Fingerprint-Jacking: Practical Fingerprint Authorization Hijacking in Android Apps

Xianbo Wang[1], Yikang Chen[1], Ronghai Yang[1,2], Shangcheng Shi[1], Wing Cheong Lau[1]

[1]The Chinese University of Hong Kong
[2]Sangfor Technologies Inc.

Abstract

Many mobile devices carry a fingerprint scanner nowadays. Mobile apps utilize the fingerprint scanner to facilitate operations such as account login and payment authorization. Despite its security-critical nature, relatively little effort has been devoted to the security analysis of fingerprint scanner, especially from the system security aspect. In this paper, we introduce fingerprint-jacking, a type of User-Interface-based (UI) attack that targets fingerprint hijacking in Android apps. We coin the term from clickjacking, as our attack also conceals the original interface beneath a fake covering. Specifically, we discover five novel attack techniques, all of which can be launched from zero-permission malicious apps and some can even bypass the latest countermeasures in Android 9+. Our race-attack is effective against all apps that integrate the fingerprint API. As apps' implementation flaws intensify the fingerprint-jacking vulnerability, we have designed a static analyzer to efficiently identify apps with implementation flaws that can lead to fingerprint-jacking. In our evaluation of 1630 Android apps that utilize the fingerprint API, we found 347 (21.3%) apps with different implementation issues. We have successfully performed proof-of-concept attacks on some popular apps, including stealing money via a payment app with over 100,000,000 users, gaining root access in the most widely used root manager app, and much more. We have also reported related vulnerability to Google, which is identified as CVE-2020-27059 and will be fixed in next Android patch release. Finally, we give guidance to Android app developers for secure fingerprint implementation.

1. Introduction

Due to its high accuracy and low cost, fingerprint scanners are the most commonly installed biometric hardware in smartphones nowadays to facilitate authentication/authorization in mobile apps. Operating systems (OS), as the bridges between hardware and software, should provide an easy way for apps running on smartphones to access the fingerprint scanner. Android, as the most widely used OS in smartphones, does provide a set of Application Programming Interfaces (APIs) dedicated for apps to use fingerprint scanners. With the fingerprint API, developers do not need to worry about the variation in the hardware, even when the fingerprint scanners are from different vendors, or in different types (rear-mounted or in-display).

While convenient, the security of fingerprint authentication is essential as well. In addition to the security of the scanner itself, most smartphones also use Trusted Execution Environments (TEE), a hardware-forced isolated execution environment to enforce the security of fingerprint data processing and storage. Meanwhile, fingerprint security from the software aspect also worths attention, for instance, the authenticity of the user interface (UI). When the fingerprint scanner is waiting for physical touch, the UI must provide the user with a clear context and show exactly the purpose of the fingerprint. Ideally, a secure UI should be enforced by the OS, or even from the hardware level.

When studying the design of the Android official fingerprint API, as well as apps' implementation details, we observe three important issues:

- The `FingerprintManager` API [1], the most widely used fingerprint API in Android, delegates the duty of implementing fingerprint UI to app developers.
- A significant portion of apps implements the fingerprint UI improperly, leaving the app susceptible to UI attacks during the fingerprint authentication process.
- Countermeasures introduced in Android 9+ for mitigating the fingerprint hijacking attack contain flaws and can be bypassed.

Based on these observations, we discovered several attack techniques that enable fingerprint hijacking in apps with the FingerprintManger API. We model such type of attacks as fingerprint-jacking, with the definition: **fingerprint-jacking is a type of attack that the attacker conceals the actual fingerprint UI beneath a deceptive covering, luring the victim to touch the fingerprint scanner and conduct unintended action.** More specifically, we introduce five novel techniques, namely *translucent-attack*, *wakeup-bypass*, *splitscreen-bypass*, *crash-bypass*, and *race-attack*, which make fingerprint-jacking practical: 1) All attacks can be launched from a zero-permission malicious app or even from a web page. 2) Some attacks can bypass the countermeasures introduced in Android 9+, or are still effective even if the apps implement the APIs correctly. The security impact of fingerprint-jacking can range from money stealing to root access gaining in different apps.

To summarize, we have made the following technical contributions:

- We discovered five novel attacks for Android fingerprint hijacking, which are practical as they can be launched from a zero-permission malicious app or a web page.
- We identified flaws in Android's countermeasure, enabling the fingerprint-jacking attack in even Android 9+. We also found a race-condition bug in Android that can make even apps with no implementation flaws vulnerable to fingerprint-jacking.
- We designed a static analyzer to check fingerprint-jacking vulnerable implementations in apps. It automatically identified 347 (21.3%) apps with implementation flaws within 1630 evaluated fingerprint apps, including some popular apps with millions of installs.

2. Background

2.1. Android security mechanisms for fingerprint

Fingerprint sensors are used for authentication and authorization, which are the keys to protect privacy. In most cases, authorization is combined with authentication as only an authenticated user is allowed to grant the authorization. Use cases for authentication include device unlock and account login. Payment and access-granting are common usage scenarios for fingerprint authorization. Due to the essentiality of fingerprint, Android has deployed various mechanisms to protect its security.

