due to possible system-level protection (C) and app-level mitigations (D). We studied several physical inputs, including fingerprint scanning, face recognition, NFC tag reading, and microphone recording. In this section, we present existing protections and practical attacks for each of them. Among them, fingerprint scanning and face recognition are the most employed physical inputs for in-app authorization. As such, we will discuss in detail the attacks against these two types of authorizations and refer them as fingerprint-jacking and facejacking respectively. After that, we present PHYjacking with other inputs, including screen touch, NFC tag reading, and microphone recording. As a preview, Fig. 5 shows screenshots of PHYjacking in action against some real apps using different physical inputs.

A. Mitigations in Apps

Apps using dedicated sensor(s) for authorization usually need to manage the opening and closing of the sensor(s) by themselves. Android does not document how the system can protect apps from hijacking for most physical inputs during authorization (except for a tapjacking mitigation [9]). On the other hand, some official guidelines mention that the hardware resource should be closed and released after the usage "for use by other application" [10]. In fact, it is important for apps to promptly cancel listening to physical input and close the sensor during authorization. Correct implementation of this behavior can mitigate the hijacking attacks to some extent. More specifically, to properly protect the app against PHYjacking attacks, developers need to detect and stop **background authorization** by themselves since system protection is not always available or reliable. The correct practice is relatively simple: interrupt the input listener or close the sensor in the `onPause` event of the authorization Activity. As a result, whenever a new Activity switches to the foreground, the fingerprint authorization process will be interrupted. We refer to this implementation as ***cancel-on-pause*** in the rest of the paper.



(a) Fingerprint-jacking (b) Facejacking (c) SSO Hijacking (d) NFC Hijacking (e) Voice Hijacking (f) Permission Hijacking

Fig. 5: PHYjacking against real-world apps with different physical inputs

B. Protections in System APIs

The possibility of UI confusion poses the risk of hijacking. If the system or app can guarantee that the user sees the actual authorization request, the risk can be mitigated. We study the source code of different versions of Android and APIs to check if the foreground presence of authorization Activity is securely enforced during physical input listening. Here, we focus on fingerprint and camera APIs. We will outline protections of other physical inputs like screen touch and NFC when introducing the corresponding PHYjacking attacks.

a) Fingerprint API. As discussed in II-D, Android has two sets of fingerprint APIs. One is the legacy `FingerprintManager` API that has been available since Android 6 (2015). The other one is the new `BiometricPrompt` API introduced in Android 9 (2018), which provides “a safe, familiar UI for user authentication” [1]. Before Android 9, there is no hijacking protection in the `FingerprintManager` API, where apps may occupy the fingerprint scanner in the background. A malicious app can quickly launch the target Activity and then switch it to the background, hide the context, and lure the user to input the fingerprint. To bridge the gap, Android 9 added a patch [11] to the `FingerprintManager` API to mitigate the fingerprint hijacking risk. The patch aims to perform a consistency check between the foreground Activity and the fingerprint Activity whenever the Activity stack is changed (*e.g.*, when there is an Activity switch). We call this mitigation mechanism ***check-on-stack-change***. The exact mitigation mechanism also appears in the `BiometricPrompt` API. Unlike the `FingerprintManager` API, where the UI during fingerprint authentication needs to be implemented by the app developer, the new API provides a unified dialog with customizable text content. The cancellation of the fingerprint listener is also handled by the API, which enforces *cancel-on-pause*.

b) Camera API. Like fingerprint APIs, in older versions of Android (before Android 9), there is no restriction for apps to use cameras in the background. Apps using the camera, especially for authorization purposes, should close the camera immediately when it is not in the foreground (in the `onPause` event). Starting from Android 9, according to the documentation, new limitations are introduced for accessing sensors from the background, including disabling background camera access [12]. Apps that require camera access in the background should use foreground service and declare the corresponding permission. While the foreground service is running, Android displays a persistent notification to inform the user about it. This protection mechanism also applies to the legacy camera API (Camera 1) and the new camera API (Camera 2).

C. Common Pitfalls in App Implementations

As mentioned earlier, to shield users from PHYjacking attacks, app developers should implement the *cancel-on-pause* behavior in their apps since system level protection is not always available. However, we find that many apps do not provide such protection or have a flawed implementation. More specifically, we have found two common, flawed implementation patterns:

- Many apps do not handle the physical input cancellation by themselves, which is referred to as ***never-cancel*** implementation.
- Some apps only cancel the fingerprint authentication in `onStop` or `onDestroy` event of the corresponding Activity instead of the `onPause` event, we call this incorrect implementation pattern as ***pause-failure***.

D. Weakness in the System Protections

For some security-essential sensors or physical interfaces, Android has recently added some system-level protections, as we have seen in fingerprint and camera APIs. However, upon examining their mechanisms and actual implementations, we have identified several weaknesses that can lead to PHYjacking attacks.

a) Mitigation bypass in the fingerprint API. By analyzing the *check-on-stack-change* mitigation introduced in Android 9 to fingerprint APIs, we identify a weakness caused by an incorrect assumption: it assumes that when the fingerprint listener starts, the authentication Activity is in the foreground. However, if the fingerprint scanner starts after the authentication Activity is covered by another Activity, the check will never be triggered. There are several ways to achieve such condition, depending on various implementation patterns of apps and different system features. These tricks will be presented when we use them to construct fingerprint-jacking attacks in VI-C. There is also another unaddressed risk: the "draw over apps" window is not in the Activity stack and thus would never trigger the check. Nevertheless, such an issue is beyond the scope of our threat model which requires zero permission.

b) Noneffective protection in the camera API. Although official documentation states that apps cannot access the camera from the background starting from Android 9, it only applies to cases where the app is trying to access the camera from a background service or scheduler [13]. In contrast, we find other ways for apps to run tasks when it has no foreground Activity. For instance, when the Activity is switched to the background, the process is cached without being killed immediately. The threads created by the Activity can still run for several minutes before the system scheduler decides to kill it. Meanwhile, the camera is left accessible for precisely one minute in our experiment on Android 11. This behavior renders the protection useless against our PHYjacking framework where the target face recognition Activity under the malicious covering can still access the camera. The one-minute time window is long enough for a malicious app to lure the user into completing the face recognition process.

VI. Hijacking Physical Inputs

With the confusion setup techniques discussed in IV and mitigation bypasses introduced in V, we have workable solutions for all key links (A), (B), (C) and (D) in Fig. 4. In this section, we show the details of practical PHYjacking attacks targeting different physical inputs.

