Attacks and Defence on Android Free Floating Windows

Lingyun Ying*
lingyun@iscas.ac.cn

Yao Cheng†
ycheng@smu.edu.sg

Yemian Lu*
luyemian@tca.iscas.ac.cn

Yacong Gu*
guyacong@tca.iscas.ac.cn

Purui Su*
purui@iscas.ac.cn

Dengguo Feng*
feng@tca.iscas.ac.cn

* Institute of Software, Chinese Academy of Sciences, Beijing, P.R.China
†School of Information Systems, Singapore Management University, Singapore

ABSTRACT

Nowadays, the popular Android is so closely involved in people's daily lives that people rely on Android to perform critical operations and trust Android with sensitive information. It is of great importance to guarantee the usability and security of Android which, however, is such a huge system that a potential threat may arise from any part of it. In this paper, we focus on the Free Floating window (FF window) which is a category of windows that can appear freely above any other applications. It can share the screen space with other FF windows, dialogs, and activities. An FF window is flexible in both its appearance and behaviour features. We analyse the behaviour features of FF windows, including the priority in display layer and the capability of processing user-generated events. Three types of attacks via FF windows with delicate design in their appearance and behaviour features are demonstrated, i.e., DoS attack against Android system, GUI hijacking by targeting overlap, and input inference using FF windows as a side channel. To address the threat caused by FF windows, we design a priority framework for FF windows, which protects a sensitive activity/FF window declared by developers from being attacked by any malicious FF windows. A complementary solution is proposed to mitigate the confusion attack from malicious activities. Finally, we provide Android with suggestions on how to manage FF windows.

Keywords

Android, free floating window, DoS attack, GUI hijacking, input inference.

1. INTRODUCTION

Nowadays, smartphones are closely involved in users' daily lives. Users rely on smartphones and the applications on smartphones to process their personal issues, e.g., managing financial accounts. The more users rely on mobile smartphones, the more malwares emphasize on these mobile systems [1]. Thus, the reliability and usability of smart systems are of great importance to users. Among various mobile systems, Android dominates the market with an 82.8% share in 2015 Q2 [2].

Android is an operating system that even more than one application can run concurrently in background, each time there is only one application interacting with users. However, there is no explicit information showing to users about the active user interface and the application it belongs to. Therefore, it can be told which application is interacting with users only according to the application's appearance visible on screen. Users normally cannot distinguish two applications with similar user interfaces [3, 4].

The difficulty in differentiating applications leads to two types of attacks. One is touchjacking [5, 6] which is derived from the clickjacking on desktops. Touchjacking blinds users from seeing the application they are interacting with. It seduces users to click the designated position by showing interesting clickable items on screen, i.e., bait, whereas the activity that users are indeed dealing with is the activity covered by the bait, e.g., an activity downloading malware or costing money. Another type of attack is GUI confusion attack [3]. GUI confusion also prevents users from realizing which application they are interacting with. The difference is that it draws user interface elements over the target activity or switches to another similar activity without the user's knowing. The application interacting with a user is the one on top of victim activity, which is achieved by adding a deceptive window or starting a malicious activity.

We notice that there is a category of windows which can appear above any other applications. They can share screen space with other windows, dialogs, and activities. We call them Free Floating windows (FF windows). The existence of an FF window depends on the existence of the process it belongs to. This is different from an activity with `windowIsFloating` flag, whose existence depends on the existence of its parent window. "Free" not only means that the FF window can float above any application no matter the covered window is from the same or different application, but also indicates the flexibility in defining its appearance and behaviour features. An FF window has a set of attributes that decide the way it acts, such as whether it accepts or declines the events generated by a user. The well-known alert window which requires permission `SYSTEM_ALERT_WINDOW` is a type of FF window. There are a lot of other floating windows with different features which are obscure to the public.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897897

We investigate into FF windows in this paper, especially emphasize their inconspicuous behaviour features. Based on the comprehensive understanding of FF windows, we analyse their potential threats to Android system and to other Android applications running on the same platform. We show that attacks using FF windows can obtain users' system unlock credentials, hijack the security alerts, get the implication about user input, and even perform DoS attack against Android system. Finally, the corresponding solution is discussed. In summary, we make the following contributions.

- To the best of our knowledge, we are the first to systematically analyse the attributes of Free Floating windows (FF windows) and behaviour features due to such attributes, including types and flags.
- Three types of attacks are presented based on the behaviour features we analyse, i.e., DoS attack against Android systems, GUI hijacking by targeting overlap, and input inference using FF windows as a side channel. The applications carrying out the above attacks can pass the audit of Google bouncer[7] and are published to the Google Play [8, 9].
    1. The DoS attack can disable any Android device from version 2.3 to the latest 6.0 without any permission, which we have updated to Android and received the confirmation.
    2. The GUI hijacking can mislead a user using FF windows with delicate design in appearance and behaviour features, e.g., blocking the security alert and obtaining the unlock pattern of Android system or a third-party application.
    3. FF windows can be used to provide side channel information to the inference of user input. The experiments demonstrate that it significantly increases the possibility of the password-guessing attack.
- We propose and implement 1) a priority framework for FF windows to defend the above attacks from FF windows, and 2) a complementary solution to confusion attack based on activities. We also discuss how to restrict the capability of FF windows and manage the use of FF windows.

The remaining of this paper is organized as follows. Section 2 introduces the free floating window and analyses its behaviour features. Section 3 demonstrates three types of attacks via FF windows, including DoS attack via FF windows, GUI hijacking by targeting overlap, and using FF windows as a side channel. Later in Section 4, we discuss the causes of the attacks and propose a defence framework to mitigate the threats from FF windows. Some suggestions are also provided in this section. Section 5 summaries the related work. Finally, we conclude our paper in Section 6.

2. FREE FLOATING WINDOWS

Before we present an in-depth look at FF windows, we first introduce the concept of "window." Activity, which is known as one of the most important components on Android, draws its user interface in a window given by Android. Each window has a single view hierarchy containing `Views` [12] which are interactive UI elements. Different from activities, windows are managed dependently by `WindowManager`. Multiple windows can share space in one screen, e.g., the status bar, the navigation bar, and the Home screen.

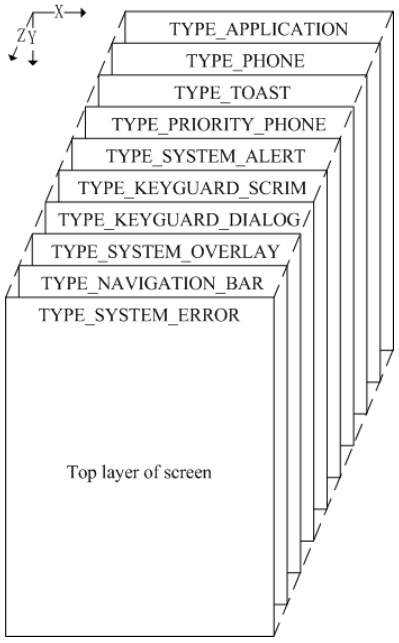


Figure 1: The priority of FF windows with different `WindowManager.LayoutParams.type` values. The more front the layer is, the closer it is to the device user.