2.1.1. 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 installation time. All permissions need to be declared statically in the Android manifest file. To use the fingerprint sensor, the app needs to request the USE_FINGERPRINT permission, which is under the normal level and is auto-granted. Therefore, every app that integrated with fingerprint functionalities will declare this permission in its Manifest file.

2.1.2. Hardware. The permission model for apps can limit their resources access and capabilities. However, for apps running in an insecure environment, *e.g.*, rooted system, 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 separated hardware (*e.g.*, TrustZone for ARM) that can execute code in an isolated world. With TrustZone, the raw fingerprint data is encrypted and stored securely even when the Android OS is fully compromised.

2.2. Fingerprint API in Android

Android released the first official fingerprint API in Android 6.0 (2015), wrapped in the `FingerprintManager` class. This API was the only official API for apps to use the fingerprint scanners across vendors, not until the new `BiometricPropmt` API [2] rolled out in Android 9. The most noticeable difference in this new API is that it provides a unified UI when prompting the user to input their fingerprint. Apart from easing developers to implement the UI of their own, it also reduces the risks of problematic implementation by app developers. However, this API is not backward compatible with devices before Android 9. According to the Android official website [3], until March 2020, more than 60% of devices are still running versions older than Android 9. This is why we find a significant portion of apps in the market are still using the `FingerprintManager` API.

To further mitigate risks and guarantee *What You See is What You Sign* (WYSIWYS), new Android versions also patched the code of `FingerprintManager` API to add some foreground-checking logic to guarantee the app occupying the fingerprint scanner is running in the foreground.

As our attacks only apply to the classic `FingerprintManager` API, we will use it to illustrate the typical usage of fingerprint API. The general usage can be simplified to the following steps:

1) Generate a cryptographic key: apps usually use fingerprint to decrypt a key for the purpose of encryption/decryption or signing/checking. The generated key will only be available to the app after fingerprint authentication.
2) Start listening to fingerprint scanner: call the `FingerprintManager.authenticate()` to listen on the fingerprint scanner.
3) The app can stop the fingerprint listener by calling `CancellationSignal.cancel()`, for example, when the app is switched to background.
4) After the fingerprint is scanned successfully, the callback will be invoked with the unlocked cryptographic key.

2.3. Android Activity lifecycle

Activities are pages with widgets that are shown to users and are basic components of Android apps, with the analogy to windows in PC systems. An Activity, whose lifecycle is modeled and managed by the Android framework, can be created, paused and destroyed. Fig.1 shows the whole picture of the Activity lifecycle model.

When user enters a new Activity, the `onCreate` event will be triggered first, then follows 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.*, it is covered by a translucent Activity, it stays in the paused state instead of being stopped. Finally, when the user brings the background Activity to the foreground, the `onRestart` and `onStart` (only if it's previously stopped), as well as `onResume` events will be triggered sequentially.