A. Existing Attacks

Before introducing new PHYjacking techniques, we first review some existing attack techniques and discuss their limitations. The earliest work mentioning the idea of using Android UI confusion for fingerprint hijacking is [14]. However, at that time, the Android official fingerprint API was not even released yet, and the researchers did not discuss the technical details behind. We test their approach on Android 6 and construct a similar two-step attack: 1) The malicious app invokes the fingerprint listening Activity in the target app. 2) After a while, the malicious app in the background launches an Activity to cover the fingerprint app. We call this attack the *Trivial-attack* in this paper. For this attack to work, the following implicit requirements have to be satisfied:

- The app has the *never-cancel* flaw, namely, it never cancels the input listener.
- Android system allows apps running in the background to continue listening to the physical inputs. The `FingerprintManager` API before Android 9 allows this, so does the camera API.

Thus, this attack can only work for a small set of flawed apps and is disabled on the recent version of the fingerprint API. The authors of [15] conducted a more recent study on Android 7. They introduced fingerprint UI attacks with the "draw over apps" overlay as well as the trick of screen dimming, both require some special permissions being granted by the user to the malicious app. We will refer them as *Float-attack* and *Dimming-attack* respectively. Similarly, authors of [16] also used the "draw over apps" window for clickjacking on the system camera app to get images without the camera permission in the malicious app, which is also a case of *Float-attack*. All of these existing attacks either rely on special permissions or can only be applied in some restrictive circumstances. We seek more practical and powerful attacks.

B. Facejacking: Exploiting Flawed App Implementations

The *Trivial-attack* assumes *never-cancel* implementation flaw in apps, which is a bit demanding in practice as many apps follow the official documentation to close the sensor somewhere. However, as we have discussed in V-C, the timing of the closing call also matters. We observe that a significant portion of apps suffer the *pause-failure* (as later will be shown in the evaluation results in VII-A). Based on such observation, we propose an attack using the translucent Activity to cover the target Activity, which can avoid interrupting the physical input. We name this attack as *Translucent-attack*. As introduced in IV-B1, Activity covered by the translucent covering stays in the paused state and never triggers the `onStop` event. Fig. 6 illustrates this attack with comparison to the *Trivial-attack*. Combining this technique with ineffective system protection, all apps using Android camera API for authorization without proper cancellation are vulnerable to the face recognition hijacking attack, facejacking in short. This technique also applies to fingerprint-jacking against apps using fingerprint API on pre-Android-9 devices, where the *check-on-stack-change* mitigation is absent.

C. Bypassing Fingerprint-jacking Mitigation in Android 9

The *Translucent-attack* as well as the *Trivial-attack* only work for fingerprint-jacking before Android 9. Due to the *check-on-stack-change* mitigation added in Android 9, the fingerprint listener will be interrupted immediately on launching the covering. However, as we discussed in V-D, the patch mechanism is imperfect. It assumes that the fingerprint listener starts before the covering is launched. If we ensure the fingerprint listener is initialized only after the presence of the malicious covering, we can avoid triggering the mitigation code. We introduce two new mitigation bypassing attacks based on this observation.

1) Wakeup-bypass. One corner case overlooked by the mitigation is when the device wakes up. The attacker can cover the fingerprint Activity with malicious Activity before the device goes to sleep. After wake-up, both Activities resume, and then the underneath one is paused. If a fingerprint Activity automatically starts listening at the `onResume` event (we call this ***auto-resume*** pattern) and has the *pause-failure* flaw, the fingerprint sensor can work in the background, avoiding any stack change and thus evading the checking. Fig. 7 illustrates

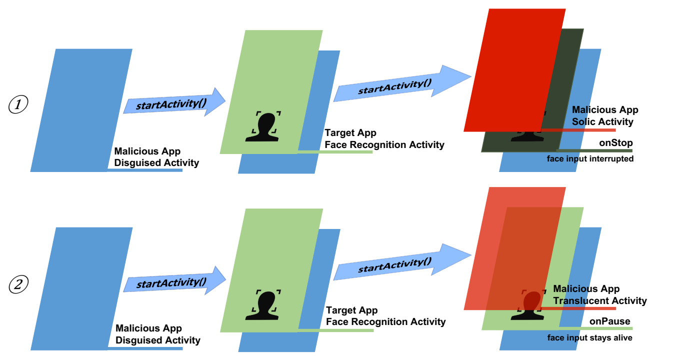


Fig. 6: Illustration of *Translucent-attack*, with comparison to *Trivial-attack*. If the app cancels the authorization in the `onStop` method, *Trivial-attack* in ① will not work, but our new *Translucent-attack* as shown in ② will work.

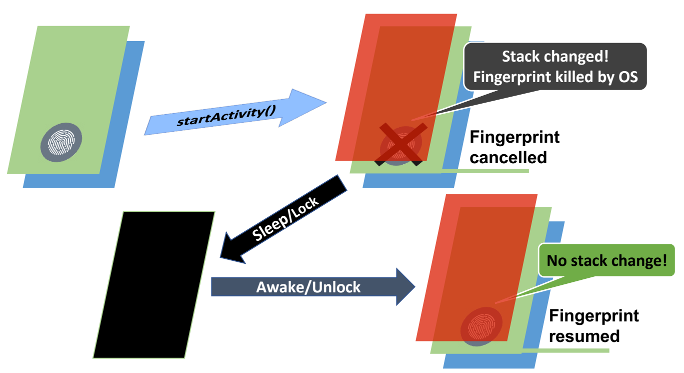


Fig. 7: Illustration of the *Wakeup-bypass* in Android 9 and later. After the device wakeup, there is no stack change, so there is no checking when the background Activity starts fingerprint listening.

this process. In practice, waiting for the device to sleep after setting up the fingerprint and covering Activities may not be effective, as the user may switch to other Activities during this period. A better approach is to monitor the `ON_SCREEN_OFF` event in the background and launch the attack right before the device sleeps. To summarize, the *Wakeup-bypass* can work in all Android versions, but with the following assumptions:

- The attacker or the malicious app can layout the desired Activity stack before the device sleeps and wait for the victim to wake up the device.
- The app's implementation must contain both the *auto-resume* and the *no-pause* patterns.

Note that we can eliminate the *auto-resume* assumption by combining tapjacking in some scenarios.

2) Multiwindow-bypass. We can treat the *Multiwindow-bypass* as a variant of the *Wakeup-bypass*, assuming that the device is running under the multi-window mode (supported by Android since version 7 and is commonly used on tablets). Although multiple Activities are visible to the user in the multi-window mode, only the top Activity in one active window can be in the resumed state. We find that there are two scenarios for the active window to switch from one to another: 1) When the user taps in the inactive window, it will become active, and the top Activity in it will be resumed. 2) When the user resizes the window, the larger window will become active regardless of which window the user previously was interacting with. When these two cases happen, the Activity stack does not change but the Activity status changes and events get triggered. Thus, we can create an attack similar to the *Wakeup-bypass* assuming the device is running in the split-screen mode where two windows are present. One is the active window with the malicious app running inside (*malicious-window*). Another is the adjacent window where the target victim app will be put into (*victim-window*). The workflow of the attack is illustrated in Fig. 8 and the steps for launching the attack are described as follows:

1) The malicious app in *malicious-window* sets up the desired Activity stack for the *victim-window*. This can be done by starting Activities with the `FLAG_ACTIVITY_LAUNCH_ADJACENT` flag.
2) Wait for the user to interact with or enlarge the *victim-window*. When it happens, the covering and the target Activity underneath are both resumed to enable fingerprint hijacking without triggering the *check-on-stack-change* mitigation.