An FF window is a developer-defined window. Its appearance and event processing capability can be defined by developers. FF windows with different attributes present different behaviours during their display, e.g., some of them can process user events while some cannot.

There are two approaches to launch an FF window. One is to define an FF window using `WindowManager`, and launch it by calling `WindowManager.addView()`. The other way is to create an FF window by `AlertDialog`, and show it on screen by `AlertDialog.show()`. The behaviour feature of an FF window is set by `WindowManager.LayoutParams` no matter whether the FF window is launched through `WindowManager` or `AlertDialog`. Among all the attributes in `WindowManager.LayoutParams`, the most influential attributes are `WindowManager.LayoutParams.type`, which regulates the priority in display layer on screen, and `WindowManager.LayoutParams.flags`, which defines the capability of event processing.

`WindowManager.LayoutParams.type` has a set of values which define the priority that an FF window enjoys when being placed along the Z-axis (perpendicular to the device screen towards the device user). We investigate into those values, and summarize the priority of some important types. The more prior it is, the closer it is to users. Figure 1 shows the priority of FF windows with different `WindowManager.LayoutParams.type` values on Android 5.0. For an FF window of certain type, it can overlap an FF window of any type behind it, e.g., an FF window with

WindowManager.LayoutParams.type	Focusable?	Required Permission
TYPE_PHONE	Yes	SYSTEM_ALERT_WINDOW
TYPE_TOAST	Yes	-
TYPE_PRIORITY_PHONE	No	SYSTEM_ALERT_WINDOW
TYPE_SYSTEM_ALERT	Yes	SYSTEM_ALERT_WINDOW
TYPE_SYSTEM_OVERLAY	No	SYSTEM_ALERT_WINDOW
TYPE_SYSTEM_ERROR	Yes	SYSTEM_ALERT_WINDOW

Table 1: The `WindowManager.LayoutParams.type` values which can be used by a third-party application. Types are listed ascendingly in priority order.

Markets	Google Play	Wandoujia [10]	Anzhi [11]
# of applications	21,491	17,046	25,731
# of applications requiring permission SYSTEM_ALERT_WINDOW	1,001	6,716	8,622
Ratio	4.66%	39.40%	33.51%
Collection period	2015.05.06-2015.05.08	2015.09.19-2015.10.15	2015.09.22-2015.10.15

Table 2: The usage of `SYSTEM_ALERT_WINDOW` permission in samples from three popular Android markets.

`TYPE_SYSTEM_ERROR` can appear above all the other types in Figure 1. For FF windows of the same type, the latter created one is on top of the former created one along the Z-axis. It is worth noting that not all of these types in Figure 1 can be used by third-party applications, and there are other types with priority higher than `TYPE_SYSTEM_ERROR` are specially for system use. Table 1 lists the types that can be used by a third-party application in the ascending order of priority in appearance along Z-axis. The type with highest priority that a third-party application can use is `TYPE_SYSTEM_ERROR`. Except for `TYPE_TOAST` requiring no permission, the other types require the calling application to have `SYSTEM_ALERT_WINDOW` permission which is a common permission requested by averagely around 25% of the random samples from the Google Play and two biggest application markets in China (Table 2).

`WindowManager.LayoutParams.flags` is another most influential attribute of FF windows, which defines whether an FF window can get the system focus or process the events. Some `WindowManager.LayoutParams.flags` values related to the capability of processing user-generated events are shown in Table 3. Flags can be set individually or jointly. If no flag is set, an FF window acts like an activity. It consumes all the events generated within its window range. The covered activity cannot respond to the click events, even it is partially visible outside the FF window. Differently, some FF windows do not respond to the click events inside the window range, e.g., an FF window with `FLAG_NOT_TOUCHABLE` passes all the events to whatever touchable windows behind it. Moreover, complicated behaviour features can be defined by assistant flags which can be set jointly with the major flags and do not change the behaviour defined by a major flag, e.g., `FLAG_WATCH_OUTSIDE_TOUCH`. An FF window with `FLAG_WATCH_OUTSIDE_TOUCH` is able to process an `ACTION_OUTSIDE` event enveloping an `ACTION_DOWN` event generated outside the window concurrently with the touched window. When there is an `ACTION_DOWN` event generated outside the window with `FLAG_WATCH_OUTSIDE_TOUCH`, Android system first passes the event to the touched window, then re-envelops the event excluding the click position into an `ACTION_OUTSIDE` event and sends it to the above FF window with `FLAG_WATCH_OUTSIDE_TOUCH`.

We can see that the flexibility in the attribute setting of `WindowManager.LayoutParams.type` and `WindowManager.LayoutParams.flags` enriches the diversity of FF windows' behaviours, which however, also gives attackers opportunity to carry out various attacks taking advantage of FF windows.

3. ATTACK ANALYSIS

Based on the behaviour features we analyse, we identify three practical attacks based on FF windows, i.e., DoS attack via FF windows, GUI hijacking by targeting overlap, and using FF windows as a side channel. These attacks can be launched by a single application installed on the target device without root privilege. From the perspective of attackers, we demonstrate the design of these attacks along with their potential threats to Android system, Android applications, and Android users.

3.1 DoS Attack via FF Windows

DoS is to disable a target object to its intended users by occupying a critical mutually-exclusive resource or flooding too many requests which are out of the target's capability. DoS is fundamentally to monopolize or exhaust the indispensable resources, without which, the target object can no longer provide intended services. It is an efficient way to protect system resources by restricting the number of resources that an object can request, e.g., Android limits the number of Toasts [13] that an application can invoke within a short period to 50. Unfortunately, we discover that there is no limitation in the usage of FF windows.

We manage to flood Android by invoking a number of FF windows. To test the DoS attack, we develop an application called "Popper" which does nothing but pop up a large number FF windows (e.g., 1,200 FF windows) with type `TYPE_TOAST` in a short time. Popper is tested against the available Android versions from 2.3 to latest 6.0, which cover 99.8% of current Android devices according to the official distribution statistics [14]. In most cases, Popper causes reboot of Android system soon after its

WindowManager.LayoutParams.flags	The window responding to click events within the FF window area	The window responding to click events outside the FF window area	Does the FF window pass down the key events from Back button?
FLAG_NOT_FOCUSABLE*	The FF window	The covered window	Yes
FLAG_NOT_TOUCHABLE	The covered window	The covered window	No
FLAG_NOT_TOUCH_MODAL	The FF window	The covered window	No
FLAG_NOT_FOCUSABLE| FLAG_NOT_TOUCHABLE	The covered window	The covered window	Yes
FLAG_NOT_TOUCHABLE| FLAG_NOT_TOUCH_MODAL	The covered window	The covered window	No
Not with any flag above	The FF window	The FF window	No

Table 3: The behaviour features of FF windows with different `WindowManager.LayoutParams.flags` values when they appear on top of another window. In the table, "the covered window" is covered by "the FF window" with corresponding flags. *If `FLAG_NOT_FOCUSABLE` is set, `FLAG_NOT_TOUCH_MODAL` is set implicitly.

running. However, sometimes Android kills Popper before Popper pops up enough FF windows to exhaust system resources. This phenomenon is observed on Android 5.0 and 5.1. We speculate that Android 5.0 and 5.1 can detect the unusual system slowdown, thus the system ends Popper before Popper causes damage. We envelop Popper in a game application[1] and publish it to Google Play [8]. Google Bouncer does not block us from uploading.

