Analysis of Clickjacking Attacks and An Effective Defense Scheme for Android Devices

Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji
Department of Computer and Information Sciences
Temple University
Philadelphia, Pennsylvania 19122
{longfei.wu, benjamin.brandt, dux, boji}@temple.edu

Abstract—Smartphones bring users lots of convenience by integrating all useful functions people may need. While users are spending more time on their phones, have they ever questioned of being spoofed by the phone they are interacting with? This paper conducts a thorough study of the mobile clickjacking attacks. We first present how the clickjacking attack works and the key points to remain undiscovered. Then, we evaluate its potential threats by exploring the feasibility of launching clickjacking attacks on various UIs, including system app windows, 3rd-party app windows, and other system UIs. Finally, we propose a system-level defense scheme against clickjacking attacks on Android platform, which requires no user or developer effort and is compatible with existing apps. The performance of the countermeasure is evaluated with extensive experiments. The results show that our scheme can effectively prevent clickjacking attacks with only a minor impact to the system.

Keywords—*Android; security; clickjacking*

I. INTRODUCTION

Smartphone continues its popularization worldwide and has become an important part of people's daily lives. Android is the most popular and the best-selling smartphone operating system (OS), holding over 80% of global smartphone market share [1]. However, security and privacy issues are a widely recognized problem of Android, mainly because it is open-source and attackers can find security vulnerabilities from the source code.

The security of user interface (UI) is particularly important, since mobile users interact directly with the UIs of the system as well as 3rd-party apps. Specifically, users receive most information visually from the UI, and give their inputs in terms of touch, click, and key entry to the UI as well. The manipulation of UIs can pose huge threats to the interaction between user and the phone. In general, the identities of two UIs should be treated carefully regarding the UI security: the UI that the user thinks of interacting with and the UI that is actually taking the user inputs. It is very important to guarantee that these two identities are consistent, otherwise it indicates that the user is spoofed and an app is receiving the user inputs while it is not supposed to. When talking about illegally gaining user input, most people immediately blame phishing attacks. However, the app to which user inputs are sent to is not necessarily the malicious one. Instead, it can be the victim in a clickjacking attack. For example, it is covered intentionally by a malicious UI that is "transparent" to user inputs (does not accept user inputs). As the result, the victim app is "forced" to take the user inputs while the user is expecting to interact with the app on top. Hence, we look into the mobile UI spoofing attacks in two categories based on whether the malicious UI intends to obtain the user input, namely the phishing attack (steal user input) and clickjacking attack (redirect user input to the victim UI).

In this paper, we focus on mobile clickjacking attacks. We start with the general pattern and key steps to implement a clickjacking attack. Then, we give a detailed analysis of the potential risks posed by clickjacking. Finally, we propose an automatic, lightweight and effective defense scheme to defeat clickjacking attempts, which is able to overcome the limitations of all existing solutions. All different types of clickjacking attacks and the defense mechanism are implemented on a Nexus 4 smartphone running Android 5.0 system. The effectiveness and overheads of the proposed scheme are evaluated with extensive experiments.

Our main contributions are listed as follows:

- We systematically study mobile clickjacking attacks from the perspective of floating window and target window, separately. We discover and analyze more unique features of clickjacking including the window flags, transparency, etc., which make our detection scheme more accurate than existing solutions which mistakenly accuse some benign apps due to their coarse policies.
- We investigate the "after-attack" disguises to keep the clickjacking undiscovered after one successful attack, which has not been considered in previous works. Specifically, we present three types of side-channels that allow the malicious app to listen to the user input events (different to [2], not necessarily lead to UI state change).
- We explore a variety of clickjacking attacks, targeted on system apps, 3rd-party apps, and other particular system UI. The threat of clickjacking is better evaluated than previous works which only have a couple of examples for illustration.
- Our detection scheme outperforms previous methods as it requires no user/developer involvement and is compatible with Android system design as well as existing apps. We implement the proposed scheme on real smartphone. The experimental results show that it is effective and efficient.

II. BACKGROUND

A. Android UI Basics

There are two confusing concepts that we want to explain in the first place: view and window. In Android, a *View* object
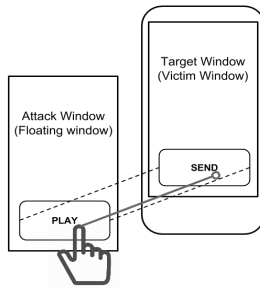
Fig. 1: Illustration of Clickjacking Attacks

is the basic building block for UI components (e.g., buttons, images, text fields, etc.). *View*s can be used to draw graphical contents. A *Window* object is an abstract base class for a top-level window look and behavior policy. By comparison, *View* objects are closer to the conventional windows we considered to display content as graphical user interfaces (GUIs), while *Window* objects are more likely a framework to hold the *View*s. In this paper, both terms "view" and "window" refer to the UI implemented using *View* object.

B. Overview

Clickjacking attack is also known as "UI redress attack". It happens when a malicious app inserts an opaque layer (or in very low transparency) on top of the screen, to trick a user to click on a specific position. The click event seemingly going to the top front window actually goes to the target window underneath. If carefully designed, the user may trigger a concealed button or link in the underlaying window. Note that the conventional webpage clickjacking attacks have been well-studied in previous works [3]–[8], most of them can be migrated to mobile platform to protect mobile browser and webpages. In this paper, we focus on clickjacking attacks on mobile application UIs and system UIs. An illustration is presented in Figure 1, in which the attack window appears as a game menu and covers on top of the target window. When the user clicks to start the game, the click is instead received by the "SEND" button in the target window (e.g. a premium SMS). Obviously, the target window is launched by the malicious app after the attack window has blocked user's sight. This is because if instead the target window is launched first (by the user), then the only way that the clickjacking can succeed is to have the user herself open the malicious app (attack window) while the target window is on front. This condition (predicting user behavior) is too strong for any real-world app.

