Android UI Deception Revisited: Attacks and Defenses

Earlence Fernandes, Qi Alfred Chen, Justin Paupore,
Georg Essl, J. Alex Halderman, Z. Morley Mao, Atul Prakash

{earlence,alfchen,jpaupore,gessl,jhalderm,zmao,aprakash}@umich.edu
University of Michigan, Ann Arbor

Abstract. App-based deception attacks are increasingly a problem on mobile devices and they are used to steal passwords, credit card numbers, text messages, etc. Current versions of Android are susceptible to these attacks. Recently, Bianchi *et al.* proposed a novel solution *"What the App is That"* that included a host-based system to identify apps to users via a security indicator and help assure them that their input goes to the identified apps [7]. Unfortunately, we found that the solution has a significant side channel vulnerability as well as susceptibility to clickjacking that allow non-privileged malware to completely compromise the defenses, and successfully steal passwords or other keyboard input. We discuss the vulnerabilities found, propose possible defenses, and then evaluate the defenses against different types of UI deception attacks.

1 Introduction

App-based user-interface (UI) attacks pose an increasing threat to smartphone users [30,9,29]. In such attacks, a malicious app tricks the user into entering sensitive input into a window the malware controls or providing *incorrect* input into a window the malware does not control. UI attacks are particularly serious since they collect or control information at the end point closest to the user. Once a malicious app gets a foothold on a mobile device, it is possible for it to steal credentials and cause the user to grant additional privileges, totally compromising the device.

One class of such attacks is app-based phishing. For example, *svpeng* is a malicious mobile app that deceives the user into providing sensitive information, including bank credentials and credit card numbers, to a window that the attacker controls (see Figure 1). This deception can be stealthy and convincing due to the rich ways in which modern mobile apps share the screen. We discuss other types of attacks in Section 2.

UI deception attacks are possible due to two primary reasons: (a) The smartphone GUI environment is complex and allows for intricate interactions between GUIs to support a multitude of use cases, and (b) Users cannot verify the provenance of an app (or window) they are interacting with. A few approaches have been attempted to defend against these attacks, for example DECAF [23] uses

dynamic app exploration to detect ad UI fraud, and Liu *et al.* [24] use OCR techniques to automatically detect spoofed keyboards. However, these techniques are limited to specific types of UI attacks, and are usually based on heuristics and not foolproof.

A more general defense approach against UI deception was recently proposed by Bianchi *et al.* and involves a two-layered defense consisting of a static analysis and a runtime security indicator modeled after the well-known HTTPS lock icon and EV infrastructure [7]. Bianchi *et al.* conducted a user study that established that the security indicator helped 76% of users correctly identify UI deception under a particular attack setting.

Since Bianchi *et al.* strike directly at the root cause of the problem—lack of attribution, we believe this is the right approach to tackling UI deception. Therefore, our aim is to investigate the security properties of this defense in further detail.

The first part of our work finds that the state-of-the-art defense is still vulnerable to a subtle form of clickjacking attack that we built. Our clickjacking attack exploits a race condition and uses a newly discovered IPC side-channel to precisely time the display of attack windows to steal user input without causing any change in the visualization of the security indicator.

The second part of our work discusses challenges in achieving a secure defense against UI deception. The primary challenge is ensuring temporal integrity of the security indicator in a seamless and correct way. To that end, we introduce the concept of an *Overlay Mutex*, whose purpose is to give the user a guarantee that no other windows may interfere with the current GUI the user is interacting with. The following are our contributions:

1. *New Attacks:* We analyze the security of a state-of-the-art defense against UI deception and discover a subtle class of clickjacking attacks. Furthermore, we introduce a new IPC-based side-channel on Android and use it to elude security checks proposed in prior defenses. To the best of our knowledge, this is the first known exploitation of Android Binder IPC statistics to enhance UI deception (Sections 3, 4). We have uploaded demo videos [13], and proof-of-concept code [1] of our work.
2. *New Defenses:* We introduce the Overlay Mutex as a UI stack modification and discuss how it achieves temporal integrity for the security indicator, even in the presence of sophisticated, side-channel based clickjacking attacks. Our design is intended as an addition to the work of Bianchi *et al.*, thereby extending the state of the art in UI deception defense (Section 5).
3. *Evaluation:* We evaluate our defense against known UI deception and against our new side-channel based clickjacking attack and find that the overlay mutex successfully ensures temporal integrity of the security indicator, helping to block clickjacking attacks.