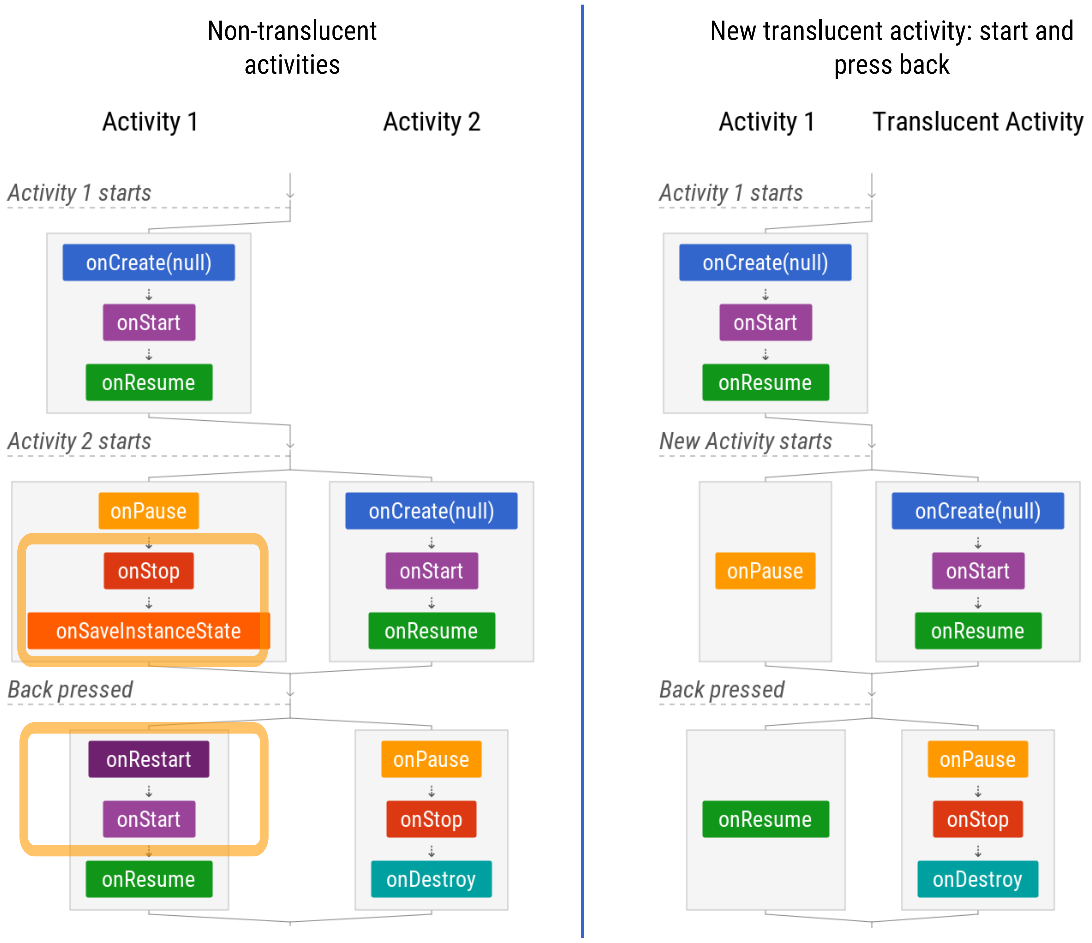

Figure 1. Diagram of Android lifecycle, highlighting the difference between normal v.s. translucent Activity. It shows the lifecycle when Activity 1 brings Activity 2 to the foreground, then Activity 2 was destroyed and Activity 1 gets back to the foreground. This diagram was made by J. Alcérreca [4].

The implementation of the fingerprint authentication process is tangled with the Activity lifecycle. A common practice is to start fingerprint listening in the `onResume` event of the associated Activity and cancel the listener in the `onPause` event. In this way, the app will start listening to the fingerprint sensor once the Activity is shown and will close the sensor once the Activity disappears.

3. Threat Model

The overall model of the fingerprint-jacking attack is similar to other UI redressing attacks in Android. In this paper, we assume that the victim's device is not rooted, otherwise, the attacker can perform more privileged attacks other than UI attacks. We also assume that the attacker has no physical access to the targeting device. Regarding the attacker's capabilities, our attack can be applicable under different settings:

Malicious app attacker. 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 the malicious app requires no permission that users need to explicitly grant, and its capabilities are the same as other normal apps, which are limited by the Android framework. Finally, we 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.

Web attacker. Under this threat model, we assume that there is no malicious app installed on the victim's device. The attacker can only lure the victim to visit a crafted web page in the mobile browser. We also assume that the attacker knows or can guess that 1) the targeted app is installed on the victim device, 2) some assistant apps, which can be some popular benign app, *e.g.*, Facebook, are installed.

4. Fingerprint-Jacking Attacks

In this section, we will introduce five novel fingerprint hijacking attacks. Before that, we will first briefly introduce three existing attacks. Then, we will discuss in detail the key idea of our novel attacks and their advantages. In the end, we will summarize the overall fingerprint-jacking attack process while considering different conditions and scenarios, as well as how to launch the attack from a web page. As an overview, Fig.2 shows the overall taxonomy of related attacks and our new attacks.

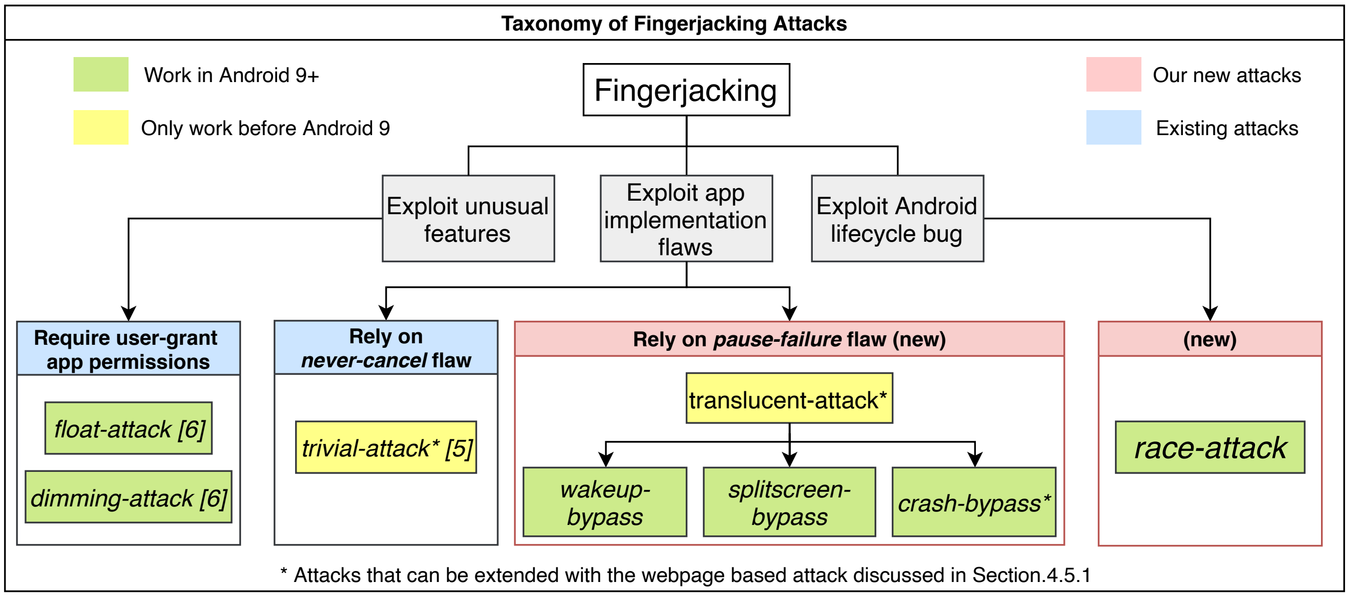
Taxonomy of Fingerjacking Attacks
Work in Android 9+
Only work before Android 9
Our new attacks
Existing attacks
Fingerjacking
Exploit unusual features
Exploit app implementation flaws
Exploit Android lifecycle bug
Require user-grant app permissions
float-attack [6]
dimming-attack [6]
Rely on never-cancel flaw
trivial-attack* [5]
Rely on pause-failure flaw (new)
translucent-attack*
wakeup-bypass
splitscreen-bypass
crash-bypass*
(new)
race-attack
* Attacks that can be extended with the webpage based attack discussed in Section.4.5.1