In Popper, we use FF windows of `TYPE_TOAST` to avoid requesting for any permissions. Except for FF windows, we have verified that other windows added by `WindowManager`, including `Dialog` and `PopupWindow` can also cause the reboot of the Android system in the similar way.

The consequences caused by such DoS attack might be serious, since Android OS is widely used in various critical scenarios [15], e.g., Android-auto navigation[16], aircraft navigation [17] and patient monitoring [18]. The sudden failure of Android may result in serious damage. Moreover, the reboot can be launched at any time. A malicious application could frame an innocent application by launching DoS attack every time the innocent application starts. Or, a malicious application could listen to the `BOOT_COMPLETED` broadcast and keep launching the attacks once the victim Android finishes its reboot to immerse the victim Android system in the loop of reboot.

We have reported to Android the above vulnerability along with our analysis, which has been confirmed immediately[2]. This vulnerability is due to the chain reaction of the unavailability of file descriptors. During adding an FF window to current display, a `socketpair` is created to carry out the communication between `InputManager` and the calling application. Each socket needs a file descriptor. When Popper creates a large number of FF windows in such a short time, there is no more file descriptor to dispatch, which leads to the failure in creating `socketpair`. And thus, there is no connection establishing between `InputManager` and the calling application. The problem comes with that `WindowManager` fails in dealing with such exception. It results in the the crash of `ActivityManager`, which further kills the `system_server` process. The `system_server` process is such an important system process that if it is killed, `Zygote`, which is created at a very early stage during Android boot and can be seen as the parent process of all applications, is exited. In this case, Android has to reboot to recover all the damage.

3.2 GUI Hijacking by Targeting Overlap

The previous GUI confusion attacks mainly switch activities to take over the control flow [6] or create an inescapable fullscreen window that traps users within the attack applications [3]. Different from the previous focuses, we concentrate on 1) the capability of FF windows in display priority and event processing, and 2) the delicate design in combining of different FF windows to launch targeted attacks. We demonstrate that subtle GUI hijacking is very powerful in intercepting the communication between users and applications, e.g., retrieving the data passing from users to applications (Section 3.2.2), and tampering with the information from applications to users (Section 3.2.3). Compared to the previous confusion attacks, GUI hijacking using FF windows is stealthier since there is no animation of switching activities, and moreover the attack application does not appear in the recent task list.

3.2.1 General attack process

Normally, one subtle GUI hijacking attack is only customized for one target application due to the variety in functionality and design of the target application. Figure 2 illustrates the process of attacking one particular application. The main effort is during offline phase. Attackers need to design the attack work flow and attack FF windows according to the target application.

In order to avoid any suspect, the attack windows are supposed to be "user-transparent" so that a user can hardly tell the difference when the attack FF windows are popped up on top of the target. User-transparent experience should last during the whole process from the popping up of the attack FF windows to their exiting. Based on the user-transparent principle, there are several crucial problems need to be settled during offline design: 1) When is the right time to pop up the attack FF windows? 2) Which area of the target application will be covered? 3) How should the appearance of the attack FF windows look like? 4) Which type of FF windows will be used according to the

[1]In order to protect a normal user from being attacked, the DoS attack in published game is intentionally designed to be triggered only under certain operations. To trigger the DoS attack, one needs to sequentially and continuously input number 20, 4, and 7 to Level 5, 1, and 8 in the game, respectively. That is to open Level 5, input number 20, and go back to the main menu. Similarly for Level 1 and 8.
[2]CVE-2015-6648 is assigned to this vulnerability.

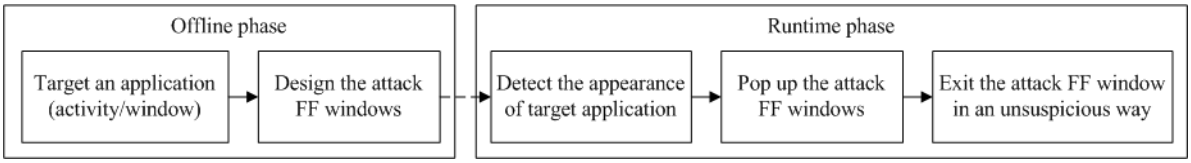


Figure 2: GUI hijacking process.

priority in display layer? 5) Whether the attack FF windows accept user-generated events? 6) How to exit the attack FF windows in a reasonable and unsuspicious way?

Later, when the attack application is distributed to devices, it can just act as designed, i.e., detecting the appearance of target application (e.g., one of the activities/windows in the target application), popping up the attack FF windows, processing the click events, and exiting after finishing tasks.

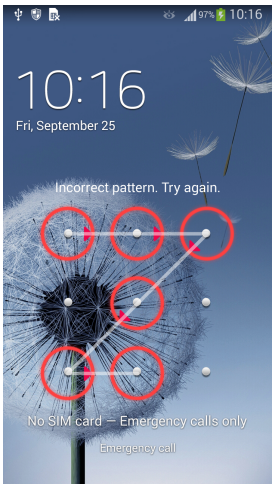
3.2.2 *Hijacking the system keyguard*

Keyguard is a protection mechanism provided by Android, which is invoked when a device is idle. It protects the touch screen from being accidentally touched, e.g., an application is accidentally launched or a phone call is unexpectedly made. When a keyguard is set with a secret unlock pattern which is usually a consecutive drawing among 9 dots, it can further protect the private data on the device from being accessed by people other than its owner. In this section, we show how to take advantage of FF windows to hijack the system keyguard with patterns, and intercept the secret pattern when it is input to the device by the device user[3].

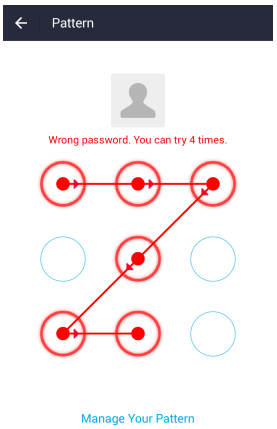
Choose the type of the attack FF windows. First of all is to determine the type of FF windows which are going to be used in keyguard hijacking. We need an FF window that can display itself above the system keyguard interface. According to Figure 1, there are two FF window types that can float above a keyguard, i.e., `TYPE_SYSTEM_OVERLAY` and `TYPE_SYSTEM_ERROR`. Another requirement for this FF window is that it should be touchable since it is going to process the secret pattern that a user inputs. An FF window of `TYPE_SYSTEM_OVERLAY` does not qualify in this scenario as it is not touchable. Therefore, we use an FF window of `TYPE_SYSTEM_ERROR` to hijack the system keyguard.

Determine the launch timing of the attack FF windows. The attack FF window needs a signal to trigger its launch. Normally, the keyguard should be the first interface after an Android device is waked by a user, which signals a system event named `ACTION_SCREEN_ON`. Hence, if the attack application listens to this event, it can determine the waking of the device. In case the keyguard is not active by the time the device is waked, the attack application needs to further check whether the current screen content is the keyguard by calling `KeyguardManager.inKeyguardRestrictedInputMode()`. If it is, the attack FF window should be launched.