Therefore, the attack window needs to be added before the target window, and remains on top when the target window is launched; otherwise, the user will see the target window launched without her command. Intuitively, the attack window cannot be in the same type as normal apps, which will be covered by the target window launched later. In fact, the attack window must be a floating window which has higher priority to stay on top of normal app windows (detailed in Section IV-A). Besides, the attack window has to be untouchable (does not take user inputs), so that user inputs can penetrate this camouflage layer and take effect on the target window below.

III. RELATED WORKS

A recent work conducted by Bianchi et al. [9] comprehensively study the Android UI attacks. In their work, GUI confusion attacks, including both the phishing attack and clickjacking attack are handled together. They first develop a static analysis approach to detect apps that try to interfere with the UI in response to some action taken by the user (or another app). However, unlike the phishing attacks that are performed right after the user launches the target app login UI, a clickjacking attack does not have a specific timing (the launch of the attack window and the target window are both controlled by the malicious app). Hence, clickjacking attacks can bypass this method as they are independent of user's or other app's action. Meanwhile, some benign apps are captured instead (e.g., locker apps will always stay on top of the app that is being protected until the user authenticates herself). They also devise an on-device defense by modifying the Android system. A trusted indicator is added to show the identity of the app that the user is interacting with, and its developer. Extended-Validation (EV) HTTPS infrastructure is used to verify if an app is indeed associated with a specific domain name. Warnings will be given if the user is interacting with an unverified app or an external window is present over the top activity. The limitations of this solution are three-fold: (1) Users have to make the final decision, which requires mobile users to understand the potential attacks they are facing and evaluate the risk in each specific situation. (2) Yellow alert will be issued to a verified app covered with an external window. However, some apps create an "always visible" window (e.g., Facebook Messenger provides the ability to chat while using other apps), and false alarms are generated when dealing with such benign apps. (3) The HTTPS EV certificates adopted will force the apps to be associated with domain names, and the certificate itself costs more than $100 each year. The verification of apps and developers is not a straightforward approach to excluding benign apps. The reason why they are using this approach as a complement is that the common features of different mobile UI attacks alone are not sufficient to detect any of the UI attacks. Instead, we split the general UI attacks to phishing and clickjacking, and develop separate detection schemes based on each's unique features (i.e. [10] and this work). Next, we discuss the existing solutions that are specific to the clickjacking attacks on mobile platform.

Niemietz et. al [11] study the mobile app clickjacking attacks through case studies. They propose to add a security layer between all neighboring apps (without implementation), so that no user input can pass across apps. However, some benign app windows are designed to be untouchable (transparent to user input) and floating on top of the screen (illustrated in Section IV-A). Blindly rejecting pass-through user inputs could cause serious incompatibilities with such apps. Besides, Android system creates flags like *FLAG_NOT_TOUCHABLE* and *FLAG_NOT_TOUCH_MODAL* for UIs, to offer the option to let the input events above/outside of a window to be delivered to the window below. The proposed solid security layer will spoil the intrinsic design of Android, and cause many existing apps to malfunction.

Android framework offers a touch-filtering mechanism as an option for app developers to protect the windows of their apps from receiving any user input event when obscured by another visible window. Specifically, the touch filtering can be activated by calling *setFilterTouchesWhenObscured* function, or by setting the *android:filterTouchesWhenObscured* attribute of a layout to true (for all views contained or only selected

TABLE I: Window Types for Floating Window

Window Type	Layer Value	Permission Required	Description
TYPE_PHONE	3	SYSTEM_ALERT_WINDOW	user interaction with the phone (in particular incoming calls)
TYPE_TOAST	7	None	transient notifications
TYPE_PRIORITY_PHONE	8	SYSTEM_ALERT_WINDOW	priority phone UI, needs to be displayed even if the keyguard is active
TYPE_SYSTEM_ALERT	10	SYSTEM_ALERT_WINDOW	system alert window, such as low power alert
TYPE_SYSTEM_OVERLAY	19	SYSTEM_ALERT_WINDOW	system overlay windows
TYPE_SYSTEM_ERROR	22	SYSTEM_ALERT_WINDOW	internal system error windows

views). This method is also suggested by Niemietz's follow-up work [12]. When enabled, the protected views will simply discard inputs whenever a toast, dialog or other window (may belong to a benign app) appears above. However, this solution is incompatible with benign apps that contain a floating window. This explains why Android phone manufacturers won't set up this flag by default for their system apps. What we need is a smart defense scheme that can make the filtering decision in real-time and only filter out misled input.

Fernandes et al. [13] find that the security indicator in [9] checks the identity of the foreground app periodically, so that the malicious app can launch clickjacking attacks right between two consecutive checking points. The timing of the binder IPC calls can be leveraged as the side-channel to predict the next check. Instead, they propose Overlay Mutex which guarantees that a background non-system app cannot overlay a window on top of another app's window while it is using soft keyboard. However, it will block the pop-up notification window of benign 3rd-party apps, and is conflicting with Android flag FLAG_NOT_FOCUSABLE which does allow an overlaid window to interact with the soft keyboard. We need to design the clickjacking detection scheme to be compatible with Android system and all existing apps.

IV. MOBILE CLICKJACKING ATTACKS

In this section, we describe the mobile clickjacking attacks in detail. Specifically, we first introduce the floating window and the target window, which are the basic components in a clickjacking attack. Next, we present a step-by-step analysis of the general attack model. Then, we discuss the potential threats posed by clickjacking attacks.

A. Floating Window

Floating window is a special type of window that can remain on top of normal app windows. By using a floating window, UIs of multiple apps can be displayed together, which facilitates the cooperation between apps and simplifies user operations. For instance, some apps keep a small and semi-transparent floating icon on screen, to provide quick access to useful functions (e.g., switching network connections, GPS, taking photos, etc.). These apps are welcomed by mobile users because of their convenience. These functions are usually accessible within 2 clicks, which otherwise would require tedious app switching operations including pausing the current activity, going to home menu or recent app list, activating the target app, and getting back to the original app after completion. Some other apps create an untouchable semitransparent floating window. For example, in screen camera apps, a full-screen translucent camera preview is projected into a floating window so that pedestrian users can see the traffics/obstacles

ahead while texting; system performance monitor apps can display the real-time CPU and memory usage in such windows.