2 Threat Model and Example UI Attacks

Our threat model assumes that the Android OS is not compromised via root exploits. Malicious apps are assumed to be unprivileged, which is the norm on

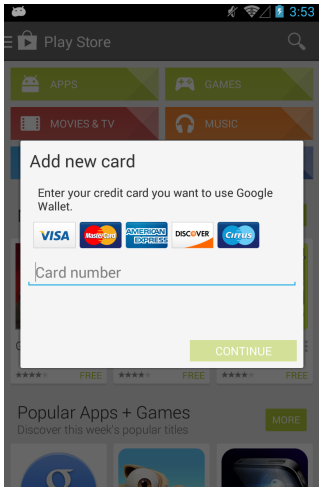

Fig. 1. The svpeng malware overlays Google Play with a phishing window soliciting credit card details.

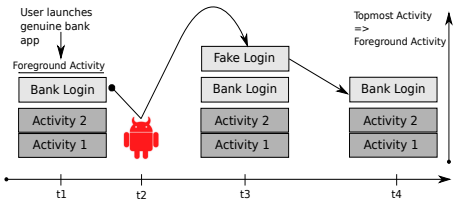

Fig. 2. An attack scenario to steal banking passwords. Alice launches the genuine bank app at t1. Mallory, the attacker, detects a change in the activity stack at time t2 and immediately injects the fake login activity at t3. After Alice provides the password, Mallory's fake login activity removes itself simulating a failed login attempt and reveals the real login activity. Ideally, Mallory wants (t2 - t1) and (t3 - t2) to be small. Real attacks achieve this in approximately 45ms.

Android. Prior work showed that even unprivileged apps on Android can spoof other apps to steal input, overlay windows of other apps, and perform activity hijacking attacks to inject overlay windows when passwords are entered [11,17,7]. We also note that Trojan apps have been successfully distributed on the Play Store [22,33]. We assume that soft-keyboards are not compromised or malicious. Recent research has made strides in achieving this goal [10]. We now discuss examples of powerful attacks that are possible under our threat model.

Direct Phishing. This type of attack pops up a window soliciting sensitive information, such as passwords, by presenting a malicious window (only associated with a background service) that looks exactly like a trusted app window. A sophisticated example of this attack is mentioned by Felt *et al.* [8], where a malicious app embeds a Facebook Like button. When the user clicks on this button, the attacker presents a malicious Facebook login window copy and steals login information. Other variants involve displaying attack windows out of context to the user and mimicking trustworthy screens.

Activity hijacking. A recently discovered malware strain—Trojan-Banker.AndroidOS.Svpeng.A [21], a member of the well-known svpeng family performs context-sensitive phishing. Figure 1 is a screenshot of the attack in action. Once on the device, it waits until the screen displays the `AssetBrowserActivity` of the genuine Play Store and then pre-empts the UI to display its fake version of the authentic credit card dialog.

The general structure of such attacks is shown in Figure 2. The crucial parts involve detecting a state-change in the UI and precisely timing the launch of a spoofed activity soliciting credentials. This kind of attack is context-sensitive because users are lulled into a false sense of security since they are within a

genuine app. However, the attacker deceives users through a quick and well-timed overlay [3,4,12]. Unprivileged malware can also use a recently discovered side-channel that leaks UI state transitions to build precisely timed UI deception attacks [11].

Clickjacking. In a classic clickjack attack, an attack UI element is displayed just in time to steal a portion of user input without alerting the user [20], [5]. On Android, different kinds of clickjacking is possible. Attackers can use Toast windows (transient windows) to steal input and let that input passthrough to the underlying window. Attackers can also randomly display transparent windows that grab input and do not let it go to the underlying window, thus stealing portion of the input and simulating a random failure that explains why an input did not appear on the underlying window. This attack is interesting because even in the presence of security indicator-like defenses, users might miss changes because the attack relies on quickness of UI transitions and the inability of the human motor system to easily cancel actions once issued [5].