Design the attack FF windows. This attack FF window is designed to be transparent. After the device user wakes his/her phone and the keyguard is shown, the transparent attack window is popped up appearing itself on top of the keyguard because of its high priority. When a user intends to draw his/her secret pattern to unlock the keyguard, the attack FF window intercepts all the touch events and obtains the pattern. The system keyguard knows nothing about what happens and of course the system is still locked.



(a) Hijack the system keyguard.



(b) Hijack Alipay.

Figure 3: Hijack the pattern unlock interface.

Exit the attack FF windows in an unsuspicious way. After obtaining the target pattern, it needs an unsuspicious way to exit and give the focus back to the keyguard. We design its exiting as follows. As shown in Figure 3a, after the user finishes his/her drawing (a pattern like "Z"), the attack FF window shows the trail that the user just drew but without the last dot. It looks exactly like the legitimate feedback when the last touch is failed. At the same time, an error is shown by another FF window, which says incorrect and recommends one more try. The FF window showing the error message is in the size just enough to show a line of error message. Sometimes this FF window may overlap with part of the original message from the keyguard, which may cause suspect. To avoid such suspect, the attack FF window showing error message dynamically gets the current wallpaper using `WallpaperManager` and fills up its background with the same part of the wallpaper by calculating according its display coordinates, which is like a patch showing an error message with the original wallpaper background. When the user is taking another try, all the attack FF windows exit and the keyguard gets the input focus back.

Apply the attack to other applications. This hijacking can also be applied to third-party applications. There are many applications using patterns to unlock

[3]The system keyguard using PIN can also be hijacked in a similar way. In this paper, we hijack the keyguard using pattern for demonstration.

themselves, e.g., Alipay[4] [20] uses unlock pattern in its Android application. We carry out experiments on Alipay as we do to the system keyguard, and manage to get its unlock pattern as shown in Figure 3b. In this case, the type of attack FF windows does not have to be `TYPE_SYSTEM_ERROR`. An FF window of `TYPE_TOAST` is enough for hijacking Alipay, since an FF window of `TYPE_TOAST` can overlap the target activity which is of `TYPE_APPLICATION`.

3.2.3 Hijacking the security alert

The security mechanism on Android is based on permissions. Android lists all the permission that an application requests during the application's installation. If a user clicks the install button, it means that all the required permissions would be granted. Otherwise, the installation fails. It works in an all-or-nothing mode[5], which perplexes the users who intend to use an application but do not willing to grant all the permissions it requests. To address this issue, mobile phone vendors customize their systems based on Android to support dynamic permission grant, such as Huawei's EMUI [21]. In fact, this customization is a compulsive requirement by China Communications Standards Association. It requires an explicit user confirmation at the first time of an application accessing a resource protected by a permission [22].

EMUI proposes a runtime permission control to solve the dilemma of all-or-nothing. On a Huawei device, an application is installed in the traditional all-or-noting mode. The difference happens when an application with granted permission accesses the protected resource on Huawei device, e.g., an application with `READ_CONTACTS` permission accesses the contacts. At the time of access, EMUI prompts a security alert dialog showing the information about the application and the sensitive resource it is trying to access as shown in Figure 4a. EMUI enables its users dynamically decide whether to allow the access or not. If a user allows the access, the application can truly get access to the protected resource. Otherwise, the access fails. For attackers, this is a barrier which prevents them from accessing desired resources. However, the security alert has no guarantee of its appearance in front of users. In the following, we demonstrate that a malicious application is able to obtain user consent with a higher possibility in EMUI when accessing the desired resources by hijacking the security alert using FF windows.

Choose the type of the attack FF windows. To hijack a security alert, the first step is to determine the type of the attack FF window, so that it can overlap the target security alert. Since the EMUI is a close-sourced product, we have to try the types in Table 1 to see which one can meet our requirement. We try `TYPE_TOAST` first to check out whether it can appear on top of the security alert, because `TYPE_TOAST` is the only one in Table 1 that does not require any permission. Unfortunately, `TYPE_TOAST` cannot cover the security alert. Hence, we use `TYPE_SYSTEM_ERROR` which enjoys the top priority among all the other types requiring `SYSTEM_ALERT_WINDOW` permission.

[4]Alipay, which is operated by Alibaba Group, is the largest financial application in China with 2.78 billion transactions in total of RMB 900 billion by early 2014[19].

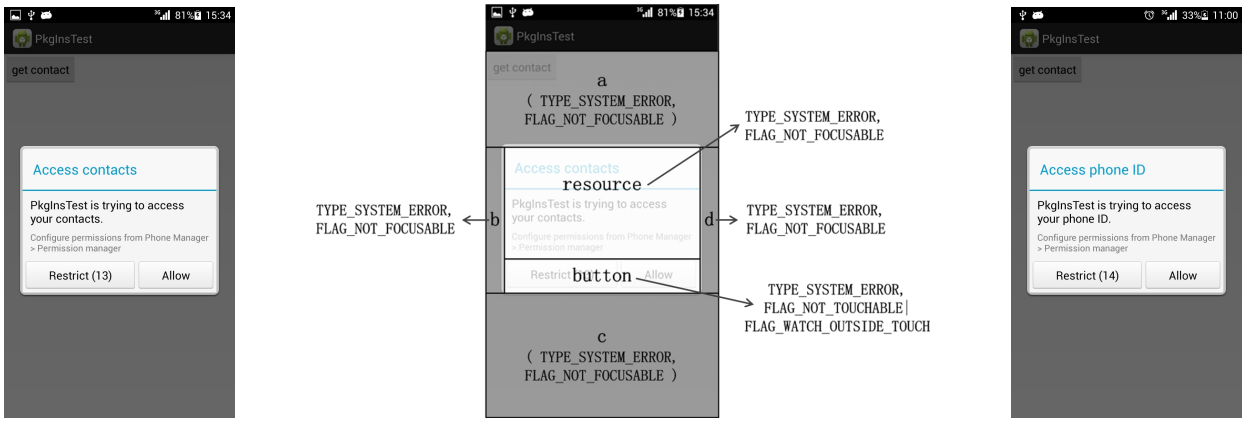
[5]Differently, the latest Android 6 allows a user to revoke the grant even after the installation.

Design the attack FF windows. Normally, a user prefers to allow an application to access a less sensitive resource, e.g., device ID, instead of a more sensitive resource, e.g., contacts. Therefore, we design such hijack which covers the true resource name with a less sensitive resource to trick users into allowing the access. At the same time, we let the fake alert work as the original security alert. It means that the dialog disappears if and only if a user clicks one of the allow/restrict buttons. Any touch out of this area does nothing to the display. Since different areas behave differently, it requires the combination of different FF windows in one screen. As illustrated in Figure 4b, the screen can be divided into 6 areas when the security alert shows. Area "resource" displays the resource that an application is trying to access. Area "button" is to respond the user click. And the other 4 areas "a", "b", "c", and "d" are the remaining areas which do not respond to any touch event. We adopt 6 FF windows with different behaviour features to cover those areas in Figure 4b. The FF window above "resource" area deceives the user by displaying a less sensitive resource name instead of the original one, e.g., informing the user that the device ID is being accessed instead of the contacts. This FF window is designed in the exact same style of the original "resource" area. We keep the other 5 FF windows transparent.