In practice, the attack can be launched in a phishing fashion: put a new Activity in the split-screen and disguise it as a legitimate service feature, *e.g.*, side-by-side reading, to lure the victim to interact with it.

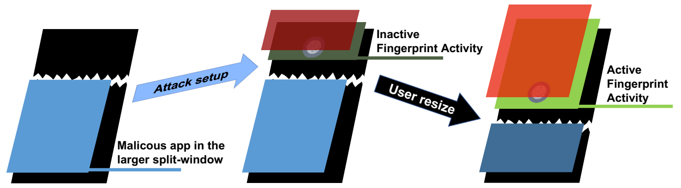


Fig. 8: Illustration of *Multiwindow-bypass*. When the user enlarge the upper window, the Activities inside it receive the `onResume` event and then fingerprint-jacking happens.

D. One Trick to Rule Them All: the Race-attack

All the attacks mentioned above rely on apps' failure in handling physical input cancellation properly. While these flawed implementation patterns are widespread, one may think that once developers do things right, PHYjacking should no longer be a concern. Unfortunately, with the *Race-attack* we discovered, this is not true. Furthermore, the *Race-attack* can also bypass the *check-on-stack-change* mitigation in fingerprint APIs while requiring fewer user interactions comparing to the aforementioned bypassing techniques. A sample implementation of a malicious app leveraging the *Race-attack* is given in Appendix C.

The *Race-attack* exploits a race-condition bug in Android Activity Lifecycle as shown in IV-B3. The bug can be triggered reliably and the attack can be launched in a deterministic manner due to the relatively long racing time window (about 100 msec). Since the bug breaks the Activity lifecycle and leaves the background Activity in the resumed state, it can be exploited to bypass the *cancel-on-pause* mitigation in apps. Meanwhile, we have found that the *check-on-stack-change* mitigation in Android 9+ is also invalidated as the creation of the covering can be faster than the fingerprint initialization. So the stack change happens before the mitigation is ready. With the *Race-attack*, we can even launch hijacking attacks against apps using the new `BiometricPrompt` API, despite the unified dialog provided in the API has the *cancel-on-pause* mitigation. As shown in Fig. 5a, some apps do not set context information on the dialog and assume that users can see the underneath host Activity. In this case, while we cannot cover the dialog without dismissing it, we can insert the malicious covering between the dialog and the host Activity with the *Race-attack* to hide the context.

Table II summarizes all of the attacking techniques described above while highlighting their requirements and dependencies for the case of fingerprint-jacking. Red cells are conditions where the attack would not work. An attack technique with more green cells in its row is more powerful. The table shows that our new techniques can enable zero-permission attacks on Android 9 and above, and the most powerful attack is the *Race-attack*.

E. Hijacking Other Inputs: Tap, NFC, and Voice

Fingerprint and face capture are two major biometric authentication inputs. However, other physical inputs can also be involved in authorization processes. Screen touch is the most common physical input on mobile devices. It is widely presented as a confirmation button tap in many authorization scenarios, *e.g.*, Single Sign-On (SSO) and permission granting. NFC tag reading is another authorization input that is being used in some apps. The microphone is also a major input source on mobile, but it is less common for apps to use voice input during authorization. We manage to find such a use case in a top-tier social app. In this section, we demonstrate the possibility of hijacking these alternative inputs via some interesting examples. Compared to the fingerprint-scanner and camera API, there tends to be fewer hijacking protections on these inputs, and similar attack flow can be applied.

a) Zero-permission tapjacking for SSO profile stealing. Many authorization processes require user consent by tapping on a particular widget on the screen. As just another physical input, the screen tap can also be hijacked following the PHYjacking framework, only that it lures for screen touch instead of fingerprint scanning. This attack is generally known as tapjacking. Android provides a security mechanism to help apps preventing tapjacking. Apps can either detect the `FLAG_WINDOW_IS_OBSCURED` flag to determine whether a screen tap has passed through some covering layer or directly set the `android:filterTouchesWhenObscured` attribute to a widget. However, this protection is disabled by default. Apps that do not explicitly enable them are left unprotected. According to our PHYjacking framework, this protection is considered to be an app mitigation (D) instead of OS protection (C) since it is not applied globally.

Single Sign-On (SSO) is one of the most common situations where a single tap is the core authorization action, thus becoming a sweet tapjacking target. Under the covering layer of Fig. 5c is the recognizable interface of the "Login with Facebook" function provided by Facebook. Other apps, as relying parties (RP), can retrieve user profile information, including users' actual identity, from Facebook. Here, Facebook plays the role of an Identity Provider (IdP). Most IdPs simplify the authorization process to a single tap if the user has already logged into the IdP app. This kind of single-button consent window is the only gate of the authorization process. Malicious apps can also request user data from IdPs, but users can notice its intention from the consent window and reject it. However, if a malicious app can cover the consent page with another window while passing through tap events, it can easily trick the victim to tap the confirmation button unintentionally. Such single tap will grant the malicious app access to the victim's private information on the IdP. To make the attack unnoticeable, we need precise timing to bring the covering to the front right after the consent window is presented. We have successfully addressed this challenge by reverse-engineering the SSO libraries and move logics that cause delays, *e.g.*, network requests, to the malicious app. Refer to VI-F3 for a detailed description.

b) Hijacking NFC tag scanning. Android API for accessing NFC tags is quite different from fingerprint scanner or camera APIs. The OS has a specific tag dispatch system [18] that relies on the intent filtering mechanism to pass some NFC tag data. Instead of the "open sensor – read data – close sensor" paradigm used in camera and fingerprint scanner, the NFC scanner is always in the listening state, and apps only need to set a receiver that will be triggered when an NFC tag is scanned. Meanwhile, Android provides neither standard user interface nor any hijacking protection during NFC scanning. Although apps cannot cancel NFC scanning, they can stop tag data reading or interrupt follow-up authorization when in the background to protect themselves from hijacking. Nevertheless, it is unlikely for app developers to be aware of this risk and implement the corresponding mitigations. For apps without any hijacking mitigation, a malicious app can simply conduct PHYjacking against the NFC reading interface using a translucent covering. Fig. 5d shows a PHYjacking example in action which targets a popular mobile wallet app that is waiting for an NFC tag scanning to complete a payment transaction.

c) Hijacking microphone input. Voice recognition is another biometric authentication technology. However, it is not as common among mobile apps when comparing to more mature solutions like fingerprint scanning and face recognition. Since the microphone on mobile devices is mainly for general audio input, the security consideration in the API is not as thorough as fingerprint APIs. As a general media input device, it follows a very similar design with the camera API. In other words, an app needs to close the microphone properly to prevent