3 What The App Is That?

Recently, Bianchi *et al.* introduced a defense against UI deception [7] (What the App is That—*WhatTheApp* for short in this paper). This system consists of two portions: (1) a static analysis portion that is used to flag apps that may perform UI deception attacks for execution at an App Store and (2) a security-indicator based Android system in which apps are EV-certified and the security indicator provides a visual indicator of any UI deception attempts by background apps. Bianchi *et al.* note that while the static analysis may catch some instances of UI deception, the analysis is intended as a market-level tool and not as a complete defense against UI deception. In fact, the XCodeGhost attack on the Apple vetting process is clear evidence that on-device solutions are necessary [2].

Bianchi *et al.* address that need by including an on-device defense mechanism, partly based on the security indicator design of the HTTPS lock icon and EV bar in browsers. They propose a clever security bar for mobile devices that uses a reserved portion of the screen to identify the top activity to users as well as indicate via a lock icon whether the display is tampered with (Figure 3). They conducted a human subjects study where 76% of participants correctly determined when UI deception attacks took place under an attack setting. Based on a security indicator design in [17], the security bar consists of a security image only known to the OS to further prove the authenticity of the indicator. Based on the HTTPS lock icon, they included a lock icon which could be green (no tampering of display contents), yellow (display possibly tampered with), or red (non-certified app executing).

Listing 1.1 shows the core logic for managing the contents of the bar, based on the code that they have publicly released for their system. This core logic executes once per second and determines the color of the lock icon and the identity of the app the user is interacting with. The logic works as follows: (a) if the top activity is owned by `SYSTEM`, it displays a green lock and an indicator

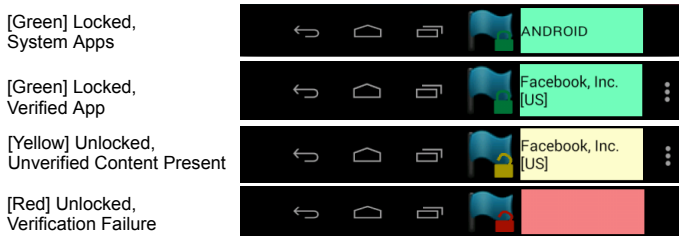

Fig. 3. *WhatTheApp* visualization for various screen states. The first image is for system apps, the second for a verified app, the third is shown when a window is overlayed on a verified app and the last visualization is when an unverified app is opened.

```
1  //execute every second
2  LockStatus getVerificationStatus() {
3    if (topActivity identity is SYSTEM)
4      return GREEN_LOCK_ICON
5    else
6    {
7      if(topActivity identity is not VERIFIED)
8        return RED_LOCK_ICON
9      else if(topActivity identity is VERIFIED)
10     {
11       if(current WindowStack only contains windows of topActivity and SYSTEM)
12         return GREEN_LOCK_ICON
13       else
14         return YELLOW_LOCK_ICON
15     }
16   }
17 }
```

Listing 1.1. Logic of UI defense proposed in [7].

Table 1. Effect of a transient window when the user is in a verified app.

Condition	Visualization
Verified App	Green Lock Icon, Verified App Identity
Malware window appears	Yellow Lock Icon, Verified App Identity
Malware window hidden	Green Lock Icon, Verified App Identity

displaying "ANDROID." Thus, the user should be able to trust that the input provided will go to a system application. (b) Otherwise, it verifies whether the app for top-level activity is EV-certified. If not, a red lock icon is displayed (i.e., an unverified app is the top activity). Otherwise, the name of the app from the EV certificate is displayed. An additional check is made to ensure that all the windows in window stack are owned by the verified app. If not, a yellow lock is displayed. Otherwise, a green lock is displayed. Green lock is meant to convey to the user that the entered input will go to the app identified in the security indicator. We found the prototype code to be consistent with the paper.

4 *WhatTheApp* Vulnerabilities—Timing Attacks and Side Channels

Design choices that lead to attacks. The *WhatTheApp* design choice to verify the screen contents at a particular frequency embodies the time-of-check-time-of-use design flaw, because the screen contents can change just after a check is complete, forcing the indicator to display stale information for the user.