We also set the event processing capability for each FF window according to their desired behaviour features. The FF window above "button" area should pass down the user clicks inside its window range to the original buttons to make the user consent effective. According to Table 3, "button" FF window is set to `FLAG_NOT_TOUCHABLE` to pass down the click events. For the other 5 FF windows, since there is no need to pass down any user click to the covered areas, they are set to `FLAG_NOT_FOCUSABLE`. In this setting, only the covered window within "button" area can receive click events.

Determine the launch timing of the attack FF windows. These FF windows need to be launched at a proper time, that is when the security alert is shown on screen. To determine the proper timing, the malicious application, which is going to access a sensitive resource, pops up a invisible FF window with `FLAG_NOT_TOUCHABLE` before performing any access operation. This FF window can get the input focus when it is on the top layer of the screen. When the application is accessing the sensitive resource, the security alert would appear to warn the user of the access details. Once the security alert is shown, the invisible FF window popped up before would loss focus, which would trigger a callback of the FF window, i.e., `onWindowFocusChanged`. Once the `onWindowFocusChanged` is triggered, the attack FF windows can be launched.

Exit the attack FF windows in a unsuspicious way. The original security alert would disappear after the user clicks one of the decision button in the "button" area, because the FF window in this area passes down the user click to the original security alert. At the same time, all the 6 attack FF windows should disappear with the original alert to avoid any suspect. The challenge is that it requires all the 6 FF windows to be aware of the click event happened in the "button" area. To solve this problem, we introduce the assistant flag `FLAG_WATCH_OUTSIDE_TOUCH` which has been explained in Section 2. An FF window with `FLAG_WATCH_OUTSIDE_TOUCH` can listen to an `AC-`



(a) The security alert dialog in Huawei EMUI.

(b) The design of the attack FF windows.

(c) The effect of hijacking security alert.

Figure 4: Hijack the security alert dialog.

TION_OUTSIDE event which is created by Android system when an ACTION_DOWN event happens outside the window range. For an FF window with FLAG_NOT_TOUCHABLE (e.g., the "button" FF window), all the screen-wide area is considered as "outside". It means that if there is no touchable window above it along the Z-axis, any click event happens in any area of the screen can trigger an ACTION_OUTSIDE event to the "button" FF window. To guarantee the "button" FF window receives an ACTION_OUTSIDE event if and only if the click is within its window area, we pop up it first to place it below the other 5 attack FF windows along the Z-axis. The other 5 FF windows are popped up afterwards, whose order does not matter. In this situation, any click happens outside the "button" area would not introduce any ACTION_OUTSIDE event to the "button" FF window. At the time when the "button" FF window receives the ACTION_OUTSIDE event, it sends a message to the other 5 attack FF windows and destroys itself. Upon receiving the message, the other 5 FF windows destroy themselves. Till now, the exit is completed in a resonable and unsuspicious way.

The attack effect is shown in Figure 4c. After a user clicks one of the decision buttons, the attack FF windows would all disappear. A user would believe that (s)he allows an application to access the device ID, however, the fact is that the application acquires a different dynamic-granted permission and accesses more sensitive resources (e.g., contacts) without his/her knowing. The dynamic grant mechanism is bypassed. Similarly, this attack can be launched against other customized systems (e.g., MIUI [23] developed by Xiaomi Inc.) and third-party security applications (e.g., LBE [24]).

3.3 FF Windows as a Side Channel

Users usually has a regular input pattern when inputting on phone, e.g., the intervals between two characters during input [25, 26]. We demonstrate in this section that FF windows could serve as a side channel to get the intervals and further infer user input, which can increase the possibility of the password-guessing attack[6].

We target at the payment password used in financial applications. It is adopted by financial applications, e.g., Alipay and WeChat Payment[7], as an additional password when a user is confirming a payment or a transfer request. Different from the account password, a payment password usually consists of 6 digits. A typical keyboard from WeChat Payment layout is as shown in Figure 5a.

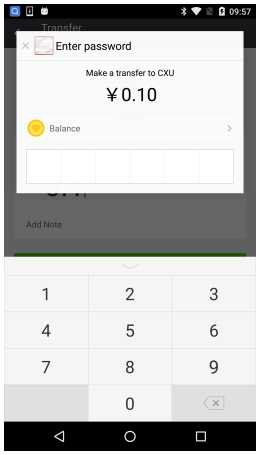
Design the attack FF window. We are going to use an FF window to obtain the intervals between two clicks, based on which the input digits can be inferred with certain possibility. Normally, when a click event is generated by a user, it is passed to the touched window, i.e., keyboard in this case, which is going to process the event according to its click position. Simultaneously, this event is passed to all FF windows with FLAG_WATCH_OUTSIDE_TOUCH on top of the touched window via ACTION_OUTSIDE which excludes the information about click position. Therefore, an FF window with FLAG_WATCH_OUTSIDE_TOUCH on top of the touched window is able to access the precise time when a user click happens, and hence the interval between two user clicks. Besides obtaining the input intervals, this FF window should not interfere any touch event. According to Table 3, we set it to FLAG_NOT_TOUCHABLE which does not respond to any click event no matter inside or outside the FF window so that it would not affect the input to the keyboard. In order to be stealthy as much as possible, the FF window is transparent and in pixel size.

The attack procedure. The procedure contains two phases, i.e., model training and model applying.

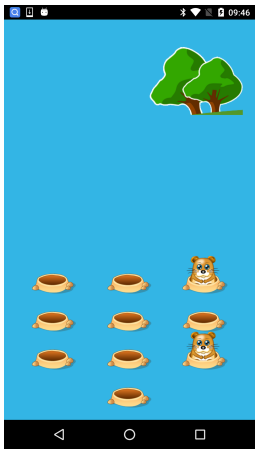
In the model training phase, we build the user input pattern model. In this phase, it shows a 2-digit number each time and requires a user to input the correct number. The input activity is similar to the activity requiring payment password in WeChat Payment so that the user input pattern we build could apply to the scenario of payment password input. The user click positions and the intervals between

[6]The regular input pattern can be reflect only when the keyboard layout is fixed, a self-defined keyboard which randomly shifts the position of each key every time does not affect by this attack.

[7]WeChat Payment is one of the most popular payment option in China. It is a payment feature integrated in WeChat [27] with around 400 million users by middle 2015 [28].



(a) The interface requiring payment password in WeChat Payment.



(b) The interface of the model training game.

Figure 5: Inferring payment password using FF windows as a side channel.

N \ Edit distance	0	1	2	3	4
10	0	1	14	31	48
50	0	8	24	46	84
100	2	12	31	54	77
200	6	19	39	63	62

Table 4: The distribution of N most-likely sequences in terms of edit distance between inferred digit sequences and the original ones.