Figure 2. Overview and taxonomy of new and existing fingerprint-jacking attacks.

4.1. Review of Existing Attacks

The basic idea of the fingerprint-jacking attack is crafting visual content to lure the victim to input his/her fingerprint. We were only able to find two existing works that mentioned related attacks.

The first is the UI confusion attack demonstrated in Black Hat 2015 [5]. The speaker showed the concept of using a lock screen to lure the fingerprint for money transfer. However, at that time, the Android official fingerprint API had not even been released yet and there was not much technical detail about this attack. Based on testing and guessing, we reconstruct the process of this attack is as follows:

1) Initiate the fingerprint listening in the fingerprint app from the malicious app.
2) The malicious app starts a new Activity to cover the fingerprint app and trick the victim to touch the fingerprint scanner.

We will call this attack the *trivial-attack* from now on. For this attack to work, the following implicit requirements have to be satisfied:

- The fingerprint app will not cancel the fingerprint authentication when the corresponding Activity is switched to the background.

- Android OS allows apps running in the background to continue occupying the fingerprint scanner.

In current practice, with the Android fingerprint API, both requirements are not trivial to satisfy. We will address them in the discussion of our new attack techniques.

A more recent study was performed by [6] on Android 7. The authors proposed fingerprint UI attacks with the floating window or the trick of dimming screen, both require some special permissions being granted by the user to the malicious app. We will refer to them as *float-attack* and *dimming-attack* respectively. In our new attacks, we eliminate the permission requirement of the malicious app to achieve better practicability.

4.2. Exploiting the Translucent Activity: Common Failure in Pause Handling

The *trivial-attack* assumes that the fingerprint app does not cancel listening even when switched to the background (we call this behavior ***never-cancel***), which unluckily is rare in practice. However, we observed that there are a significant portion of apps (data will be shown in Section.5) that have fingerprint cancellation logic but implemented incorrectly. More specifically, our key observations are as follows:

- Many apps only cancel the fingerprint authentication in `onStop` event of the corresponding activity instead of the `onPause` event, we name this implementation pattern as ***pause-failure*** throughout this paper.
- In the Android system, when an activity with translucent property covers the original activity, only the `onPause` event of the underneath activity is triggered.

With these observations, we propose an attack that uses the translucent Activity to cover the fingerprint Activity, which can avoid interrupting the fingerprint authentication. We name this attack as *translucent-attack*. The `translucent` property of an Activity can be declared with the `<item name="android:windowIsTranslucent">true</item>` in the theme definition without any permission. Fig.3 illustrates this attack with comparison to the *trivial-attack*.

4.3. Bypass the Patch Introduced in Android 9

The *translucent-attack* as well as the *trivial-attack* only work on or before Android Orea (8 & 8.1). We found that a patch [7] was added to the `FingerprintManager` API since Android 9 to mitigate the fingerprint hijacking risk caused by *pause-failure* from OS level. The logic of the patch is to perform a consistency check between foreground Activity and the fingerprint Activity whenever the Activity stack is changed (*e.g.*, when an Activity is put above another).

However, we soon realized that the patch itself contains flaws. Basically, the seemingly invincible mitigation overlooked some corner cases: if Activity stack change never happened, the underneath Activity can continue listening on the fingerprint scanner even if it is not in the foreground. With this in mind, the challenge becomes: how can we cover an Activity without changing the Activity stack.

4.3.1. *wakeup-bypass*.

One corner case the mitigation overlooked is when the device wakeups. The attacker can cover the fingerprint activity with malicious activity before the device sleeps. After wakeup, both activities are resumed and the underneath one is then paused. If a fingerprint Activity automatically starts listening in the `onResume` event (we call this ***auto-resume*** pattern) and has the *pause-failure* flaw, the fingerprint can work in the background, avoiding any stack change and thus invalidate the checking. This process is illustrated in Fig.4.

In practice, waiting for the device to sleep after setting up the fingerprint and covering Activities is less practical, 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. When the victim subsequently wakes up the device, fingerprint-jacking happens.

To summarize, the wakeup-bypass can work in all Android versions, but with the following assumptions:

- The attacker can set up the desired Activity stack before the device sleeps, and wait for the victim to wakeup the device.
- The app's implementation must have both the *auto-resume* and the *no-pause* patterns.

Note that the *auto-resume* assumption can be eliminated by combining the touchjacking attack, as we discuss later in Section 4.5.2.