Furthermore, the design choice to allow arbitrary window pre-emptions requires the user to *always* be cognizant of indicator contents—prior research discusses that users do not constantly pay attention to security indicators [15]. Ideally, a user should only check an indicator once before commencing a security sensitive operation.

Clickjack attacks. We found that a malware app can bypass the periodic check in *WhatTheApp* by quickly rendering a rogue window on top of an app's UI and then hiding that window. Often, one tap on the soft keyboard is successfully stolen. To the end user, this appears as a random fluctuation in input and the user may not even notice the absence of a single character—missing input can be auto-corrected or, for passwords, the user may simply retype the password on an error. For passwords, as a result of malware stealing some characters, bruteforce attacks become easier due to entropy reduction.

We tested all our attacks on an Android emulator running Android 4.4 with *WhatTheApp* modifications, the same version as published by Bianchi *et al.* To make the malware inject the rogue window only when password windows were displayed, we used a previously known side channel on Android that is described in [11] that allows monitoring the window transitions of other apps by unprivileged malware.

In practice, we found the attack to be only partly effective since, at times, the lock icon would turn yellow if the check logic of Listing 1.1 happens to trigger while the rogue window is displayed (hand caught in the cookie jar, so to speak). Our attack used the `WindowManager.addView` API to quickly create attack windows whenever an app we wished to attack became a foreground app and displayed the password window[1]. The detailed effect of showing and hiding a malware deception window while a user is interacting with a verified app is shown in Table 1.

Leveraging new side-channels. We discovered a new side-channel that makes the previous attack very effective—in most cases, it was missed entirely by *WhatTheApp* defenses. The side channel enables a background malware app to discover whenever a security check occurs, enabling it to predict the approximate time of the next check and thus precisely time the display of a phishing window to occur during an interval prior to the next check. The end effect was that malware could steal user's input without causing any changes in their security bar (the lock icon stayed green during our tests).

Our side channel exploits the publicly available information on the timing of the binder IPC calls. Binder is the *de facto* Android IPC mechanism. The security bar in *WhatTheApp* runs as its own process and makes a binder IPC call to verify screen contents every second using a timer. A test malware app monitored the timing of the binder IPC calls and inferred the ones that came from the *WhatTheApp* security indicator. To the best of our knowledge, this is the first exploitation of binder calls on Android for a side channel.

[1] We chose to attack Facebook in our examples, though it could equally well have been a banking app.

```
1 587398: call  from 1323:1455 to 1141:0 node 585 handle 34 size 72:0
2 587399: reply from 1141:1389 to 1323:1455 node 0 handle -1 size 212:4
3 587400: async from 1323:1323 to 928:0 node 1341 handle 22 size 80:0
4 587401: call  from 1323:1323 to 928:0 node 2428 handle 38 size 100:0
```

Listing 1.2. Sample binder transaction logs. Each line corresponds to a single transaction. Each line is of the form (debug id), (txn type), (source Pid/Tid), (dest. Pid/Tid), (bookkeeping data). The first 2 lines contain relevant transaction logs from the security bar to the security check service.

In particular, binder IPC calls are referred to as binder transactions. In a single call-return IPC, there is one transaction for the outgoing call and one transaction for the incoming reply. The binder kernel driver publishes debugging statistics to `/sys/kernel/debug/binder`. There are several publicly accessible logging files under this directory. These files are present on factory images of real devices. Of particular interest is the `transaction_log`[2] file that lists a history of transactions occurring between processes on the system. Sample transaction logs are shown in Listing 1.2. For every transaction, there is a line containing a debug identifier, transaction type (call, reply, async), source process and thread id, destination process and thread id, and other bookkeeping information. Therefore, our test malware app can determine that a transaction occurred between two processes, but it does not know the name of the called remote method. The system server on Android hosts multiple important services, including the window manager. Any process hosting UI makes several calls a second to the window manager portion of the system server. Thus the attack app has to disambiguate and locate the security check call by *WhatTheApp* from other events in the log.