N \ Edit distance	0	1	2	3	4
10	2	27	36	48	70
50	26	32	45	55	31
100	34	43	39	63	10
200	51	40	37	60	1

Table 5: The distribution of N most-likely sequences in terms of edit distance between inferred digit sequences and the original ones under the condition that the first digit is obtained accurately.

every two digits are recorded, which are used to construct the Hidden Markov Model (HMM) model. HMM model is suitable for our input inference. The HMM is different from a Markov Model (MM) which is a way of describing a finite-state stochastic process with the property that the probability of transitioning from the current state to another state depends only on the current state [29]. In HMM, the current state of a finite-state process cannot be directly observed. Only some outputs from the state are observed, and the probability distribution of possible outputs given the state is dependent only on the state [30]. In our scenario, every two consecutive digits are considered as a non-observable state and the intervals as the outputs. The possibility of next digit pair in the user input depends only on the current digit pair, which is suitable to use HMM.

Next in the model applying phase, we simulate the scenario of inputting payment password. A 6-digit number is shown on screen. A user is asked to remember and input this number correctly. At the same time, there is a transparent FF window popped up to obtain the intervals between two clicks. We adopt Top-N HMM [30], which outputs N most-likely 6-digit sequences.

For demonstration purpose, we describe the procedure in a technical manner. In real-world scenario, the process should be presented in an interesting way, e.g., a game. We have developed an application "whack a mole" with two modes corresponding to the above two phases in attack procedure. In the model training phase, there would be no specific 2-digit number leading a user to input. The intervals are collected during whacking the moles. For example in Figure 5b, we can get the intervals between 3 and 9 by showing two moles at the same positions where 3 and 9 are located in the keyboard. A simplified version of this application is published to Google Play [9].

Experiment. We conduct experiment to evaluate this attack. In the model training phase, we prepare 100 2-digit numbers and each number repeats 50 times. Volunteers are recruited to go through the designed experiment. A volunteer is asked to go through all the numbers and complete the input. The input numbers and the intervals between two digits are recorded.

In the model applying phase which simulates the attack scenario of inferring the payment password, the volunteer is asked to input a set of 6-digit numbers (which consists of 189 numbers in this experiment). In this phase, the invisible FF window which can obtain the click intervals is popped up. The intervals are used as input to the model we trained in previous training phase. The edit distance is used to measure the distance between the inferred digit sequences and the original ones. The results are shown in Table 4. It is shown that using the 100 most-likely sequences, there are 12 numbers among the 189 numbers with 1 in edit distance which means the inferring sequences miss 1 digit compared to the original sequence.

Since the probability distribution of the next input digit depends only on current state, we further investigate the results when the first input digit is accurately known. To obtain the first digit a user inputs, we use an FF window with `FLAG_NOT_FOCUSABLE` to cover the whole keyboard at the beginning of user input. This FF window can get the click event including the click position, thus we can know the exact digit the user inputs. After the first digit is obtained, this FF window is replaced with the original pixel-sized FF window with `FLAG_NOT_TOUCHABLE` and `FLAG_WATCH_OUTSIDE_TOUCH`. The following process is the same as the previous. From the perspective of a user, it seems like that the first click fails. It can be seen in Table 5 that the accuracy enhances a lot when the first digit is known in advance. Compared to the normal attack mode, in the 100 most-likely sequences, there are 43 sequences among the 189 numbers missing just 1 digit. We even get 2 of the 189 numbers with 0 edit distance in 10 most-likely sequences.

As shown in experiment results, it is effective to increase the possibility of the password-guessing attack by obtaining the intervals using FF windows. The results are even better when the first digit is obtained using an FF window in advance.

4. DEFENCE

According to the attack analysis in the previous section, FF windows can be abused to launch attacks such as DoS attack against Android system, GUI hijacking, and input inference attack. In order to protect Android system and its users from such attacks, we discuss the cause of the attacks and propose defence mechanism in this section. Specifically, we propose and implement a priority framework for FF windows and a solution to GUI attack via activities. Finally, we provide suggestions on how to enhance the management for FF windows.

4.1 The Priority Framework for FF Windows

The GUI hijacking attack and the input inference attack via FF windows are both due to the fact that a user is unaware of the untrusted FF windows on top of the user interface which (s)he intends to interact with.

There are several challenges to prevent such attacks. 1) Under the current management for FF windows, it is useless to set the target FF window to type with higher priority. On Android, the priority of an FF window in layer order is strictly according to its type as shown in Figure 1. For FF windows with the same type, the FF window popped up later is on top of the former one. Normally, the attack FF windows are launched after the target FF window appears. Therefore, even if the target FF window is set to `TYPE_SYSTEM_ERROR`, which is the most prior type a third-party application can use, the attack FF window can still appear on top of it as long as the attack FF window is also set to `TYPE_SYSTEM_ERROR`. 2) Another challenge is that FF windows do not respond to the Back button or Home button, it is difficult for a user to exit a malicious FF window even when (s)he discovers it. 3) In addition, it is not ideal to prohibit third-party applications from using FF window types with high priority. High priority types, such as `TYPE_SYSTEM_ERROR`, have already been widely used in the third-party applications, e.g., applications with customized user interactions when screen is locked. It would cause incompatibility between Android system and a lot of such applications to prohibit the use of high priority types.

To solve the challenges, we present a priority framework which requires no change in the current type setting and involves no operation from users. This priority framework is able to protect a sensitive activity/FF window, e.g., an activity accepting passwords or an FF window showing security alerts, from being overlapped by any FF window, if it is designed not to. If a developer designs an activity/FF window to be sensitive and be free from being overlapped, we call this activity/FF window a *non-overlappable* activity/FF window. To mitigate the attacks via FF windows, a non-overlappable activity should not be overlapped by any other FF windows, no matter the FF window is from the same or different application, and no matter the FF window is non-overlappable or not. Similarly, a non-overlappable FF window should not be overlapped by any FF window, if there is no non-overlappable FF window occupying the screen at the moment. The details of our solution are explained below.

Technically, we achieve the above goals by introducing an attribute `overlappable` to `WindowManager.LayoutParams`. It is used to specify whether the activity/FF window can be overlapped or not, which by default is true, i.e., behaving like a normal activity/FF window. A developer can set this attribute in his/her activity/FF window to explicitly declare this activity/FF window should not be overlapped by any FF window from any other applications. This attribute regulates a window[8] in the following way:

1. For overlappable windows, they behaves according to the original rules from Android, i.e., following the priority order based on the types and showing the window popped up later on top of the former one with the same type.

2. A non-overlappable third-party window is always on top of an overlappable third-party window.

3. For non-overlappable third-party windows, an activity is on top of an FF window to protect an activity from being overlapped by a malicious FF window which is also set to non-overlappable, i.e., false value in the attribute `overlappable`.

4. For non-overlappable third-party FF windows, the one popped up later is below the former one.

5. For the sub-windows (e.g., `Dialogs` and `PopupWindows`) of a non-overlappable third-party window, their relative order along the Z-axis follows the original rules in Android.

6. A non-overlappable system window is always on top of a third-party window.

7. For non-overlappable system windows, they behaves according to the original rules from Android, i.e., following the priority order based on the types and showing the window popped up later on top of the former one with the same type.