Usually the floating window is created by a service, so that it can still be floating on screen even if the host app has been brought to the background. Besides, Android system enforces a "solid" layer to block the penetration of user inputs between windows created by different activities, an untouchable floating window that wants the user inputs to pass through (like the screen camera apps) can bypass this restriction only if it is running in a service. A floating window can create fancy UIs and improve user experience. However, it may be used to launch clickjacking attacks if carefully designed by the attacker. In the rest of this paper, the term "floating window" refers to the malicious attack window used to perform clickjacking attacks, unless explicitly specified.

Unlike in an activity where associated views are loaded together with the activity, views in a service have to be added explicitly. This can be achieved using the *addview* API of the *WindowManager*. When calling this API, several parameters can be specified in *LayoutParams* which will affect how the window is laid out, including the width/height, position, general window type, behavioral flags, and desired bitmap format. The size and position are common parameters. As an attack window, the floating window has to occupy the entire available area on screen. This is because only a full-screen floating window (except status bar and navigation bar) can ensure that the user does not get any visual clue of the attack happening underneath (i.e. the loading of the target window). The fifth parameter window format defines the desired bitmap format. Any of the formats defined in *PixelFormat* can be used for floating window. Below we explain the other two parameters: Type and Flags.

1) Window Type: To achieve the goal of floating, we pick out the window types that are prioritized to stay above normal app windows, and permitted to be used in a 3rd-party app (the malicious app), as listed in Table I.

In Android, each window is associated with a Z value (Z-order). Windows with larger Z-order are placed on top of those with lower Z-order. The Z-order of a window is decided by both the window type (Layer Value) and its position in the "window stack". The window management is a complicated task: when there is a change in the "window stack", Z-orders of all windows will need to be re-calculated. However, in terms of detecting a malicious floating window, we only have to know two basic rules: (1) For windows with different Layer Values, the one with larger Layer Value can stay above the lower ones. (2) For windows with the same Layer Value, the most recently launched one will appear on top of the previous ones. In a clickjacking attack, the floating window is launched

TABLE II: Window Flags of Floating Window

Window Flag	Description
FLAG_NOT_TOUCHABLE	this window can never receive touch events.
FLAG_NOT_FOCUSABLE	this window does not need to interact with a soft input method.
FLAG_ALT_FOCUSABLE_IM	invert the state of FLAG_NOT_FOCUSABLE with respect to how this window interacts with the current method.

in advance and needs to cover the target window launched afterwards, so the floating window must have a Layer Value greater than the target window.

In Table I, the window types are listed in an increasing order of the Layer Value. For a given target window, the attacker only has to pick one window type with greater Layer Value. Meanwhile, we noticed that all of the listed window types require SYSTEM_ALERT_WINDOW permission except TYPE_TOAST. According to a study of 1260 malware samples and 1260 top free benign apps [14], SYSTEM_ALERT_ WINDOW permission is not among the top 20 commonly requested permissions of both malware and benign app samples. And it's widely used among some categories of apps including anti-virus apps, locker apps, etc. Still, if the malicious app can avoid using it, the malware could become less suspicious. Note that if a TYPE_TOAST floating window is created with *addView* API, it will not go disappeared after a certain period of time like *Toast.show()* API does. Instead, the floating window remains on the screen untill *removeView* API is called to explicitly remove the window.

2) Window Flag: Flags provide various behavioral options for windows. Unlike the single-choice window type, there can be multiple flags to be set for a given window. We describe the flags relevant to floating windows in Table II.

Since the floating window has to be untouchable, the FLAG_NOT_TOUCHABLE must be set so that all click and touch events will be passed down to the target window. The FLAG_NOT_FOCUSABLE controls whether a given window can take key input focus and other button events (e.g., Back button). If set, these will be delivered to the next focusable window behind. In Android, the input focus can only be given to one window, namely the top focusable window. Hence if the FLAG_NOT_FOCUSABLE is not set for the floating window, the target window will become unfocusable. The FLAG_ALT_FOCUSABLE_IM is to invert the state of FLAG_NOT_FOCUSABLE. If both flags are set, the window will behave as if it needs to interact with the input method; while if FLAG_NOT_FOCUSABLE is not set and this flag is set, then the window will behave as if it doesn't need to interact with the input method.

Generally, the attacker wants to trick the user to operate on the target window with a few clicks. Attacks with a large number of clicks/touches or text input are much harder to accomplish, as the attacker will have to design a series of UIs and ensure that the user will always click on the desired position. The floating window is advised not to set any one of the FLAG_NOT_FOCUSABLE and FLAG_ALT_FOCUSABLE_IM, to keep the input focus from the target window. In this way, even if the user clicks on the text field of the target window (if there is any) by accident, the input method will not be

activated and give sound/vibration feedbacks to alert the user. A special case is that FLAG_NOT_TOUCHABLE and FLAG_NOT_FOCUSABLE will be automatically added for windows of TYPE_SYSTEM_OVERLAY if they are absent (Android thinks that such types of windows must not take input focus, or they will interfere with the keyguard).

B. Target Window

The target window can belong to a 3rd-party app, system app or be other system UIs (e.g., confirmation dialogs). Some of these UIs' functionalities are security-sensitive and require specific permissions to use programmatically, e.g., SMS messenger, camera, system settings, etc. However, the malicious app could launch the target app UI while the floating window is on, and entice the user to manually operate according to the attacker's will (e.g., pressing attractive dummy buttons which sit right on top of actual buttons). As the invocation of the target UI usually does not need any permission, the clickjacking attacks can "achieve" the malicious intention without declaring those sensitive permissions.