To do that, our test malware measured the sizes of the transaction data and discovered that the call-reply transaction pair corresponding to the security check had unique values for the sizes at approximately one second intervals. This allowed the test malware to determine the transactions of the security indicator (nav bar) display process and the security check service. The test malware then measured the time intervals between transactions for these two processes by sampling the log file at an interval of 50 ms. We determined that the interval between consecutive events was approximately 1 sec +/- 600 ms. In some cases, the error was upto 3 seconds. This error is due to scheduling policies, context switches, and memory pressure. Thus, the side channel we discovered was relatively noisy.

Experimentally, we found that the malware could overcome the noise in the side channel by only performing attacks when the channel indicated relatively stable time intervals between the checks (Listing 1.3). Our test malware computed the time interval between the previously observed security check and the latest observed security check. Ideally, we expect this delta to be close to 1 sec-

[2] At the time of writing, this file was publicly accessible to any app. Recently, the current version of Android (Marshmallow) tightened its SELinux policy to prevent arbitrary apps from accessing the file.

```
1  void sideChannelAttack() {
2    while(shouldAttack()) {
3      TempTxnLog = getTxnLog("/sys/kernel/debug/binder/transaction_log")
4        (srcPid, destPid) = getPids("EVBar", "VerifierService")
5        TxnLog = filterLogs(srcPid, destPid)
6        if(TxnLog contains new transaction from srcPid to destPid)
7        {
8          delta = NOW - prev_txn_timestamp
9          if(delta >= 800 AND delta <= 1200)
10           show_attack_window(400)
11
12         prev_txn_timestamp = NOW
13       }
14
15     sleep(50)
16   }
17 }
18
19 boolean shouldAttack() {
20   TopOfStackActivity = readTopOfStack()
21   if(TopOfStackActivity == "com.facebook")
22     return true
23   else
24     return false
25 }
```

Listing 1.3. Control algorithm exploiting Binder side channel.

Table 2. Effect of using a side channel to predict attack times. Tests were run for 6 hours.

Condition	Successful Attacks/Total Attacks	%Success
No side channel (random)	5475/11199	48.8%
Side Channel	9923/10783	92%

ond (based on public code for *WhatTheApp*). If the delta value was between 800ms and 1200ms, then the malware assumed that side channel information was relatively stable and it launched an attack window; otherwise, it waited for the next stable interval. Thus, this algorithm is conservative and does not utilize all possible intervals, only those that it deemed stable for attack. Experimentally, we found this algorithm to be effective, as detailed below.

Quantifying the efficacy of the side channel. We conducted an experiment to quantify how much the side channel helps improve the stealthiness of the clickjack attack. On an Android device running *WhatTheApp* defenses, we deployed a Facebook App and a background malware app (unprivileged). We launched Facebook automatically and brought it to the login screen for 20 seconds, then exited Facebook, slept a random amount of time (3 to 5 seconds) and repeated the process (this was done automatically). The background app launched its attack when Facebook's login screen was up. It then overlayed an attack window fashioned like a keyboard for 400ms ten times during the 20sec interval. The aim was to realistically simulate user login and a malware's attempt to capture parts of the password during the login without getting detected by *WhatTheApp*'s defenses, which were executing all the time. This process was repeated for 6 hours to give us substantial amount of data for statistically valid results.

We compared two attack strategies (Table 2): (1) with no side channel information on the security checks; and (2) with exploiting the new side channel to predict the time till the next security check. The first strategy waited for Facebook to start and then began an attack sequence where a clickjacking window is shown for 400ms followed by a pause of 600ms where no window a shown,

ten times with each 20-sec duration when the Facebook's login window was up. Since the security check of *WhatTheApp* is 1 second apart, our random strategy aimed to randomly synchronize with the security checks so that it could elude the check for 10 consecutive clickjack events per login window display. We found that out of a total of 11199 attack events, 5724 attack events were caught and marked as yellow. This was generally consistent with our intuition in that if the first clickjacking event evaded the check (60% odds, since the attack window was 400ms in each second), others were likely to evade as well in that attack window.

The side-channel strategy used the binder transaction statistics as an additional signal to help time the 400ms long attacks. The process was same as the above in other respects, except that the 400ms attack was only launched when dictated by the side channel. We again ran the experiment over 6 hours with Facebook configured identically. There were a total of 10783 attacks out of which, only 860 were caught (i.e., security indicator's lock turned yellow)[3].