TABLE II: Implementation and Environment Assumptions on Different Fingerprint-Jacking Attacks

Attacks		System requirement	Implementation flaw dependency		Implementation pattern dependency		Attacker capability requirement	
		Require Android Ver.<9	Rely on *never-cancel*	Rely on *pause-failure*	Require *auto-resume*	Require *no-button* [1]	Require a malicious app	Malicious app's permission
Known attacks	*Trivial-attack* [2][14]	✓	✓	✓	✗	✗	✗	None
	Float-attack [15]	✗	✗	✗	✗	✗	✓	SYSTEM_ALERT_WINDOW[3]
	Dimming-attack [15]	✗	✗	✗	✗	✓	✓	WRITE_SETTINGS[3]
New attacks	*Translucent-attack*	✓	✗	✓	✗	✗	✗	None
	Wakeup-bypass	✗	✗	✓	✓	✗	✓	None
	Multiwindow-bypass	✗	✗	✓	✓	✗	✓	None
	Race-attack	✗	✗	✗	✗	✗	✗	None

[1] *no-button* means no additional interactions like button-tap before fingerprint. ✗ indicates that the attack can still work by combining tapjacking.
[2] As details of the attack demonstrated in [14] were not given, some conditions here for *Trivial-attack* are based on our own testing.
[3] Special permissions that require users' explicit approval. Only before 2019, SYSTEM_ALERT_WINDOW was auto-granted to apps from Google Play [4], [17].

PHYjacking attack. The microphone API even shares the same security issue we identified for the camera API: an Activity switched to the background can continue accessing the microphone, even though official documentation states that apps can no longer access the microphone in the background starting from Android 9. Fig. 5e shows it is possible to use a translucent covering for PHYjacking against a popular social app which uses voice input for login authentication.

F. Exploring Other Attack Techniques for Impact Escalation

With the attack techniques mentioned above, we already can construct practical PHYjacking attacks. However, there are still two limitations. First is the requirement of a malicious app, even though it is zero-permission. The second is the non-persistency of translucent covering. Although the attacker can use the "relaunch when paused" trick to keep it on top, users can still dismiss it by tapping the home button, which is not an issue in the "draw over apps" overlay. In this section, we discuss possible workarounds for these two limitations.

a) Possibility of launching PHYjacking from web pages. Under certain conditions, PHYjacking attacks can be launched solely from a malicious web page. To achieve this, the attacker first needs to find a way to initiate the targeting authorization process from a web page, with can be done with remote app linking mechanisms [19], *e.g.*, deep links (`payapp://dopayment?mode=fingerprint`). One common scenario where this can happen is for mobile wallets to support website payment by registering custom deep link. The more challenging part is to construct a delusive UI covering from the web page. This requires the attacker to find an Activity in an installed app that satisfies 1) being translucent, 2) can be launched using deep-link or other remote linking mechanisms from the browser, 3) its content is (partially) controllable. We call this kind of Activities *covering-gadgets*. A potential place for the *covering-gadget* to appear is in apps that load external websites in WebView. Previous research has identified such controllable WebView Activities in many popular apps [20], but it is unlikely for most Activities to set the translucent attribute. To investigate whether such *covering-gadgets* indeed exist, we conducted a small scale experiment and were able to find two popular apps containing potential *covering-gadgets*. The details are presented in Appendix A.

b) Escalating to the "draw over apps" permission. Apart from third-party apps, tapjacking can also apply to some Android system components. Some permissions are considered dangerous in Android and only granted if switched on by users. The overlay permission (`SYSTEM_ALERT_WINDOW`) is one such dangerous permission. Apps with this permission can draw persistent floating windows over other apps. Previous work has discussed how to abuse this permission for mighty attacks [4]. It is also known to be exploited by malware [6]. Google used to grant this permission to apps installed from Play Store automatically, but later changed to a stricter policy that only grants it to predominant apps [17]. Although the switch of some critical permissions like the "a11y" are generally protected with the tapjacking mitigation mentioned in VI-E1, we find that the "draw over apps" permission switch is not protected in Android 11 and earlier versions. Thus, an attacker can use zero-permission tapjacking to lure the user to enable this permission. Fig. 5f demonstrates the attack in action. Since the "draw over apps" overlay is not considered as an Activity in the stack, the mitigations including the *cancel-on-pause* by apps and the *check-on-stack-change* in the fingerprint API are both ineffective against malicious overlays based on it.

c) Precise SSO Hijacking Timing Control. To hide the SSO window from the user, the malicious app needs to launch the covering immediately after the SSO Activity appears, which is non-trivial. Most of the SSO libraries provided by IdPs display the interface only after completing a few network communications, introducing non-deterministic delays. If we launch the covering Activity too soon, the SSO Activity will become the foreground. If we run it too late, the victim may notice the covering transition. To achieve the precise timing control, the malicious app needs to know the state of network requests. Although researchers have shown that the state can be inferred from some side channels [21], it is more efficient and robust to move the network requests from the SSO library to the malicious app. In this way, the malicious app can know the exact response time and send Intent messages to directly invoke the SSO Activity along with the covering.

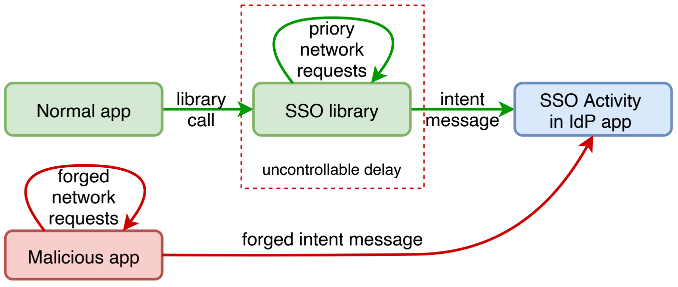


Fig. 9: Normal workflow of SSO v.s. attack flow. The malicious app can implement functions of SSO library to directly invoke SSO Activity for fine timing control.

The Intent message for SSO invocation usually contains some dynamically fetched parameters and calculated values like message authentication code (MAC). To correctly forge the requests and construct the Intent message, the attacker needs to reverse-engineer the library of the target SSO platform. As a proof-of-concept, we analyze the library provided by a popular social platform and show the feasibility of this attack: If we directly call the SSO library, the authorization window has a 3 to 5 seconds delay, which reduced to less than one second after moving the network request and Intent invoking logic into our malicious app. Fig. 9 illustrates the attack flow.

VII. Measurement Study

Many of the attacks we discussed depend on apps implementation patterns. In some cases, an app with proper implementation can mitigate the PHYjacking attack. To understand the real impact of this attack, we design a static analyzer to automatically examine the implementation of a large number of apps in practice. As tapjacking can serve as a stepping stone for other more sophisticated PHYjacking attacks, we also analyze the apps to see how many of them enable the optional tapjacking mitigations provided by Android.