4.3.2. *splitscreen-bypass*.

This can be regarded as a variant of the *wakeup-bypass*. In Android 9, the multi-window mode allows apps to run in a split-screen. Note that although multiple Activities are visible to the user in the multi-window mode, only one window will be active at a time and only the top Activity in that window will be in the resumed state. We find that there are two scenarios that the active window will be switched from one to another.

- When the user taps in the inactive window, it will become active and the top Activity in it will be resumed.

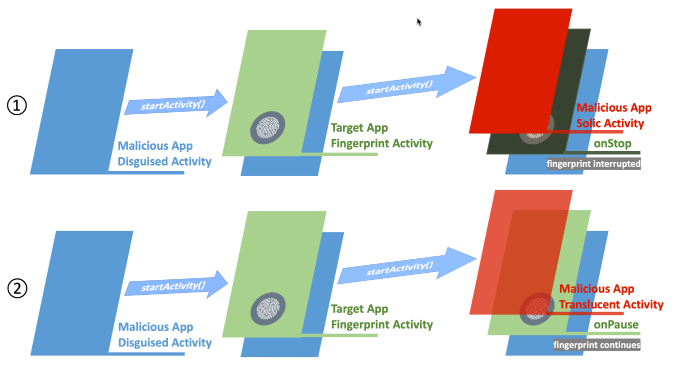

Figure 3. Illustration of *translucent-attack*, with comparison to *trivial-attack*. If the app cancels the fingerprint in the `onStop` method, *trivial-attack* in ① will not work, but our new *translucent-attack*as shown in ② will work.

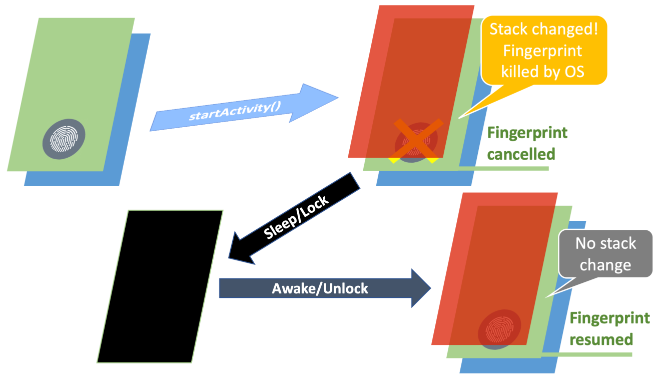

Figure 4. Illustration of the *wakeup-bypass* in Android 9 and later. After the device wakeup, there is no stack change, so there is no checking when the background Activity starts fingerprint listening.

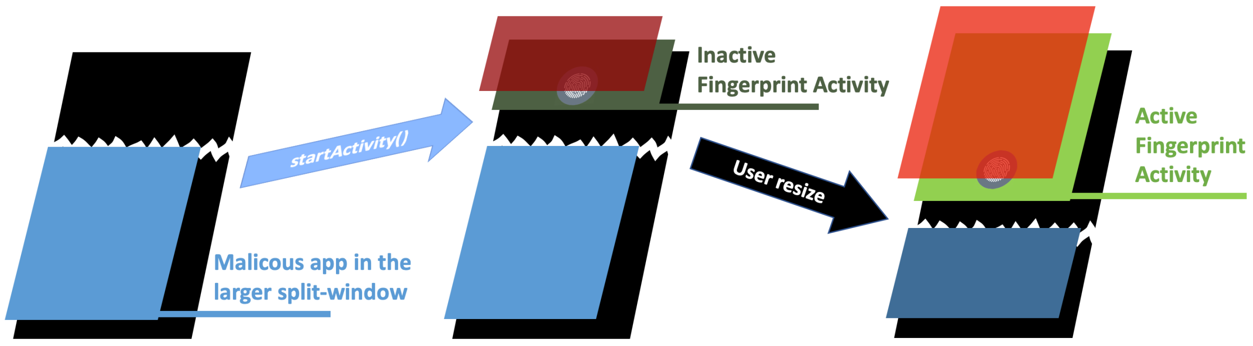
startActivity()
Malicous app in the larger split-window
Inactive Fingerprint Activity
User resize
Active Fingerprint Activity

Figure 5. Illustration of *splitscreen-bypass*. When the user resizes the upper window from 1/3 to 2/3 of height, the placed Activity in the upper window receives `onResume` event and then fingerprint-jacking happens.

- When the user resizes the window, the window with size more than a half will become active regardless of which window the user previously was interacting with.

When these two cases happen, there will be no change in Activity stack, while it changes the Activity status and triggers events. Thus, we can create an attack similar to the *wakeup-bypass* with the scenario of two-split windows. One is the active window with the malicious app running (*malicious-window*), another is the adjacent window where the victim fingerprint app will be put into (*victim-window*).

1) The malicious app in *malicious-window* sets up the desired Activity stack into 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 the *victim-window* or resize the window so the *victim-window* gets larger than half-screen.

This attack not only can bypass the countermeasure in Android 9 but also works in all other versions with multi-window support (Android 7+).