In summary, the random strategy, without side channel information, was found to be successful approximately 48.8% of the time compared to 92% of the time for the strategy using the binder log side-channel. Statistically, this is a significant increase in stealthiness. Our attack demonstrations are available here [13][4].

Overlay attacks on a system window. Based on the code of the *WhatTheApp* prototype (Listing 1.1, lines 3-4), we noticed that if the top activity was `SYSTEM`, the green lock icon stays on all the time. This was the case even if a malware window was layered on top of the system window. This enabled our test malware to freely display attack windows over sensitive system apps such as the default EMail client and WiFi password boxes and capture the input. We also tried this attack using random show/hide durations and the lock icon visualization always remained green. We believe this attack is possible due to an implementation flaw in *WhatTheApp*—the defense does not inspect the identities of all windows on the stack when the foreground window is of type `SYSTEM`.

5 Proposed Design

Addressing the problem we illustrated with the *WhatTheApp* UI defense turns out to be tricky with some tradeoffs. We thus describe two solutions to the problem, with the first solution a variant of Bianchi's solution to make it harder to evade check logic and the second solution introducing an additional mechanism to render window overlay attempts by background apps harmless. The second

[3] There are slightly fewer attacks than the random strategy since the channel strategy is conservative and chooses to let some intervals go without attack attempts rather than to risk detection.

[4] Our attack demonstrations in the video use a window type that captures input without passing the input to the underlying window. We subsequently verified that our attacks are feasible using toast windows, that allow input to be both captured and passed to the underlying window; in this case the user will not notice missing characters.

solution is more robust from a security perspective, and thus more appropriate for high-assurance environments, but it also limits some aspects of the way Android apps are allowed to interact with users.

Before we describe the two designs, we note that this paper is not about the design of security indicators. We assume that both designs that we are presenting use security indicators in the style of *WhatTheApp* since end-users found them to be effective according to a systematic user study conducted by Bianchi *et al.* Instead, the goal of this paper is to provide systematic defenses against the newly discovered vulnerabilities that are discussed in Sections 3 and 4.

5.1 Design 1: Improving attack detection with existing UI defenses

An obvious first step towards defense is to plug the newly discovered side channel by preventing access to information in `/sys/kernel/debug/binder` to apps. However, other side channels cannot be ruled out. There are other shared resources (e.g., lock on the window manager) that could potentially leak timing data. A more robust strategy could be to randomize the security check time intervals in *WhatTheApp*. Unfortunately, even that would not rule out successful attacks. Some characters could still be stolen via clickjacking as intervals where no checks take place would still exist.

Furthermore, transient windows like Toasts or Chat heads or Now-on-Tap-style widgets occur in Android normally. Those would turn the lock yellow, perhaps desensitizing a user to occasional occurrences of a yellow lock.

WhatTheApp has a potential implementation error where `SYSTEM`-owned windows can be pre-empted without a change in indicator visualization because the code assumes the system is safe if the top level activity is system-owned. A simple fix is to use the same logic for system windows that is used for other apps, namely, checking that all activites in the stack are from the `SYSTEM` when the top-level activity is `SYSTEM`. We confirmed that this fix helps, however, clickjack attacks remain feasible.

Thus, overall, interceptions of user input by malware via clickjacking becomes harder, but remains feasible, despite the additional defenses. The fundamental problem is that short-duration clickjacking attacks can go undetected if they fall between security checks.

5.2 Design 2: Secure Entry Mode using an Overlay Mutex

Design 2 blocks clickjacking attacks while the soft keyboard is being used. In the current proof-of-concept prototype, we focused on deploying the defenses of design 2 whenever the soft keyboard is used since that is a typical method for entering passwords and sensitive input, leaving generalizations to other inputs as future work. Note that our overlay mutex algorithms, however, are independent of the specific mechanism used for activation. As in *WhatTheApp*, design 2 requires the user to verify the green lock and app name in the security bar once, after a soft keyboard comes up, but before entering any sensitive text input (e.g., a password). Unlike *WhatTheApp*, Design 2 guarantees that any keyboard input

entered goes to the identified app only, even if background malware attempts to perform clickjack attacks to intercept input secretly by exploiting the binder side channel.