The launching of the target UI can be done either programmatically or manually. Manual launch could take lots of extra efforts (e.g. navigating in the app or settings menu). By comparison, the code-based approach will not increase the overall attack complexity, and is much more stealthy. The invocation of the target UI through code requires inter-application communication. Android provides two types of channels for apps to communicate with each other: *Intent*s and *Scheme*s. Furthermore, there are two forms of intents: explicit intents and implicit intents. Explicit intents specify the exported app component to start by class name, while implicit intents declare a general action to match against the intent filters of all exported components (i.e. components that are invokable by other apps). Scheme is implemented based on intent, which allows an app or website to use a URL to invoke another app that has registered the scheme of that URL.

A practical issue for implicit intents and schemes is that when multiple compatible intent filters (from different components) or scheme "handlers" are available, a system dialog is displayed which lists all candidate apps alphabetically and prompts the user to manually select one. Though the malicious app can calculate the target app's position by scanning all components via *PackageManager*, a simple bypass for this is to directly use an explicit intent. We use explicit intent to launch target windows in our experiment.

C. Clickjacking Attack Implementation

So far, we have introduced the fundamental knowledge of the floating window and target window. Next, we give a step-by-step analysis on how to implement a clickjacking attack.

1) Step 1 - Launch the Floating Window: Through former study, we have determined that the first step to perform a clickjacking attack is to launch the floating window. That is, when the malicious app starts to attack, it first adds a full-screen floating window on top of the screen. The type of floating window can be chosen from Table I, whose Layer Value has to be greater than the pre-selected target window. The flags of the floating window must contain FLAG_NOT_TOUCHABLE, while it is highly suggested that FLAG_NOT_FOCUSABLE and FLAG_ALT_FOCUSABLE_IM are not set.

2) Step 2 - Launch the Target Window: The target window is launched right after the floating window, to receive the user inputs. Blinded by the floating window, the user would "click" the dummy object while the click event is actually delivered to the target window. One may consider that as an end for clickjacking attacks. However, such attacks may be discovered immediately after a one-time success. Since the floating window is designed to pass over the user inputs, the malicious app may not be notified of when the input events take place. This is a crucial issue for clickjacking attacks as the user will sense the abnormality if the malicious app cannot "react" in a timely manner to her input action. In order to remain undiscovered, the malicious app must either "listen" to the input taking effect on the target window, or find a way to detect the expected result of the user input.

3) Step 3 - Monitor User Input and Respond: The durability of the malicious app is also an important factor from the attacker's point of view, which has been neglected in all previous works. After the user falls into the clickjacking trap (e.g. click onto the fake UI), the malicious app needs to know as soon as possible when the user input is performed, and react in a way as if the user is really interacting with it. In our work, these "after-attack" disguises are considered. Based on each particular target window, there could be three kinds of ways for the malicious app to track the user input events:

- As the caller of the target window, the target app may issue a callback to the malicious app (e.g., calling the target activity with *startActivityForResult()* and register the *onActivityResult()* callback method).
- The target app (or the system) may issue a broadcast intent to inform all related modules about the specific user action or the phone's state change.
- The malicious app may detect if the expected user action has been made by observing the relevant database.

If it's the first case, the malicious app can directly insert the code for proper response into the callback function. For example, the malicious app wants to take a picture by stealthily starting the camera app with ACTION_IMAGE_CAPTURE intent. After the user is spoofed to click the capture button, the image is returned to the malicious app. Then the malicious app can save or send out the image while on the surface (floating window) starting a game as the user has expected. Note that for camera-based attacks, the phone needs be set to the silent ringer mode *RINGER_MODE_SILENT*, in which sound and vibration will be shut down. The audio volume and vibration state will be recovered (if modified) after the attack.

If the target app does not give a callback, the malicious app will have to seek for other channels. In Android, some of the user actions will be broadcasted so that the relevant apps can keep up with the most recent phone status and adapt themselves accordingly. For example, the malicious app may launch the date settings panel by sending an intent with DATE_SETTINGS action, to trick the user to modify the date (and/or time) so that the scheduled calender events, reminders, and alarming events are all disturbed. The time & date setting is considered security sensitive, the required permission SET_TIME is not granted to 3rd-party apps. However, this protection strategy can be easily bypassed via a clickjacking attack. Meanwhile, the malicious app can register a broadcast receiver with the TIME_SET action to catch the moment that the date/time is changed.

Under other situations where no broadcast channel is available to listen to, the malicious app could register an "observer" for the relevant database as a way to eavesdrop on the user action of interest. For instance, there is a broadcast intent SMS_RECEIVED_ACTION for incoming SMS but no notification is issued for outgoing SMS. However, Android uses content providers to store common data such as contact information, calendar information, media files, messages, call logs, etc. After programmatically filling the phone number and content into a SMS draft, the malicious app can monitor the SMS database (basically a content provider that contains all SMS messages). Specifically, a *ContentObserver* is registered with "content://sms" as the content URL (no SMS-related permission is needed). Since the time that the user takes to click and send the message is very short (we assume the user is spoofed and tempted to click without hesitation), the change in SMS database during that period is sufficient to indicate that the expected "click to send" action by the user has been made.

In either of the three circumstances above, after the user input is detected (the clickjacking attack has succeeded), the malicious app needs to respond instantly and properly, according to the dummy function that the user selected from the floating window. Based on whether to continue the attack, the floating window may be removed (the attack ends) or another target window may be loaded (the attack continues).

D. Potential Risks of Clickjacking Attacks

The threats posed by clickjacking are decided by the various target windows under attack. In general, the ideal target should be invokable through code, which means the component holding the target window (usually an activity) has to be exported (i.e. *android:exported* attribute is set to true). Additionally, the malicious app should be able to monitor the user action on the target window. To evaluate the potential threats, we present the typical clickjacking attacks targeted on system apps, 3rd-party apps and system UIs, respectively.