We implement the prototype on Android 5.0 by adjusting the layer of a window in `WindowManager` according to its `overlappable` value. `WindowManager` is a system service responsible for managing windows appearance and position including the Z-ordered list of windows. If a third-party window is set to "false" in `overlappable`, i.e., should not be overlapped, we place it on top of the window of `TYPE_SYSTEM_ERROR` along the Z-axis. Because `TYPE_SYSTEM_ERROR` is the most prior type that a third-party window can use. If a system window is set to non-overlappable, we adjust it according to its type. If it is more prior than `TYPE_SYSTEM_ERROR`, we left it to original rules. Otherwise, we adjust its position along Z-axis to make sure it can appear on top of all third-party windows including the non-overlappable ones. After that, Android can protect a sensitive window explicitly specified as non-overlappable by a developer from being overlapped by any FF windows. This solution introduces insignificant changes (105 lines of core code) to Android system.

4.2 Solution to GUI Attack via Activities

In the priority framework, we mitigate the attacks from FF windows by setting the sensitive activities or FF windows to a non-overlappable status. However, it is mentioned in previous work [3] that GUI attacks via activities, which we do not focus on in this paper, can also cause serious

[8]Since an activity is equivalent to the FF window with `TYPE_APPLICATION` in layer priority, we use "window" to refer to an FF window or an activity.

security issues. GUI attack via activities is to deceive user by covering the current activity with a malicious activity from another application. When Android switches the activities between applications, it is different from the way to start an FF window. Normally, Android switches off the current activity before starting the next activity. Since the current activity is no longer in front of the screen by the time of switching, it cannot be protected by attribute `overlappable` when replaced by another activity.

We propose a complementary solution to our priority framework to solve the attacks from activities. We adopt the way of showing information on the navigation bar [3]. Because the navigation bar is designed to be easily accessed during the use of Android. Even in the fullscreen mode, it can be accessed by either clicking or swiping on screen. However, there is a challenge to guarantee the appearance of the navigation bar. An FF window of type `TYPE_SYSTEM_ERROR` can create an "inescapable" fullscreen user interface [3], which may result in the inaccessibility of the navigation bar. To solve the challenge, we adjust the display priority order of the navigation bar. Different from previous solution [3] which prohibit third-party applications from using FF windows of `TYPE_SYSTEM_ERROR`, we adjust the window of `NAVIGATION_BAR` or `NAVIGATION_BAR_PANEL` on top of the window of `TYPE_SYSTEM_ERROR` along the Z-axis. This adjustment impacts little on the normal use of FF windows. Because a navigation-bar-related window is only for the system use showing at the bottom of the screen, and a benign FF windows with `TYPE_SYSTEM_ERROR` should not try to block the user from accessing the navigation bar. In such case, `TYPE_SYSTEM_ERROR` cannot be used to cover navigation bar any more, so that we can guarantee the appearance of the assistant information.

We show the icon and the package name the application that a user is interacting with on the navigation bar. The priority framework in Section 4.1 can prevent the sensitive activity/FF window declared by developers from being overlapped by FF windows. So if the sensitive activity/FF window is not overlapped by another activity, it must be on the top layer. Hence, the information about the top application, which the top interactive activity belongs to, is sufficient to give users hint about whether the activity/FF window is covered by any malicious activity. The icon in navigation bar is designed to be clicked to show information about the top application as shown in Figure 6. The window showing the application information is set to the `TYPE_HIDDEN_NAV_CONSUMER` which is only for the use of system application and enjoys the top priority in display order to guarantee its appearance to users. The user can check whether the activity currently on the top layer is the one that (s)he is intended to interact with by the icon, package name, or detail information after clicking the icon. Different from an FF window which does not respond to Back button or Home button, an activity is easy to be ended by clicking the Back button or Home button once the user notices anything unusual, e.g., the top application is not the one (s)he intends to interact with.

4.3 Discussion

Different from activities, it is difficult for a user to notice the existence of malicious FF windows since the FF windows can be highly customized with various attributes. Even if the user has noticed the existence of an FF window, (s)he cannot distinguish which application the FF window is from. Moreover, unlike the activities which can be closed by click Back button or Home button on the navigation bar, FF windows respond to neither of them, which may stuck the user in an interface responding to nothing.

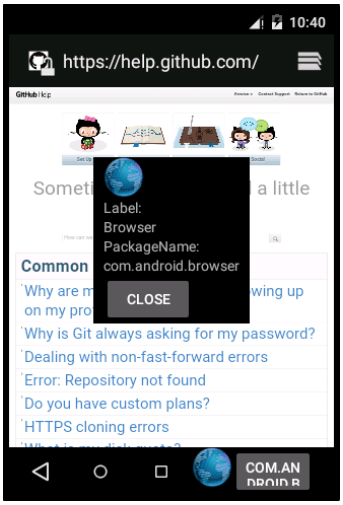


Figure 6: The demo of solution to GUI attack via activities.

To eliminate such difficulty, we suggest Android should restrict the power of FF windows and provide specific management of FF windows. As analysed, the only way to exit an FF window is to close the application it belongs to. Android could be enhanced to add management for FF windows, e.g., users can long press the Home button to check the active FF windows and the applications they belong to and close certain FF window by closing its corresponding application. To protect the usability of the Home button, it is suggested that Android should prevent FF windows from disabling the Home button, as it does to activities since version 4.0.

In addition, in the DoS attack in Section 3.1, the Android system dies when a large number of FF windows, `Dialogs`, or `PopupWindows` exhaust the system resources. Like the prevention of other DoS attacks, it is an effective way to enforce restriction on the use of system resources. Android has limited the number of Toasts since Android 4.0, that any request to pop up more than 50 Toasts will be ignored. Similarly, to defend against such DoS attack, we suggest that Android restricts the number of windows, including FF windows, `Dialogs`, and `PopupWindows` that an application can invoke.

5. RELATED WORK

The DoS attack on Android is to occupy or flood the resources in Android system, so that Android can no longer provide normal services. Huang et al. [15] launch the DoS attack by occupying the critical lock in Android. They observe that Android system services often use the lock mechanism to protect critical sections or synchronized methods, many of which share the same lock. If an application takes a lock for a long time, other services share the same lock would freeze. Further, the watchdog thread would detect the unavailability of the lock it monitors and force Android to reboot. There are also related works

showing that DoS on Android can be caused by Toast [31] and Flash SMS [32]. Lineberry et al. [31] reveal that an application can carry out DoS attack by popping up a large number of Toasts which exhaust the limited indices that the system can create. It leads to the restart of Android system. DoS attack based on Toasts can only work on Android with versions lower than 4.0, since this problem has been fixed by limiting the number of Toasts an application can pop up. DoS based on Flash SMS [32] relies on the limited number of Flash SMS notifications that Android can cope with. A number of Flash SMS notifications result in the restart of the SMS application, and even the restart of Android system depending on the attack situations. This vulnerability is fixed since Android 4.4.2. Different from the above three DoS attacks, DoS based on FF windows is caused by a different resource exhaustion. DoS attack based on FF windows works on nearly all Android versions from 2.3 to latest 6.0.