Overlay Mutex. To provide the guarantee, design 2 uses a novel security mechanism that we term *overlay mutex* designed as an addition to Android's UI stack. Overlay mutex utilizes an inter-process synchronization lock, as opposed to the visual green/yellow lock in Bianchi *et al.* The overlay mutex is acquired and released on behalf the foreground app, whenever the soft keyboard is shown and hidden respectively. It provides the following invariant during the period that an overlay mutex is held:

A background non-system app cannot overlay a window on top of the foreground app's window(s). Instead, an attempt to do so is converted to a safe user notification.

On Android, the invariant has a significant implication with respect to the ability of a background app to surreptitiously become the foreground app and hijack input (see the Activity Hijacking attack discussion in Section 2). The net result of the above is that a background app cannot tamper with the screen in arbitrary ways, cannot become the foreground app unless the user explicitly chooses to switch, and it cannot therefore intercept the input intended for the foreground app. We elaborate on that below.

Background apps have three options to pre-empt foreground apps. Toasts are customizable transient windows and Activities are full screen windows that can be created by malicious apps. Also, arbitrary sized windows can be created and added to the window hierarchy by directly invoking `WindowManager.addView`.

The overlay mutex mechanism prevents all pre-emption attack vectors. When the overlay mutex is active, a background app's request to display a window is blocked. The active overlay mutex converts the pre-emption window into a system-generated, fixed-size textual notification with the interrupting app's identity. Our system displays the notification on the Android status bar at the top of the screen. Since malware has no control over the size and placement of these notifications, the ability of the background app to involuntarily cause a switch or overlay a window in unexpected ways is taken away. Furthermore, the precise timing of the switch is no longer under the control of the attacker. Upon seeing a notification, users can voluntarily resume the pre-empting window by tapping on the notification, after they have inspected the identity information—interrupts are not lost, merely delayed. We do not think this is particularly limiting for practical use in high assurance apps since other cues exist. Audio and haptic alerts can serve to draw the user's attention to the notification.

Overlay Mutex Algorithm. We discuss the logic of the Overlay Mutex in the case of the Android operating system. The Overlay Mutex is two functions in the `ActivityManager`—`EnterSecureMode` and `ExitSecureMode`. Whenever policy dictates that secure input should be available (such as when a keyboard is displayed), the `EnterSecureMode` function executes (Listing 1.6). First, `EnterSecureMode` attempts to gain a synchronization lock maintained centrally by the activity management service. The second step is to verify the identity of

the foreground app and update the visualization of the security indicator appropriately. The third step is to store a state variable `CurrentVerifApp` that is used for checks during pre-emption attempts and finally release the activity manager lock. `ExitSecureMode` simply resets the value of `CurrentVerifApp`.

If secure mode is disabled (`CurrentVerifApp` is null), windows can show up at any point in time and the security indicator updates itself once a second to reflect the current state of the display—no change from *WhatTheApp*.

Consider the case where secure mode is enabled (`CurrentVerifApp` is set to an app's package name) and a background process tries to display an activity, pre-empting the user's current secure mode. If the background process is the system, as a matter of policy, we let this pre-emption attempt succeed and update our `CurrentVerifApp` state variable appropriately. The security indicator updates itself in the usual way.

However, if the background service does not have the same package name as the current verified app, then we hold that pre-emption attempt and instead show a safe notification to the user (Lines 9-10, Listing 1.4). This provides the Overlay Mutex security guarantee that windows of unrelated provenance will not appear on the display.

Listing 1.4. StartActivity

```
1  void ActivityMgr.startActivity() {
2    ActivityMgr.lock()
3    if(CurrentVerifApp == NULL or
4    callerUID == SYSTEM or
5    callerUID == CurrentVerifApp.UID) {
6      newAct = showActivity()
7      updateTopStackActivity(newAct)
8    } else {
9      ConvertActivityToNotification()
10     ShowSafeNotification()
11   }
12   ActivityMgr.unlock()
13 }
```

Listing 1.5. Adding a window to the display

```
1  status_code WindowMgr.AddView() {
2    retval = ERROR
3    ActivityMgr.lock() //IPC
4    if(CurrentVerifApp == NULL or
5    callerUID == SYSTEM or
6    callerUID == CurrentVerifApp.UID) {
7      showWindow()
8      retval = SUCCESS
9    }
10   ActivityMgr.unlock() //IPC
11   return retval
12 }
```