1) Clickjacking Attacks on System Applications: We first summarize the commonly used public components of system apps that are vulnerable to clickjacking attacks (listed in Table III). The second column shows the component name that can be used to explicitly invoke the target window. The after-attack monitor channel is presented in the forth column. As we have found, the target windows of all victim components are of the same type TYPE_BASE_APPLICATION, which is the normal window type. All these target windows have the same Layer Value 2, which means they will be shielded by any of the floating window types listed in Table I. An exception is that sometimes user operations may trigger toast messages (TYPE_TOAST). For instance, after the user modifies a contact, a toast message "Contact saved" will be displayed. When launching attacks associated with toast messages, the floating window type must have a Layer Value greater than TYPE_TOAST.

The target windows could belong to media apps (camera, sound recorder), telephony apps (dialer, messaging, contact), the package installer, the settings or the file explorer. By tricking the user to click on these windows, the attacker is able to perform security sensitive operations behind the

Target app	Target Component	Intent Action (Scheme)	Monitor Channel	Permission Bypassed
Camera	CameraActivity	IMAGE_CAPTURE	Callback	CAMERA
Camera	VideoCamera (alias)	VIDEO_CAPTURE	Callback	CAMERA
Contact	ContactSelectionActivity	PICK	Callback	READ_CONTACTS
Contact	ContactEditorActivity	EDIT	Listen to contact database	WRITE_CONTACTS
Dialer	DialtactsActivity	DIAL	Listen to contact database	CALL_PHONE
Documents	DocumentsActivity	GET_CONTENT	Callback	READ_EXTERNAL_STORAGE
Package installer	PackageInstallerActivity	INSTALL_PACKAGE	Callback	INSTALL_PACKAGES (system only)
	UninstallerActivity	UNINSTALL_PACKAGE	Callback	DELETE_PACKAGES (system only)
Settings	WirelessSettingsActivity	WIRELESS_SETTINGS	Broadcast receiver	WRITE_SETTINGS
	WifiSettingsActivity	WIFI_SETTINGS	Listen to settings database	CHANGE_WIFI_STATE
	DateTimeSettingsActivity	DATE_SETTINGS	Broadcast receiver	SET_TIME (system only)
Sound recorder	SoundRecorder	RECORD_SOUND	Callback	RECORD_AUDIO
Messaging	ComposeMessageActivity	SENDTO (smsto/mmsto)	Listen to SMS/MMS database	SEND_SMS

user's back. These operations are normally guarded by specific permissions (listed in the "Permission Bypassed" column of Table III), some are not available for 3rd-party apps (IN-STALL_PACKAGE, DELETE_PACKAGES and SET_TIME). From this perspective, the clickjacking attack actually gives the malicious app a permission escalation [15] [16], by having the privileged system apps delegate the sensitive operations (performed by the user). The system permissions usually protect APIs that are extremely security sensitive. For example, INSTALL_PACKAGE permission allows an app to silently install other apps. If a malicious app acquires this capability (even if indirectly via clickjacking), it can install other mal-wares and cause more serious consequences.
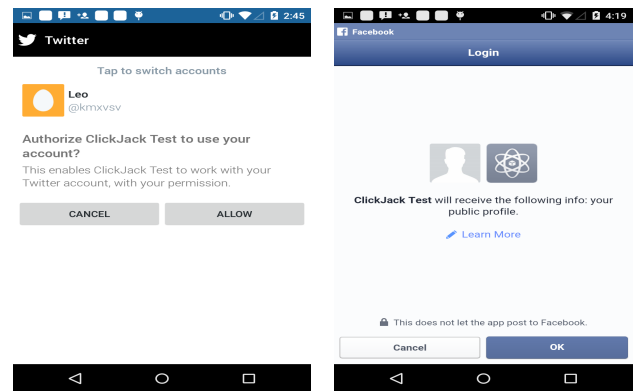
2) Clickjacking Attacks on 3rd-party Applications: We also look into the 3rd-party apps and check if their APIs could be targeted by clickjacking attacks. It has been reported in [17] that many 3rd-party apps unintentionally expose their components that may be utilized by attackers. In our work, we mainly consider apps that explicitly share their APIs, e.g., apps that release their own SDKs for other apps to use their services. This is quite popular especially among social networking apps which allow users to quickly spread information via their social networks. We use Twitter and Facebook as examples to check their potential vulnerability to the clickjacking attacks.

The Twitter and Facebook SDKs provide a set of APIs to perform commonly used operations like tweet, update mood, share photos/links, load contents from Twitter/Facebook, etc. We implement the basic "tweet" and "share link" functions on Twitter and Facebook, respectively, to demonstrate the feasibility of launching clickjacking on these apps. We suppose that the user has already logged into the client app, so that the malicious app can be authorized via a couple of clicks.

In Twitter, a *TwitterLoginButton* is placed in the target window (also launched by the malicious app). When the user is tricked to click the *TwitterLoginButton* , she will be brought to the Twitter authorization window (shown in Figure 2(a)), where the user is then tricked to click the "allow" button so that the malicious app is authorized to utilize Twitter APIs (e.g., perform a post). The post operation can be conducted programmatically as soon as it is authorized. Specifically, the malicious app calls the *getStatusesService()* method to get a *StatusesService* object for tweet post requests. Then a message is composed, and posted by calling the *update()* method of the *StatusesService* object.

The general procedure of attacking the Facebook client app is similar to Twitter. But the click on the login button is not necessary, since the actual method of logging in is private with the Twitter SDK, whereas with the Facebook SDK it is public. Hence, the malicious app can directly bring the user to the Facebook login authorization window (shown in Figure 2(b)) to gain user's authorization via clickjacking. Then, the posting action on Facebook, however, requires explicit user approval. After these two authorizations, the malicious app will have the privilege to post. For example, a *GraphRequest* object can be created to post a shared link to the user's wall ("me/feed").

(a) Twitter Login Authorization   (b) Facebook Login Authorization

Fig. 2: Clickjacking Attacks on 3rd-party Client Apps

In sum, the unauthorized posting (user is not aware of) on Twitter and Facebook both require two clicks, which is easy to achieve by clickjacking attacks. The authorization window will also give a callback to the caller app, so the malicious app can know the user input event instantly and react accordingly. Note that the explicit authorization is required only once (in the first time the malicious app tries to connect with the client app). After that, even if the malicious app has been closed, it can still directly perform operations through APIs.