4.3.3. *crash-bypass*. The wakeup-bypass and split-bypass are both found by analyzing the mitigation mechanism in the original Android 9. Meanwhile, by conducting black-box testings on a few vendor-modified firmware (ROM), we also find a bypass that only works in a specific ROM, namely MIUI 11 [8], a ROM made by Xiaomi based on Android 9. We noticed that the same foreground-checking countermeasure is included. However, with the following steps, we can magically resurrect the fingerprint-jacking attack.

1) From the malicious app, invoke the fingerprint Activity in the target app.
2) The malicious app then starts a chosen Activity in some other installed app with malformed data to cause an immediate crash of that Activity.
3) After a few seconds, launch a translucent covering Activity.

TABLE 1. IMPLEMENTATION AND ENVIRONMENT ASSUMPTIONS ON DIFFERENT FINGERPRINT-JACKING ATTACKS

Attacks		Implementation flaw dependency		System requirement	Implementation pattern dependency		Attacker capability requirement	
		Rely on *never-cancel*	Rely on *pause-failure*	Require Android<9	Require *auto-resume*	Require *no-button*[1]	Require a malicious app	Malicious app's permission
Known attacks	*trivial-attack*[4][5]	✓	✓	✓	✗	✗	✗	None
	float-attack [6]	✗	✗	✓[3]	✗	✗	✓	SYSTEM_ALERT_WINDOW[3]
	dimming-attack [6]	✗	✗	✗	✗	✓	✓	WRITE_SETTINGS[3]
New attacks	*translucent-attack*	✗	✓	✓	✗	✗	✗	None
	wakeup-bypass	✗	✓	✗	✓	✓	✓	None
	splitscreen-bypass	✗	✓	✗	✓	✓	✓	None
	crash-bypass	✗	✓	✗[2]	✓	✓	✗	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 touchjacking.
[2] *crash-bypass* requires MIUI, a vendor customized ROM, and it doesn't work in the native Android.
[4] As details of the attack demonstrated in [5] were not given, some conditions here for *trivial-attack* are based on our own testing.

Having no access to the source code of the ROM, we are not able to find the exact reason that makes the bypass work at the current stage. Based on collecting clues in the haystack of logs, we suspect the cause being some proprietary game boosting service in the ROM interferes with the Activity lifecycle.

4.4. The most powerful attack: *race-attack*

All the attacks mentioned up to now rely on the assumption that the app fails to cancel the fingerprint properly. While this flawed implementation pattern is considerably common, it seems that once developers do things right, the whole fingerprint-jacking thing is no longer a concern. Unfortunately, with the race-attack we discovered, this is not true.

```
1   # Failed to trigger race condition:
2
3   20:38:55.065 fjLog: Activity.onCreate: <VictimActivity>
4   20:38:55.085 fjLog: Activity.onStart: <VictimActivity>
5   20:38:55.085 fjLog: Activity.onResume: <VictimActivity>
6   20:38:55.155 fjLog: Activity.onPause: <VictimActivity>
7   20:38:55.189 fjLog: Activity.onStart: <MaliciousActivity>
8   20:38:55.190 fjLog: Activity.onResume: <MaliciousActivity>
9
10  # Race condition triggered:
11
12  20:39:25.314 fjLog: Activity.onCreate: <VictimActivity>
13  20:39:25.378 fjLog: Activity.onStart: <VictimActivity>
14  20:39:25.379 fjLog: Activity.onResume: <VictimActivity>
15  20:39:25.454 fjLog: Activity.onCreate: <MaliciousActivity>
16  20:39:25.477 fjLog: Activity.onStart: <MaliciousActivity>
17  20:39:25.480 fjLog: Activity.onResume: <MaliciousActivity>
```

Listing 1. Activity events log when *race-attack* failed and succeeded.

We found that if two Activities are started with a very short delay (can be 0ms) in between, the Activity lifecycle can go into some limbo state. Listing.1 shows the detailed event log when trying to trigger the race condition bug. When it failed to trigger, the underneath *VictimActivity* receives the `onPause` event as expected (line 6). When it succeeded, the underneath *VictimActivity* stays in the resumed state. Regardless of the design in Android that only one Activity can be in the resumed state at a time, we notice that two Activities appear both in the resumed state when the race condition happens. With more experiments, it only requires the following to stably reproduce this limbo state:

- Use `StartActivities(Intent[])` API to launch the two Activities together.
- The second (upper layer) Activity is translucent.

This race-condition bug not only creates an Activity lifecycle limbo so that the attack does not rely on the app's implementation, but it also invalidates the foreground-checking mitigation of fingerprint API. It kills two birds with one stone! With this *race-attack*, basically, any apps on all Android versions that use the `FingerprintManager` API are susceptible to the fingerprint-jacking attack.

4.5. Combination Attack for Better Practicability

In practice, the attacker needs to adjust the attack process by considering the target app and the environment. To increase the chance, several aforementioned attack techniques can be combined, *e.g.*, while actively launching the *race-attack*, the attacker can in the meantime passively wait for the device to sleep and set up the *wakeup-bypass*. Table.1 summarizes and compares all mentioned attacks with their conditions on app implementation, Android version, and required permissions.