A. Check Call Graph for Flawed Implementation Patterns

The correct app implementation that can mitigate the PHYjacking risks can be described with one condition: *The Activity opening the input listener immediately close the listener when it is paused*. All other implementation patterns are susceptible

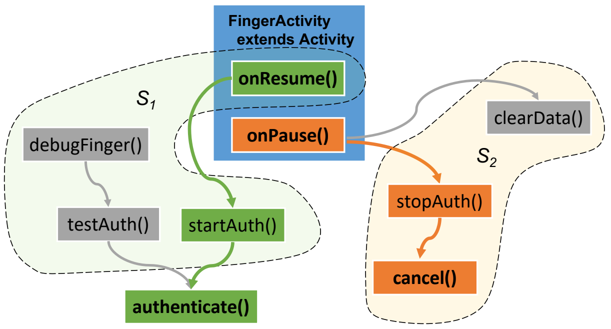


Fig. 10: Logic of the call graph analyzer with fingerprint API.

TABLE III: API Calls Defined in the Call Graph Analyzer

API	Sensor Open	Sensor Close
FingerprintManager	.authenticate()	CancellationSignal.cancel()
Camera2	.openCamera()	.close()
Camera	.open()	.release()
Camera (preview mode)	.startPreview()	.stopPreview()

to PHYjacking. We can check this condition by analyzing the call graph of an app with the following algorithm:

1) Locate the sensor opening API call and trace the call chains backward (reverse reachable) until locating a call defined inside an Activity, which is the target Activity.
2) Extract forward call chains (reachable) from the `onPause` of the target Activity and search for the sensor closing API call. If the closing call can be located, the implementation is correct.
3) Optionally, we can search for sensor closing call from `onStop` and `onDestroy` to determine whether it is a *pause-failure* or *never-cancel* flaw.

Fig. 10 illustrates our static analysis algorithm. S_1 is the reverse reachable set of the sensor opening method (`authenticate()` in the case of fingerprint API). S_2 is the reachable set from the target Activity's `onPause()` method. After constructing the call graph, the analyzer tries to link these two sets to determine if the app's implementation can mitigate the PHYjacking risks. Table III lists the sensor opening and closing calls examined by our analyzer. Note that the `BiometricPrompt` API is not included because it provides a unified interface that handles sensor closing for apps.

Implementation. We extend Androguard [22] to construct the call graph of an app and perform reachability analysis to determine whether the app has vulnerable implementation patterns. We also implement algorithms to extract class hierarchy and handle Java interfaces to fill some missing links in the call graph extracted by Androguard. The source code of this tool is made available on GitHub [23].

Results. We collected 2024 apps that declared the `USE_FINGERPRINT` permission and 6324 apps with the `CAMERA` permission. We ran 8 call graph analyzer instances in parallel on a machine with 20 cores (2.4GHz) and 64GB memory. The average testing time for each app is 77 seconds, and the average memory consumption is 1201 MB. Table IV depicts the results. We first exclude about half of the apps where related calls cannot be located, as well as 42 apps that trigger unexpected analyzer errors. All these apps are reported under the "Others" category. Among the remaining 3532 analyzable apps, almost half of them contain implementation flaws, and a significant portion of them fall into the category of *pause-failure*. Note that, flawed implementation found by the analyzer is not an immediate confirmation of exploitable vulnerability. Further manual analysis is needed to confirm the usage scenario and security impact. On the other hand, even apps with correct implementation can be vulnerable to the *Race-attack*. Appendix B provides further breakdown on

TABLE IV: Analyzer Results on API Implementation Flaws

Target	Total	Correct	*pause-failure*	*never-cancel*	Others
Fingerprint	2024	**378**	**547**	**34**	879
Camera	6324	**1428**	**1118**	**27**	3759

TABLE V: Camera API Usage of Face Recognition Providers

Provider	**API location**	**API version**	**Input Mode**			**Close camera**
			Preview	**Shot**	**Delegate**	
ArcSoft	Official demo	Camera 1	Y	N	N	`onDestroy`
Baidu	SDK	Camera 1 & 2	Y	Y	N	`onPause`
iFLYTEK	Official demo	Camera 1	N	N	Y	Never
Face++	Official demo	Camera 2	Y	N	N	`onPause`
Neuro	SDK	Camera 2	Y	N	N	Never
Luxand	Official demo	Camera 1	Y	N	N	`onStop`
3DiVi	Official demo	Camera 1	Y	N	N	`onDestroy`

the "Others" category and describes the dataset. The appendix also gives an estimation of the false positive rate.

B. Counting Apps with Tapjacking Protection Enabled

As discussed in VI-E1, there are several ways to enable the tapjacking protection provided by Android. Since those are either static attributes or single function calls, we use simple pattern matching to search for apps' protection. We randomly sampled 10,000 apps in the market for evaluation. The result is astonishing: only 403 apps have enabled the protection for at least one widget. Note that this does not mean those apps are safe against tapjacking as they may not enable the protection for all security-sensitive places. After all, as shown in VI-F2, we have found that even some native Android modules fail to enable the required protection. Worse still, while a new `FLAG_WINDOW_IS_PARTIALLY_OBSCURED` mechanism has been added in Android 10 to address the "hollow tapjacking" attack [6] (which exploited inadvertent settings of the `FLAG_WINDOW_IS_OBSURED` flag), none of the 10,000 apps we evaluated set this new protection flag.

C. Protection in Face Recognition Libraries

Apart from apps using the camera, it is also revealing to evaluate the security of libraries and demo programs provided by face-based authentication service providers. Recently, many new businesses provide cloud or offline face recognition support on mobile devices. We collect Software Development Kits (SDKs) and official demos from various mainstream face recognition service providers and analyze their camera-input implementation patterns. As shown in Table V, some of these SDKs take care of calling the camera API while others only accept images and rely on the app to call the camera API. Meanwhile, most of them also provide official demo apps as reference implementations. We check whether those SDKs and their official demo apps contain the aforementioned implementation pitfalls when handling the camera API calls. We found that only one out of five providers has the correct *cancel-on-pause* mechanism in the demo app. Two providers wrap camera API calls in their SDKs, but one of them fails to handle it correctly, which renders all the apps using the SDK susceptible to facejacking.