3) Clickjacking Attacks on System UIs: We find that the Android system UIs are vulnerable to clickjacking attacks as well. A particular example is the screen recording API: starting from Android 5.0 Lollipop, a new "android.media.projection" API is created to facilitate the screen capturing and screen sharing, without the need of connecting the device to PC over the Android Debug Bridge (ADB) as required by the previous Android 4.4 KitKat. Specifically, the *createVirtualD-isplay()* method of the "MediaProjection" class starts a screen capture session to capture the contents of the main screen into a *Surface* object. The invocation of this API does

not require any permission, but a confirmation dialog of TYPE_BASE_APPLICATION will pop up asking for user approval to start. In a clickjacking attack, the user may be spoofed to click right on the "start now" button without realizing that the screen recording is being activated. Unlike attacking system apps (e.g., calling the camera to shoot a video), the ending of the screen recording does not need user operation (manually click to stop). The malicious app can just set a timer (in a service) to release the *VirtualDisplay* after a certain amount of time. Hence, the whole attack process can be very simple and stealthy. By default, the recorded video is sent to a *Surface* object that can be displayed instantly. If the attacker wants to obtain the video, it can be saved as a video file using *MediaRecorder*. An alternative way is to extract images from the buffer by setting an *ImageReader* object as the callback of *createVirtualDisplay()*, in a frequency high enough to observe all user actions.

Screen recording is considered as a sensitive API, as all user actions will be logged. Apart from the violation of user privacy, a even more serious consequence is the leak of credentials (e.g., user name, password, credit card details) as the screen recording will capture the login and transaction processes as well. Usually, the account name entered stays unmasked while the password will turn into asterisks or bullets after a short moment of time (e.g., half second), which is long enough for the attacker to recognize each letter/number.

V. COUNTERMEASURE

The design of our defense scheme to clickjacking attacks is based on the features summarized in Section IV. To overcome the limitations of existing solutions, we add an independent module to the Android system for automatic detection of click-jacking attacks. With our system-level and real-time protection, users are relieved from checking a security indicator whenever a window pops up and developers no longer need to worry about the incompatibility side-effect caused by touch filtering. Besides, the defense scheme is much more flexible than enforcing a solid security layer between each two neighboring apps, benign pass-through inputs will not be blocked.

We start with the two essential windows in clickjacking attacks. The camouflage window must be floating, untouchable and preferably focusable. Another factor that has never been considered in previous works is the transparency (controlled by the *alpha* attribute of window layout parameters). A floating window in a benign app is usually auxiliary for the user or other apps, hence is set to semitransparent (e.g. screen camera apps); while in clickjacking, it is used to cover the target window and has to be opaque or slightly transparent. By default, we can set an empirical threshold value 0.95 for *alpha* in the detection, considering that the user can barely see through the window with an *alpha* value as high as 0.95. Users can modify this threshold according to their preferences. In contrast, there is no common feature for target windows since the target can be any normal-type app window or system UI. Hence, the identity of the target window is not clear until a clickjacking attack is launched.

Next, we analyze the dynamic features of clickjacking. An important pattern is that a successful clicking attack will always end with a user click. This means the detection is not necessary to be performed every time a window shows up,

but instead, only when an input event is being received. Note that here we mean the input has just been received by the operating system and has not been delivered to the target app. The detection logic is to presume the window that has received the input as the target window, then check if it is covered by any malicious floating window of another app. There could be multiple windows above, all of them need to be checked since we are not sure which one is actually displayed to the user (fake floating windows with full transparent view or minimized size may exist on top to "fool" the detector). If any of these windows matches the features of a malicious floating window, it is considered as a clickjacking attack.
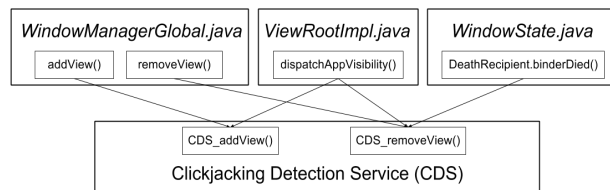

Fig. 3: CDS and the Four Probes

To check if there is a floating window among windows above, the detector should keep a record of all visible windows. Since the "window stack" in *WindowManagerService* is not maintained in the Z-order, a full re-calculation of the Z-order of all windows is required even if we are only interested in (malicious) floating windows. To avoid the unnecessary work-loads, we create an independent clickjacking detection service (CDS) in parallel with other system services (e.g., *ActicityManagerService*, *WindowManagerService*, etc). Meanwhile, four probes are set up to catch window actions. As depicted in Figure 3, the four probes are hooked to the *addView()* and *removeView()* methods of the *WindowManagerGlobal* class, *dispatchAppVisibility()* method of the *ViewRootImpl* class, and the *binderDied()* method defined in the *WindowState* class. The *addView()* and *removeView()* methods are invoked when views are created/destroyed along with their hosting activity, or added/removed explicitly through *WindowManager* APIs. The *dispatchAppVisibility()* method is called when the hosting activity becomes visible/invisible (hence so as the associated views). A special situation is that an app is uninstalled when its view is being displayed. In this case, the *binderDied()* callback method of the view is invoked and a corresponding removal will be performed at *WindowManagerService*. By hooking these four methods, CDS is able to collect all window actions including "add/remove" and "turn visible/invisible". Specifically, the creation and turn-to-visible actions go to the *CDS_addView()* API while the removal and turn-to-invisible actions go to the *CDS_removeView()* API.