4.5.1. Launching fingerprint-jacking from a web page. Under some settings, the fingerprint-jacking attack can be launched solely from a malicious web page. To perform this attack, the attacker first needs to find a way to initiate the fingerprint authentication process from a web page. On the Android system, this can be done by using the remote app linking mechanism [9], *e.g.*, deep links (`payapp://dopayment?mode=fingerprint`), which links to the Activity that associates with the fingerprint authentication process.

The more challenging part is to perform the malicious UI covering from a web page. This requires the attacker to find an activity 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 as *covering-gadget*. The attacker can find *covering-gadget* in the target app itself or in some other benign apps that already installed on the victim's device. It often appears in apps that load external websites in WebView.

```
1  <html>
2  <a href=”targetapp://fingerprint” onclick=intent()>CLICK ME</a>
3
4  <a href=”app1://webview/?url=http://evil.com” id=”a1”></a>
5  <a href=”app2://fullscreen/?color=black” id=”a2”></a>
6  <a href=”app3://evil.com” id=”a3”></a>
7  <script>
8  function intent(e){setTimeout(
9      function(){
10         document.getElementById(”a1”).click();
11         document.getElementById(”a2”).click();
12         document.getElementById(”a3”).click();
13     }, 0);
14 }
15 </script>
16 </html>
```

Listing 2. Simplified web page code for the fingerprint-jacking attack

In Listing.2, we show the simplified source code of the malicious web page for launching this attack. There are two sets of deep links on this web page, corresponding to the two steps in the attack: 1) Invoke the fingerprint app: the CLICK ME link on line 2 points to a deep link that can initiate the fingerprint authorization process in the target app. 2) Launch *covering-gadget*: another three links follow (line 4-6) are deep links that point to potential *covering-gadgets* in different benign apps. We use JavaScript to try each of them until getting a hit (line 8-14).

When a victim visits this page, the web page will launch the fingerprint authentication activity and the covering activity subsequently. In practice, the attacker is likely to be unsure about whether one particular *covering-gadget* can be used, since that particular app may not be installed on the victim's device. However, the attacker can include as many potential *covering gadgets* as he wants to increase the success rate of the attack.

4.5.2. Combining with touchjacking. The first step of launching the fingerprint-jacking attack is invoking the fingerprint authentication process in the target app externally from the malicious app. In some apps, this is not possible and screen interactions (*e.g.* button clicks) are required to trigger fingerprint authentication. As a workaround, we can conduct the touchjacking attack [10] with the translucent Activity to lure for button taps. In addition, for apps without the *auto-resume* feature (as defined in Section.4.3.1), touchjacking can make wakeup-bypass and *splitscreen-bypass* work again.

5. Static Analyzer and Evaluation Results

To facilitate checking fingerprint-jacking vulnerability in a large set of apps, especially for those with implementation flaws, we build an automatic static analyzer for Android apps. Briefly, the static analyzer is designed based on the logic: From the located `authentication()` call in the code, it extracts the call chain backward to find the Activity that initially invokes it. Then it analyzes the `onPause` method of the Activity and checks if any fingerprint cancellation logic exists.

To evaluate the impact of the fingerprint-jacking, we collected 2024 APKs that declared USE_FINGERPRINT permission and ran 8 analyzers 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 average memory consumption is 1201 MB.

The detailed results of the test are shown in Table.2. Out of 2024 apps, the analyzer cannot locate the `authenticate` call in 394 (19.5%) apps. Some apps may declare permission without actual implementation. A few false-negative cases also contribute to it, *e.g.*, apps that invoke fingerprint API with native code instead of JAVA and apps are protected with packing. We exclude these apps for further analysis. In the remaining 1630 apps, we failed to associate the fingerprint

TABLE 2. Evaluation Result with the Static Analyzer on 2024 Apps

# of collected apps	# of analyzed apps (API call found)	No API-Activity links	No implementation flaw	Found implementation flaw = *never-cancel* + *pause-failure*
2024	1630	920	363	347 = 68 (19.6%) + 279 (80.4%)
Average analysis time	Max analysis time	Average memory consumption	Max memory consumption	
76.8 seconds	11.1 hours	1.2 GB	7.7 GB	

authentication call with any Activity in 920 (56.4%) of them. Apart from the limitation of the analyzer, there are also cases that apps include many third-party libraries, some of which may contain fingerprint related codes, but are never actually used. Eventually, there are 710 apps found initiating fingerprint authentication from some activities. Among them, only 363 (51.1%) apps implement the cancellation properly, all remaining 347 (48.9%) apps contain implementation flaws that can lead to fingerprint-jacking, and within the vulnerable apps, 279 (80.4%) belong to the *pause-failure* case. This indicates that a significant portion of developers cannot implement the cancellation of fingerprint authentication correctly, especially when it involves the pause event in the Android lifecycle.