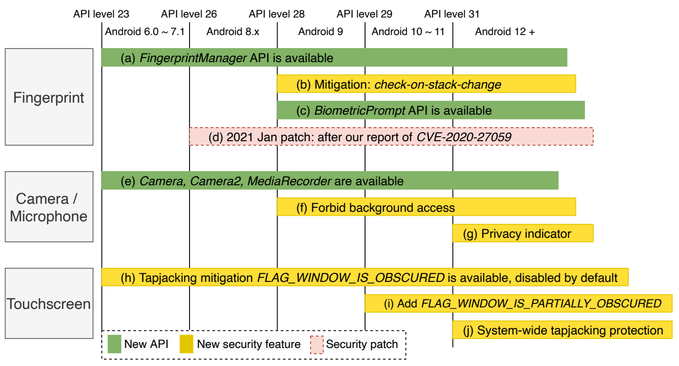

API level 23
API level 26
API level 28
API level 29
API level 31
Android 6.0 ~ 7.1
Android 8.x
Android 9
Android 10 ~ 11
Android 12 +
Fingerprint
(a) *FingerprintManager* API is available
(b) Mitigation: *check-on-stack-change*
(c) *BiometricPrompt* API is available
(d) 2021 Jan patch: after our report of *CVE-2020-27059*
Camera / Microphone
(e) *Camera, Camera2, MediaRecorder* are available
(f) Forbid background access
(g) Privacy indicator
Touchscreen
(h) Tapjacking mitigation *FLAG_WINDOW_IS_OBSCURED* is available, disabled by default
(i) Add *FLAG_WINDOW_IS_PARTIALLY_OBSCURED*
(j) System-wide tapjacking protection
New API
New security feature
Security patch

Fig. 11: Security updates in Android physical input APIs.

VIII. Discussion on Mitigations

Fig. 11 summarizes the security related changes in major physical input APIs across different Android versions, including the latest Android 12. Details on some of these changes have been discussed in V.

Google has released a patch (Fig. 11(d)) for the fingerprint API on Android 8 to 11 in January 2021 to fix the vulnerability reported by us. The patch adds a foreground checking on each authentication and can effectively mitigate all of our zero-permission fingerprint-jacking attacks. However, this checking cannot mitigate attacks with the "draw over apps" overlay. We speculate there might be usability considerations like supporting fingerprint authentication during a video call. One possible mitigation is to introduce an additional protection mechanism that temporarily hides all overlays whenever the fingerprint scanner is active.

Android 12 introduces a global indicator for camera and microphone access (Fig. 11(g)) to address privacy concerns. While this can make PHYjacking less stealthy, a malicious app can still confuse the victim by, *e.g.*, convincing the victim that the microphone is activated by the malicious app itself for some legitimate purpose. There are several comprehensive UI protection mechanisms proposed in recent research papers [24]–[26]. However, none of them have been integrated into Android, possibly due to side effects on usability or implementation complexity. The way Google has patched related issues seems to indicate that, for a mitigation to be integrated into Android, it should be the least noticeable to users and introduce minimum code changes. To prevent PHYjacking attacks, Android can apply similar mechanism in the fingerprint API to other sensor APIs like camera and microphone to ensure the visibility of the sensor-using Activity. Apps that need background access to those sensors should request special permissions from the system.

For end users, upgrading to Android 12 or installing the security patch on Android 8 to 11 (Fig. 11(d)) can mitigate zero-permission fingerprint-jacking attacks. Devices running Android 11 or earlier without the patch are vulnerable because different attacks are still applicable as shown in Table II. Camera and microphone PHYjacking risks continue to be present on all Android versions but the privacy indicator on Android 12 (Fig. 11(g)) may help victims to spot such attacks. While Android has made available for years optional tapjacking protections (Fig. 11(h,i)), they have received little adoption by app developers per our measurement study in VII-B. The system-wide, activated by default, tapjacking protection (Fig. 11(j)) offered by Android 12 can serve as a more foolproof solution.

IX. Related Work

PHYjacking can be considered as a "Confused Deputy" attack where the user acts as the confused deputy whose authority in the form of physical inputs gets misused. Android security researchers have proposed several UI confusion techniques to enable attacks like phishing, tapjacking, and keyboard logging. Many recent attacks [4], [27], [28] exploit the "draw over apps" window as the delusive overlay. Beyond that, authors of [20] perform a comprehensive study on the possibility of launching Android UI attacks from web pages. [7] analyzes the Android task state transition model and identifies several ways to confuse the task stack manager. The false transparency attack introduced in [29] confuses the user with a visually transparent malicious app for Android permission phishing. Yet, all these prominent works do not investigate the impact on other physical inputs except screen touching. Due to stringent protection mechanisms in some physical-input interfaces like fingerprint-scanning, UI confusion alone is insufficient for realizing attacks. PHYjacking often requires confusing the OS or apps for mitigation bypass, like in the *Race-attack*. Various detection and defense mechanisms against Android UI attacks have been proposed recently [25], [26], [30]–[32]. Nevertheless, they are not officially adopted by Android, possibility due to side effects on usability or implementation complexity.

The use of fingerprint scanner in Android apps is a relatively new feature and thus not a well-researched topic by the security community. To the best of our knowledge, [14] is the first UI confusion attack targeting fingerprint with a working demonstration, but Android fingerprint API has introduced many changes since then. A more recent study [15] focuses on Android 7 introduces fingerprint UI attacks with floating-window or screen-dimming tricks, both requiring special app permissions. As for camera, the floating-window is also exploited by [16] to hijack the camera app in Android and take photos without permission. In our work, we not only study general physical inputs hijacking but also investigate mitigation weaknesses and propose new zero-permission attacks.

In terms of general hardware resource hijacking on Android, the most relevant work is [33], which models the Resource Race Attack where one app may preempt some exclusive hardware resource from another. They introduce attacks on Android against the camera and touchscreen, but do not cover the rest of sensors we study. Besides, they target privacy leakage attacks like photo stealing via existing UI attack techniques like the "draw over apps" window. In contrast, our focus is on authorization hijacking.

X. Conclusion

In this paper, we propose a general framework to realize practical authorization hijacking attacks targeting various physical inputs including fingerprint scanning, face recognition, *etc.*. We introduce various new attack techniques, including exploiting common implementation flaws in Android apps, bypassing recent mitigations introduced by Android, and a powerful race-condition attack that can break the Android Activity Lifecycle model. We also discuss other impactful attacks like SSO hijacking and overlay permission escalation. Via automatic and manual analyses, as well as proof-of-concept malicious apps against prominent service-apps, we demonstrate the practicality of PHYjacking and their critical security impact.

Acknowledgement. We thank our shepherd Prof. Alexandra Dmitrienko and the anonymous reviewers for their valuable comments, which help to improve the paper considerably. This research is supported in part by the CUHK Project Impact Enhancement Fund (Project#3133292), the CUHK Direct Grant #4055155, and the CUHK MobiTeC R&D Fund.

Responsible Disclosure. We have notified Google and the vendor(s) of the affected major app(s) about our findings. Two Android bug reports were submitted to Google in June 2020. One is the mitigation bypass of the fingerprint API, and another is the tapjacking on the "draw over apps" permission switch. Both issues were assigned with high severity. The first one was assigned CVE-2020-27059 and has been fixed in January 2021 [34]. The second was determined as a duplicate with an issue reported internally by a Google engineer, which has not been fixed as of this writing (July 2021).