In the area of GUI attacks, Bianchi et al. [3] classify the GUI confusion attacks to three classes according to the Android UI objects the attacks utilize. They focus on the attacks based on Toasts, Activities, and fullscreen Windows. The most harmful inescapable fullscreen window is a type of FF window with specific attributes. Different from their work, our paper explores the attributes of FF windows, according to which the behaviour feature of an FF window varies. We further identify the attacks that based on the FF windows with different behaviour features, including DoS attacks, GUI hijacking, and user input inference.

Input inference based on information from side channels on mobile devices is widely studied. Previous works utilize the sensors on mobile devices, e.g., accelerometer, gyroscope, camera, and microphone, to get side-channel information about the input content. TouchLogger [33] demonstrates the possibility of inferring user input via device orientation data. Aviv et al. [34] use accelerometer data to infer the PIN and pattern of system keyguard with an accuracy of 43% and 73% within 5 attempts, respectively. Taplogger [35] stealthily logs the password of screen lock and the numbers entered during a phone call using accelerometer and gyroscope data generated by user input. PIN Skimmer [36] correlates sensor data to the position of the digit tapped by the user. It uses microphone to detect touch events, and the camera to estimate the smartphone's orientation. Except for accessing sensor data, e.g., accelerometer and gyroscope, it requires the calling application to have certain permission when accessing other resources, e.g., camera and microphones. In this paper, we target at the payment password in financial applications and use an invisible FF window to get the pattern when a user inputs, which requires no permission. We use the FF window to get the interval between two taps, based on which we build an HMM model, and further infer the input digits using the model we build. The experiment results show that our inference can be used to stealthily increase the possibility of the password-guessing.

The defence against GUI confusion is studied in previous work [3], which proposes an on-device defence by showing the authenticity about the application taking the top layer of screen on navigation bar. It uses the Extended-Validation certificate in HTTPS to identify the authenticity of an application. Like a browser shows a green lock image on URL bar to authentic HTTPS URLs, it shows green lock on the navigation bar to the authentic applications when there is no visible UI element from other applications. It requires developers to apply the certificate from a certificate authority for a fee, which may not applicable for small enterprises or individual developers. Moreover, as a defence mechanism, it fails to defeat the possible disturbance from malicious FF windows, e.g., a one-pixel-sized FF windows with no event processing attribute could disable the notification of authenticity by keeping the lock yellow while keeping the top application usable. Our solution propose a priority framework for FF windows, which can guarantee the sensitive activity/FF window would not be overlapped by any FF window. This is done by adding an extra attribute to `WindowManager.LayoutParams`, which can be used by developers to explicitly declare the sensitive activity/FF window. Except for the mitigation to attacks via FF windows, we also provide a complementary solution to solve the attacks based on activities. We emphasize that it is important for users to know the information about the application on the top layer of screen, instead of just notifying user the results. By adjusting the window of `NAVIGATION_BAR` on top of the window of `TYPE_SYSTEM_ERROR` along the Z-axis, we can securely show information about the current top application to users. Unlike previous solution [3], our solution requires no involvement of third-party authorities. The priority framework combined with the solution for attacks based on activities can prevent a sensitive activity/FF window from being hijacked by FF windows and activities.

6. CONCLUSION

In this paper, we study the important behaviour features of FF windows, based on which three types of attacks can be performed by FF windows with delicate attributes and subtle design, i.e., DoS attack against Android systems, GUI hijacking by targeting overlap, and input inference using FF windows as a side channel. We analyse these attacks and conduct experiments to show the threat from FF windows. To address the threat from FF windows, we propose a priority framework for FF windows to prevent any sensitive activity/FF window from being overlapped by malicious FF windows. In addition, we propose a complementary solution for GUI hijacking attacks based on activities. Finally, some suggestions are provided to Android to solve the threat caused by FF windows.

7. ACKNOWLEDGEMENT

This research is supported by the National Natural Science Foundation of China (Grant No. 61502468), the Beijing Municipal Natural Science Foundation (Grant No. 4154089), and the National Basic Research Program (Grant No. 2012CB315804).

8. REFERENCES

[1] Rene Millman. 97% of malicious mobile malware targets android. http://www.scmagazineuk.com/updated-97-of-malicious-mobile-malware-targets-android/article/422783/.

[2] Smartphone os market share, 2015 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[3] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *36th IEEE Symposium on Security and Privacy*, 2015.

[4] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[5] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security*, pages 227–243. Springer, 2013.

[6] Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[7] Android and security. http://googlemobile.blogspot.sg/2012/02/android-and-security.html.

[8] Demo - dos attack via ff windows. observe and memory. https://play.google.com/store/apps/details?id=com.lulu.kaoyanli.

[9] Demo - input inference. whack-a-mole. https://play.google.com/store/apps/details?id=com.paperwork.expfloatwindowinput.

[10] Wandoujia. https://www.wandoujia.com/.

[11] Anzhi market. http://www.anzhi.com/.

[12] View. http://developer.android.com/reference/android/view/View.html.

[13] Toast. http://developer.android.com/reference/android/widget/Toast.html.

[14] The distribution of android versions. https://developer.android.com/about/dashboards/index.html.

[15] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1236–1247. ACM, 2015.

[16] Android auto. https://www.android.com/intl/en_us/auto/.

[17] Northrop to demo darpa navigation system on android. http://www.govconwire.com/2013/04/northrop-to-demo-darpa-navigation-system-on-android-charles-volk-comments/.

[18] Android and rtos together: The dynamic duo for today's medical devices. http://embedded-computing.com/articles/android-rtos-duo-todays-medical-devices/.

[19] Milestones of alipay. http://ab.alipay.com/i/dashiji.htm.

[20] Alipay. https://play.google.com/store/apps/details?id=com.eg.android.AlipayGphone.

[21] Huawei emui. http://emui.huawei.com/en/portal.php.

[22] Technical requirements for security capability of smart mobile terminal. http://www.tenaa.com.cn/html/2407-2013%20%E7%A7%BB%E5%8A%A8%E6%99%BA%E8%83%BD%E7%BB%88%E7%AB%AF%E5%AE%89%E5%85%A8%E8%83%BD%E5%8A%9B%E6%8A%80%E6%9C%AF%E8%A6%81%E6%B1%82.pdf.

[23] Miui. http://en.miui.com/.

[24] Lbe. http://www.lbesec.com/.

[25] Alexander De Luca, Alina Hang, Frederik Brudy, Christian Lindner, and Heinrich Hussmann. Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 987–996. ACM, 2012.

[26] Nathan L Clarke and SM Furnell. Authenticating mobile phone users using keystroke analysis. *International Journal of Information Security*, 6(1):1–14, 2007.

[27] Wechat. http://www.wechat.com/en/.

[28] A change in pay. http://www.bjreview.com.cn/quotes/txt/2015-06/10/content_691513.htm.

[29] Artificial Intelligence. A modern approach. *Russell and Norvig*, 2003.

[30] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[31] Tim Wyatt Anthony Lineberry, David Luke Richardson. These aren't the permissions you're looking for. https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf, 2010.

[32] Bogdan Alecu. 0class2dos. http://www.slideshare.net/DefCamp/0class2-dos, 2013.

[33] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *HotSec*, 2011.

[34] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2012.

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

[36] Laurent Simon and Ross Anderson. Pin skimmer: Inferring pins through the camera and microphone. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones and mobile devices*, pages 67–78. ACM, 2013.