During manual verification of some vulnerable apps, we found some interesting and impactful cases. One of them is a popular mobile payment app with 100,000,000+ installs, in which the payment process can be invoked externally from an app or even a web page. This app contains the *pause-failure* flaw, making it vulnerable to most fingerprint-jacking attacks we discussed. Money stealing can be conducted by initiating a payment request to the attacker owned account and lure the victim to authorize using fingerprint. The whole attack process can be very smooth and is unlikely to be noticed by the victim. Another is Magisk Manager, the most popular open-source root manager app in Android with 50,000,000+ downloads [11]. For every app that requests the root privilege, Magisk will ask for the user's authorization, which can be done by scanning the fingerprint. We found its implementation contains the *never-cancel* flaw, which means in older Android versions, the fingerprint authorization works even when the app is switched to the background. A malicious app can easily request the root privilege and then launch the fingerprint-jacking attack to lure the victim's authorization. Note that at the time of writing, new Magisk versions have migrated to the new `BiometricPropmt` API and is no longer vulnerable to the attacks we discussed.

6. Securing Fingerprint API and App Implementations

We have reported the race-condition and mitigation bypass issue we discovered to Android in June 2020. During the triaging process, we confirmed that the issue could be reproduced even on Android 11. In November 2020, Google confirmed to release a patch for this issue in January 2021 Android Security Bulletin and assigned CVE-2020-27059 to it.

Before the official patch is released, it is more important for app developers to protect their apps from the fingerprint-jacking attack. We strongly recommend Android app developers to do the following:

- Migrate to the new `BiometricPropmt` API and deprecate the classic `FingerprintManager` API. The new API comes with a secure UI that we believe is immune to the fingerprint-jacking attack.
- If backward incompatibility is the concern, use Google's support library `androidx.biometric` [12]. It handles the compatibility problem and provides a unified UI. It survived our preliminary testing on all Android versions.
- Use a third-party fingerprint library carefully. We also tested several unofficial fingerprint libraries and found a number of them are vulnerable by default to the fingerprint-jacking attack.
- If developers have determined to use the `FingerprintManager` API and implement their own UI, they MUST explicitly cancel the fingerprint authentication process in the onPause event of corresponding Activity. Besides, they SHOULD put a confirmation button or require other additional user interactions before starting fingerprint scanning.

7. Related Work

UI attack is a widely explored topic in computer software security. Since emerging of smartphones and the Android system, the UI attack was again put under spotlights, but on a new platform. The unique model and mechanism of the Android UI system created some novel UI attacks in recent years. [13] studied the Android task state transition model and discovered several task hijacking attacks. In our fingerprint-jacking attack, the action of creating a malicious activity to cover the benign fingerprint app is also under this task hijacking category. Interestingly, the method we used to conduct the task hijacking is not in the list of their discoveries and might be new. Both [14] and [15] proposed Android UI attacks with malicious overlay, one with translucent Activity and another with floating windows. They managed to escalate the power of UI redressing attacks and demonstrate complex attacks like keyboard logging. Our attacks, unlike all traditional UI attacks that targeting things on the screen, aim at the fingerprint scanner, resulting in different threat models. [16] performed a comprehensive study on the possibility of launching Android UI attacks from web pages. Our idea of launching fingerprint-jacking from web pages and utilizing existing apps was largely inspired by this paper.

The fingerprint support in Android devices became prevalent more recently and it is reflected in the scarce of related security research. Yet, we find that two works mentioned UI attacks when studying fingerprint security. As far as we know, [5] is the first to propose the UI confusion attack against the fingerprint scanner with a working demonstration. At that time, Android just released its official fingerprint API and was not included in their research, not to mention those countermeasures added recently. The most recent related study was performed by [6] on Android 7. The authors proposed a workaround to bypass the countermeasure against the fingerprint UI attack with a floating window or screen dimming, either of which requires some special permissions. While [15] showed that the permission for the floating window was automatically granted to apps in Google Play, this seems no longer true since Google adjusted its policy [17]. Our work not only introduces five new attack techniques (except trivial-attack) but also shows the possibility to launch the attack with a zero-permission malicious app or even a web page, demonstrating the practicability of the fingerprint-jacking attack.

8. Conclusion

In this paper, we proposed the fingerprint-jacking attack with five newly introduced practical attack techniques, including attacks that exploit common implementation flaws in Android apps, attacks that can bypass the latest mitigations added since Android 9, and a powerful race-condition attack that applies to most scenarios. With the automatic analyzer and manual testing, we demonstrate the prevalence of the fingerprint-jacking attack as well as its critical security impacts.

References

[1] "FingerprintManager: Android Developers," Dec. 2019. [Online]. Available: https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager

[2] "BiometricPrompt: Android Developers," Mar. 2020. [Online]. Available: https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt

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

[4] J. Alcérreca. (2019, Jan.) The Android Lifecycle cheat sheet. [Online]. Available: https://github.com/JoseAlcerreca/android-lifecycles

[5] H. a. Yulong'Zhang, 'Zhaofeng'Chen, "Fingerprints On Mobile Devices: Abusing and Leaking," in *Black Hat USA*, 2015.

[6] 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.

[7] K. Chyn. Fingerprint should check current client when task stack changes.

[8] "MIUI," Dec. 2019. [Online]. Available: https://en.miui.com/

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

[10] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *International Symposium on Foundations and Practice of Security*. Springer, 2012, pp. 227–243.

[11] "Magisk Manager," Dec. 2019. [Online]. Available: https://magiskmanager.com/

[12] "Biometric: Android Developers," Dec. 2019. [Online]. Available: https://developer.android.com/jetpack/androidx/releases/biometric

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

[14] 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.

[15] 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.

[16] 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.

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