In Android, each time a view is added through *addView()* API, a *ViewRootImpl* object is created in the app process along with the window parameters and an *InputChannel* object (used to deliver input events to the destined window). Then these information are sent to *WindowManagerService* so that a corresponding *WindowState* object is created. Unlike the window management service *WindowManagerService*, the CDS is specified to detect malicious floating windows for clickjacking. Obviously, it only needs to keep the information related to floating detection rather than the whole *WindowState* objects. Hence, we define a new class *WindowCDS* to record windows in a floating-specific representation. Specifically, all

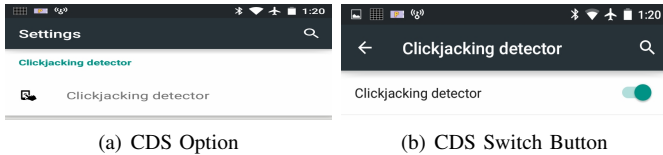(a) CDS Option   (b) CDS Switch Button

Fig. 4: Customized Settings Menu

floating-related window attributes are kept as instance variables including window type, window flags, and *alpha*. In addition, the package name is needed to determine if the "floating window" and the "target window" actually belong to the same app (not considered as an attack). Furthermore, a unique ID is required to identify windows in CDS. However, Android does not provide a unified ID for windows/views. We resolve this issue by extracting the number contained in the *InputChannel* object's name as the window ID (all windows have different *InputChannel*s). Note that even if a window is untouchable/un-focusable, it is still assigned with an *InputChannel*.

The *WindowCDS* objects are stored in the CDS window list according to the Z-order. When a new window is inserted, it will be placed according to the Layer Value of its window type (defined in the *PhoneWindowManager* class). If there exists other windows with the same Layer Value, the new one will be placed ahead of the old ones. The removal of a window is simple: just search the ID in the window list, and remove the one with the same ID; however, it may happen that the window to be removed is not found (already removed). This is because sometimes the window is turned invisible before it is removed, which means the *CDS_removeView()* API will be called twice. Hence, "window not found" is considered acceptable in CDS, no exception will be given.

So far we are able to maintain the windows information in CDS. To catch the input events, we add a hook to the *onInputEvent()* callback function of the *WindowInputEventReceiver*. In each *ViewRootImpl* object, a *WindowInputEventReceiver* is registered along with the creation of the *InputChannel*, to receive input events from the low level input dispatching module. In *onInputEvent()* function, the inputs are added into the input event queue and finally delivered to respective apps. Our defense strategy is to perform the checking for clickjacking attacks before the input event is handed over to the client app. Specifically, upon receiving an input event, the ID of the receiver window and the base package name are sent to the CDS, to check if it is under clickjacking attack. The input will stay suspended until the checking is cleared. In CDS, a malicious floating window is confirmed with four conditions: (1) its host app is different from the receiver window's; (2) its Layer Value is greater than the receiver window; (3) its flags contains FLAG_NOT_TOUCHABLE; (4) its *alpha* value is greater than 0.95. If no such window is found, it means no clickjacking risk is detected and the input event can be released to the receiver app. Otherwise, the user will be alerted.

VI. EVALUATION

In this section, we evaluate the performance of the proposed defense scheme through extensive experiments. Although currently no real-world malicious clickjacking app is available on market, we believe this is a practical and potentially dangerous problem like previous works [9] [11]

[12] [13]. We implement our clickjacking defense mechanism on a Nexus 4 smartphone running Android 5.0 system, and we add a switch button in the settings menu (as shown in Figure 4) to enable/disable the CDS (by default it is enabled).

A. Effectiveness

Our defense scheme is able to detect clickjacking attacks on all different target windows considered in this paper. Figure 5 shows the UI of a malicious game app "Space Battle" we developed for illustration. This UI is actually a floating window, and the user input will be delivered to the target window below which is the confirmation window of screen recording. With the CDS enabled, as the user clicks the play button, an alert window will be issued to warn the user about the clickjacking attack detected. The alert window is of type TYPE_SYSTEM_ERROR, hence it will not be shielded by the floating window. Meanwhile, sound and vibration alerts are also made to catch the user's attention. The name of the suspected app is displayed in the title. The user can directly uninstall the app by clicking "OK" button in the alert window. Note that our defense scheme can deterministically decide if an app is launching a clickjacking attack. This warning dialog is not requesting user's own judgement. Besides, we also test with common benign apps, including the locker apps, apps with a small "always visible" icon, and screen camera apps that utilize floating windows. All of them are correctly recognized as benign by CDS, and no false alarm is generated.
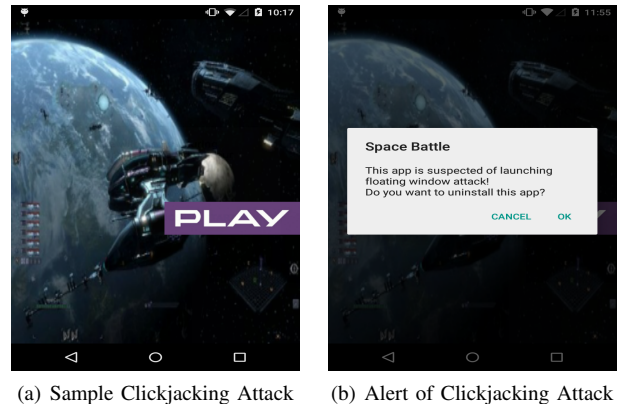

(a) Sample Clickjacking Attack   (b) Alert of Clickjacking Attack

Fig. 5: Clickjacking Attack and Defense

B. Overhead

CDS is running as an independent service which manages the floating-related information for windows, and provides the checking of clickjacking attacks for each user input event. As a much simplified version of *WindowManagerService*, the collection of floating-related information in CDS includes only five variables (window type, window flags, *alpha* value, package name, and window ID). The overhead for collecting these variables is too trivial that our experiments cannot give a convincing measurement, so we only present the overhead of the checking procedure. Figure 6 shows the average execution time of the checking process in CDS, the checking is repeated for 10 times for each target window: on UI of system apps, it takes 5.2 to 10.5 ms; on UI of 3rd-party apps like Facebook and Twitter, it takes 14.1 and 12.2 ms respectively; on system UI like screen recording dialog, it takes 16.7 ms. The time delay caused by clickjacking detection is considered acceptable.