References

[1] "One Biometric API Over all Android," Oct. 2019. [Online]. Available: https://android-developers.googleblog.com/2019/10/one-biometric-api-over-all-android.html

[2] "Android Distribution Dashboard," Mar. 2020. [Online]. Available: https://developer.android.com/about/dashboards

[3] "Unsupported devices for biometricprompt," Jun. 2021. [Online]. Available: https://android.googlesource.com/platform/frameworks/support/+/refs/heads/androidx-main/biometric/biometric/src/main/res/values/devices.xml

[4] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 1041–1057, 2017.

[5] "Android Toast Overlay Attack: "Cloak and Dagger" with No Permissions," Sep. 2017. [Online]. Available: https://unit42.paloaltonetworks.com/unit42-android-toast-overlay-attack-cloak-and-dagger-with-no-permissions/

[6] 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," *MobiSys 2019 - Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 168–179, 2019.

[7] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android This paper is included in the Proceedings of the," *Usenix*, 2015.

[8] Promon, "StrandHogg 2.0 - Android Vulnerability," May 2020. [Online]. Available: https://promon.co/strandhogg-2-0/

[9] Android Developers, "FLAG_WINDOW_IS_OBSCURED," Feb. 2021. [Online]. Available: https://developer.android.com/reference/android/view/MotionEvent#FLAG_WINDOW_IS_OBSCURED

[10] ——, "Camera API," May 2021. [Online]. Available: https://developer.android.com/guide/topics/media/camera

[11] K. Chyn, "Fingerprint should check current client when task stack changes," Mar. 2018. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/09da294c7a3fc65aa3d9b34bb332fa962fa04f4c%5E%21/#F0

[12] Android Developers, "Limited access to sensors in background," Mar. 2021. [Online]. Available: https://developer.android.com/about/versions/pie/android-9.0-changes-all#bg-sensor-access

[13] T. Sutter, "Simple spyware: Androids invisible foreground services and how to (ab) use them," *arXiv preprint arXiv:2011.14117*, 2020.

[14] Y. Zhang, Z. Chen, H. Xue, and W. Tao, "Fingerprints On Mobile Devices: Abusing and Leaking," in *Black Hat USA*, 2015.

[15] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, "Broken Fingers: On the Usage of the Fingerprint API in Android," in *Proceedings 2018 Network and Distributed System Security Symposium*, no. February. Reston, VA: Internet Society, 2018.

[16] L. Wu, B. Brandt, X. Du, and B. Ji, "Analysis of clickjacking attacks and an effective defense scheme for android devices," in *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2016, pp. 55–63.

[17] mxttie, "Answer: Draw Overlay permission for apps installed from Play Store," Apr. 2019. [Online]. Available: https://stackoverflow.com/a/55481376

[18] Android Developers, "NFC Basics," Dec. 2020. [Online]. Available: https://developer.android.com/guide/topics/connectivity/nfc/nfc#dispatching

[19] "Android Intents with Chrome," Jan. 2019. [Online]. Available: https://developer.chrome.com/multidevice/android/intents

[20] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pp. 829–844, 2017.

[21] A. Possemato, D. Nisi, and Y. Fratantonio, "Preventing and detecting state inference attacks on android," in *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[22] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications," Jan. 2020. [Online]. Available: https://github.com/androguard/androguard

[23] "Source code of static analyzer for checking PHYjacking-vulnerable implementation patterns in Android apps," Dec. 2021. [Online]. Available: https://github.com/cuhk-mobitec/PHYjacking-checker

[24] G. Petracca, A. A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," *Proceedings of the 26th USENIX Security Symposium*, pp. 379–396, 2017.

[25] C. Ren, P. Liu, and S. Zhu, "WindowGuard: Systematic Protection of GUI Security in Android," no. March, 2017.

[26] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio, "Clickshield: Are you hiding something? towards eradicating clickjacking on android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1120–1136.

[27] E. Alepis and C. Patsakis, "Trapped by the ui: The android case," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 334–354.

[28] "Attacks and defence on android free floating windows," *ASIA CCS 2016 - Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, pp. 759–770, 2016.

[29] G. S. Tuncay, J. Qian, and C. A. Gunter, "See no evil: Phishing for permissions with false transparency," *Proceedings of the 29th USENIX Security Symposium*, pp. 415–432, 2020.

[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," *MobiSys 2019 - Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 168–179, 2019.

[31] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? Deception and countermeasures in the android user interface," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 931–948, 2015.

[32] J. Liu, D. Wu, and J. Xue, "TDroid: exposing app switching attacks in Android with control flow specialization," pp. 236–247, 2018.

[33] Y. Cai, Y. Tang, H. Li, L. Yu, H. Zhou, X. Luo, L. He, and P. Su, "Resource Race Attacks on Android," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2020.

[34] Android Open Source Project, "Pixel update bulletin—january 2021," Jan. 2021. [Online]. Available: https://source.android.com/security/bulletin/pixel/2021-01-01

[35] "Demo videos of Proof-of-Concept PHYjacking attacks," Oct. 2021. [Online]. Available: https://mobitec.ie.cuhk.edu.hk/phyjacking

Appendix

A. *Experiment on searching for* covering-gadgets

To investigate the feasibility of launching PHYjacking attacks from web pages, we randomly pick 50 popular apps from our database, each with more than 10 million downloads. Then, we follow a three step filtering process:

1) Parse `AndroidManimest.xml` to find all exported (both explicit and implicit) Activities.
2) Resolve the associated theme of the Activity and determine whether the translucent attribute is enabled.
3) Analyze the exported and translucent Activities and identify those whose content is externally controllable.

We automate the first two steps by building an Activity property parser and perform the third step via reverse engineering and manual testing. Among the 50 apps under test, our tool reported 358 Activities that were both exported and translucent, with one app having 24 such Activities and 3 apps having none. Then, given a 10-hour time budget, one of our team members tried to look for Activities whose visual content can be manipulated externally. In the end, he was able to find two manipulatable Activities. One is from a news app with over 10 million downloads, which can be exploited to create a dialog with arbitrary title and description. Another is from an education app with over 90 million downloads where we have partial control of the text displayed. Such *covering-gadgets* can be exploited to launch PHYjacking from web pages. We believe with additional time, adversaries can identify more and better *covering-gadgets*, *e.g.*, where a malicious remote image or web page can be loaded.