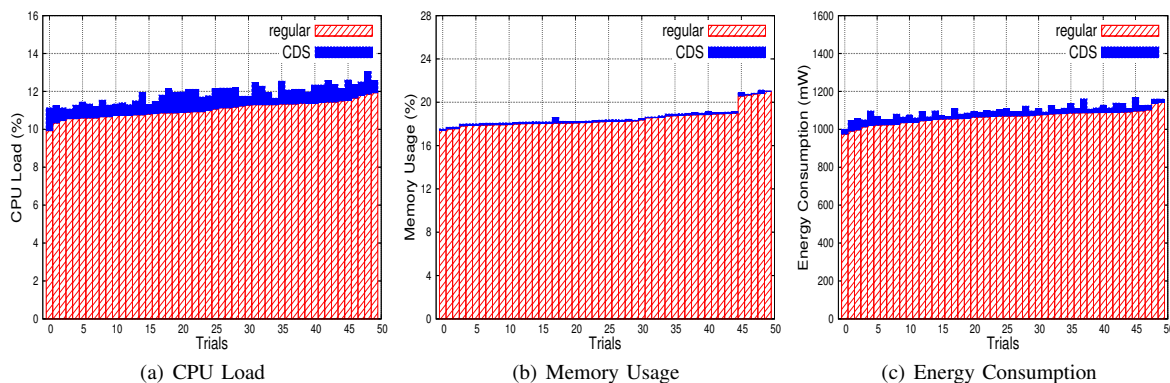(a) CPU Load   (b) Memory Usage   (c) Energy Consumption
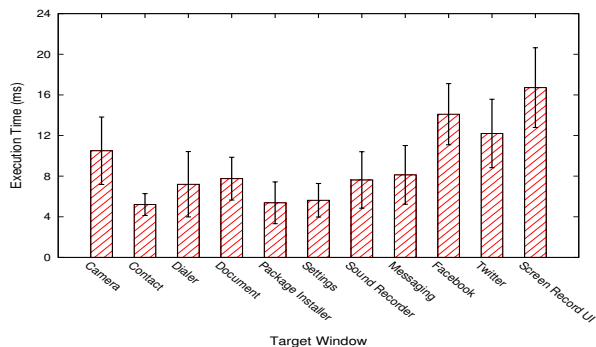
Fig. 7: Overhead of the Checking Procedure

Fig. 6: Execution Time of the Checking Procedure

Next, we measure the real-time CPU load, memory usage and energy consumption per click event when clickjacking detection is enabled/disabled. The CPU load is measured by a system monitor app Tinycore, the memory usage is obtained via the *getMemoryInfo* API, while the energy consumption is measured using the power profiling app Trepn Profiler. All three metrics are measured for 50 trials (groups). Each CPU and memory trial is the average of 90 readings, while each power consumption trial is the average of 10 readings in a one-minute profiling. As depicted in Figure 7, the 50 groups of measurements are drawn by the increasing order of the regular overhead (without CDS). The solid part on top of the bar represents the extra overhead brought by CDS. Our results indicate that CDS causes a mean increase of 0.84% in CPU load, 0.145% in memory usage, and 3.21% in power consumption (relative increment). The standard deviation over 50 trails is 0.239%, 0.076%, and 1.487%, respectively. Overall, our clickjacking detection scheme is quite lightweight, particularly regarding its CPU and memory overhead. Note that currently there is no accurate and direct measurement of energy consumption available for phones, our result on power consumption is an approximate reflection of CDS's impact.

VII. Conclusion

In this paper, we conducted a comprehensive study on mobile clickjacking attacks. We presented the key steps to implement a stealthy clickjacking attack and explored its potential threats. Finally, an automatic countermeasure for clickjacking attacks is implemented and evaluated. The experimental results show that our defense scheme is effective and lightweight. In our future work, we will improve the CDS by fingerprinting the opened-windows state, so that the checking is not performed when the window hierarchy is unchanged.

References

[1] Gartner, "Worldwide smartphone sales to end users by operating system in 4q15," http://www.gartner.com/newsroom/id/3215217, Feb 2016.

[2] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks," in *Proceedings of USENIX Conference on Security Symposium*, 2014.

[3] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh, "Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks," in *Proceedings of the 4th USENIX Conference on Offensive Technologies (WOOT)*, 2010.

[4] L.-S. Huang *et al.*, "Clickjacking: Attacks and defenses," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[5] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *Foundations and Practice of Security*, 2013, vol. 7743, pp. 227–243.

[6] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited: A perceptual view of ui security," in *8th USENIX Workshop on Offensive Technologies (WOOT)*, San Diego, CA, Aug. 2014.

[7] J. A. Shamsi *et al.*, "Clicksafe: Providing security against clickjacking attacks," in *Proceedings of IEEE 15th International Symposium on High-Assurance Systems Engineering*, Jan 2014.

[8] H. Shahriar and H. Haddad, "Security assessment of clickjacking risks in web applications: Metrics based approach," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, 2015.

[9] A. Bianchi *et al.*, "What the app is that? deception and countermeasures in the android user interface," in *Proceedings of IEEE S&P*, May 2015.

[10] L. Wu, X. Du, and J. Wu, "Effective defense schemes for phishing attacks on mobile computing platforms," *IEEE Transactions on Vehicular Technology*, Aug 2015.

[11] M. Niemietz and J. Schwenk, "Ui redressing attacks on android devices," in *Black Hat Abu Dhabi*, 2012.

[12] M. Niemietz, "Ui redressing attacks on android devices revisited," in *Black Hat Asia*, 2014.

[13] E. Fernandes *et al.*, "Android UI Deception Revisited: Attacks and Defenses," in *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, February 2016.

[14] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of IEEE S&P*, May 2012.

[15] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th International Conference on Information Security*, 2011.

[16] A. P. Felt *et al.*, "Permission re-delegation: Attacks and defenses," in *Proceedings of USENIX Conference on Security Symposium*, 2011.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of ACM MobiSys*, 2011.