B. *More Details on the Static Analysis Results*

The 2024 apps with fingerprint permission and the 6324 apps with camera permission are extracted from a set of 20,000 apps, which are uniformly sampled from the set of apps hosted by several third party app markets including apkpure, an unofficial mirror of Google Play and wandoujia, a major app market in Mainland China. About half of the apps with corresponding permissions were reported as not calling the related physical APIs from any Activities. There are several possible cases: 1) apps indeed never access related sensors/physical interfaces but have permission over-claim ; 2) apps are packed or call the sensor API in native code and 3) the static analyzer fails to link the API call back to an Activity due to incomplete call graph construction caused by algorithm limitations, *e.g.*, in handling implicit calls. Take the `Camera` API for example, for a total of 6324 apps, 3175 apps belong to the first two cases, while 566 apps fall into the third category. We directly exclude all these apps in our evaluation and thus eliminate their effect on the distribution of the dataset. The incompleteness of extracted call graph can also lead to false positives (FP) in the *pause-failure* category. To estimate the FP rate, we manually verified 40 uniformly sampled apps from this category and found only 4 cases of FP. This is expected since sensor-close calls are usually straightforward and can be correctly analyzed. Considered this estimated FP rate (10%), the total percentage of apps with implementation flaws is adjusted downward from 48.9% to 44.1%.

C. *Core implementation of the attack*

The malicious app consists of two major components. One is the main Activity, where some disguising functions are implemented and the attack is only launched when appropriate. Another is the covering Activity, which acts as the delusive covering during the authorization hijacking. Listing 2 and Listing 3 show the code skeleton of two sample target apps. Listing 4 depicts the code for a simplified implementation of PHYjacking against target apps based on the *Race-attack*. The implementation includes additional tricks like killing the target app process to lengthen the race window, as well as the support of optional tapjacking. The same techniques can be used to launch facejacking, voicejacking, *etc.*, on other target apps. Following this template, we have realized several proof-of-concept attacks against various real-world apps. To further demonstrate the viability and stealthy nature of the attack, we also provide a video under the blind test arrangement, where readers can try to distinguish between the normal app and the attack version. All the demonstration videos can be found in [35].

```java
/** FingerprintPaymentActivity.java (app: com.targetapp)
 * Activity for authenticating payment with fingerprint
 * (with android.exported set to true)
 */
public class FingerprintPaymentActivity extends Activity{
  private FingerprintManager fingerprintManager;
    CancellationSignal cancellationSignal;

    @Override
    protected void onResume() {
      super.onResume();
      initFinger();
    }

    @Override
    protected void onPause() {
        super.onPause();
        // app has the correct cancel−on−pause behavior
      cancellationSignal.cancel();
    }

    public void initFinger() {
      cancellationSignal = new CancellationSignal();
        Cipher cipher = generateCipher();
        CryptoObject cryptoObject = new CryptoObject(cipher);
        fingerprintManager.authenticate(cryptoObject,
          cancellationSignal, 0, callback, null);
    }

    AuthenticationCallback callback = new AuthenticationCallback() {
      @Override
      public void onAuthenticationSucceeded(AuthenticationResult
            result) {
          super.onAuthenticationSucceeded(result);
          proceedToPayment(); // where actual payment happens
      }
  };

}
```

Listing 2: The code of the fingerprint-based payment Activity in a sample app to be attacked by the malicious app in Listing 4

```java
/** CameraAuthActivity.java (app: com.targetapp)
 * Activity for camera−based authorization
 * (with android.exported set to true)
 */
public class CameraAuthActivity extends Activity{
  private FingerprintManager fingerprintManager;
    CancellationSignal cancellationSignal;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /* setup preview UI and listener (omitted) */
        takePictureButton.setOnClickListener(new View.
              OnClickListener() {
            @Override
            public void onClick(View v) { takePicture(); }
        });
    }

    protected void takePicture() {
        // get an image from preview frames
        // and proceed to authorization
        // (code omitted)
    }

    @Override
    protected void onResume() {
        super.onResume();
        startPreviewThread();
        openCamera();
    }

    @Override
    protected void onPause() {
        super.onPause();
        closeCamera(); // correct cancel−on−pause mitigation
        stopPreviewThread();
    }
}
```

Listing 3: The code skeleton of the camera-based authorization Activity in a sample app to be attacked by the malicious app in Listing 4

```java
1  /** MainActivity.java (app: com.maliciousapp)
2   * entrence (disguised) Activity of the malcious app
3   * launch the attack when appropriate
4   */
5  public class MainActivity extends Activity{
6
7    /** Entry point when the user run the malicious app
8     * launch PHYjackng once the malicious app is runnning
9     */
10   @Override
11   protected void onCreate(Bundle savedInstanceState) {
12       super.onCreate(savedInstanceState);
13     launchAttack()
14   }
15
16   /** Core attack code
17    * start the target Activity
18    * immediately cover it with the malicious one
19    */
20   public void launchAttack() {
21     // set the target intent, with Activity or deep−link
22     final Intent targetIntent = new Intent();
23     targetIntent.setComponent(new ComponentName("com.targetapp", "
          com.targetapp.TargetActivity")); // see Listing 2 & 3
24     targetIntent.setFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION);
25
26     final Intent coverIntent = new Intent(this, CoveringActivity.class);
27     coverIntent.setFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION);
28
29     Intent[] intents = {targetIntent, coverIntent};
30
31     // [optional trick] kill target first for longer race window
32     ActivityManager am = (ActivityManager) getSystemService(
            Activity.ACTIVITY_SERVICE);
33     if (am != null) { am.killBackgroundProcesses(targetApp); }
34
35     /* launch two intents back−to−back for race−attack */
36     delayedStart(intents, 0); // launch two intents asynchronously
37
38     // startActivities(intents); // or simply use startActivities and
            comment out line 32−36
39   }
40
41   private void delayedStart(Intent[] intents, int delay) {
42     final Intent targetIntent = intents[0], coverIntent = intents[1];
43     startActivity(targetIntent);
44     final Handler coverHandler = new Handler();
45     Runnable coverRunnable = new Runnable() {
46       @Override
47       public void run() { startActivity(coverIntent); }
48     };
49     coverHandler.postDelayed(coverRunnable, delay);
50   }
51 }
52
53 /** CoverActivity.java: malicious covering with translucent property
54  * the translucent property should to be set by adding the follwing line
           in AndroidManifest.xml
55  * android:theme="@android:style/Theme.Translucent.NoTitleBar.
          Fullscreen"
56  */
57 public class CoveringActivity extends Activity {
58   @Override
59   protected void onCreate(Bundle savedInstanceState) {
60     /* the covering needs to have the translucent property
61       set in the AndroidManifest.xml */
62     /* [only for tapjacking] enable tap−through */
63     getWindow().addFlags(WindowManager.LayoutParams.
            FLAG_NOT_TOUCH_MODAL
64       | WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE);
65     // ... other delusive UI setup ...
66   }
67
68   @Override
69   protected void onPause() {
70     /* [optional trick for tapjacking] keep the covering on−top */
71     startActivity(getIntent());
72     super.onPause();
73   }
74 }
```

Listing 4: The implementation of a sample malicious app which leverages the *Race-attack* for PHYjacking