Listing 1.6. Entering and exiting secure mode

```
1  void ActivityMgr.lock() {
2    lock(ActivityLock)
3  }
4
5  void ActivityMgr.unlock() {
6    unlock(ActivityLock)
7  }
8
9  void ActivityMgr.EnterSecureMode() {
10   ActivityMgr.lock()
11   CurrentVerifApp = getTopActivityId('EV')
12   updateSecurityBar(CurrentVerifApp)
13   ActivityMgr.unlock()
14 }
15
16 void ActivityMgr.ExitSecureMode() {
17   ActivityMgr.lock()
18   CurrentVerifApp = NULL
19   ActivityMgr.unlock()
20 }
```

Implementation Details. We prototyped the Overlay Mutex algorithms on top of the released source code of *WhatTheApp*. We modified the `ActivityManagerService` to create a synchronization lock and modified functions related to creating windows on the screen—`startActivity`, `Toast.show`, and `addView`. A noteworthy point with `addView` is that it directly adds windows

to the screen without an associated activity. If an `addView` call fails with error (like in Listing 1.5), Android will not try again. We modified the Android in-process helper library to retry adding a window upon receipt of an error from the `WindowManager`. This ensures that `addView` will succeed at a later point once the user exits secure mode.

6 Defense Evaluation

We analyzed existing UI attacks found in the wild. These attacks are representative of the general set of techniques attackers use to launch UI attacks and cover all classes discussed in Section 2. We tested and compared Design 2 with *WhatTheApp* against these attacks. Design 1 has similar limitations as *WhatTheApp*, since it does not eliminate clickjacking attacks; it only makes them less likely. We summarize the findings in Table 3 and discuss below.

Direct Phishing. Neither defense can prevent direct phishing where the top activity itself engages in phishing. However, both will provide a security indicator to alert users to the EV-certified identity of the app (if available) or indicate a red lock. The behavior of the two solutions is identical. The user is expected to verify the identity of the app and the lock status before entering input. We tested this with the following attack on the Google Wallet SDK (AliPay SDK). Alipay allows apps to fire an unrestricted intent to the Play Store to request its use. Unfortunately, any app can intercept this intent with a high priority receiver and instead become the foreground activity, as opposed to the Google PlayStore, and solicit sensitive banking information. This is similar to Felt's intent hijacking example of direct phishing [8]. For defense, the users will need to verify the security indicator in the security bar and realize that the activity soliciting the information is not Google Play Store.

Activity Hijacking. The behavior of the Overlay Mutex with this attack depends on the policy in use. In the most popular keyboard policy that we discussed, if the hijacking is attempted while the user is performing input, then the hijacking attempt is converted to a safe notification. We verified this with Trojan-Banker.AndroidOS.Svpeng.A [21]. However, if the hijacking attempt occurs during a time where the user does not have the keyboard up then, just like *WhatTheApp*, the security indicator turns yellow and the attack becomes user-detectable provided the user checks the indicator before a subsequent input attempt. At this point, the user must not enter any input and should try again, as is the case with *WhatTheApp*.

Clickjacking. Whenever the user is performing input using the soft keyboard, the overlay mutex is active and catches pre-emption attempts converting them to safe notifications thus preventing clickjacking. In contrast, *WhatTheApp* allows the pre-emption and updates the security bar visualization to a yellow unlocked icon. Since the transition of the attack window is very fast, chances are high that a user misses the change in the security indicator. Therefore, we mark *WhatTheApp* in Table 3 as unreliably detected. In the case of the clickjacking

Table 3. Summary of findings from UI deception attacks tested on Android v4.4 using our Overlay Mutex and *WhatTheApp*.

Malware behavior	**Overlay Mutex**	**Bianchi *et al.*** (verified on prototype)
Direct phishing of another app's data	User-detectable	User-detectable
Activity hijacking(e.g., Trojan-Banker.AndroidOS.Svpeng.A)	User-detectable	User-detectable
Clickjacking without side channel	Prevented	Unreliably detected
Clickjacking with side channel info	Prevented	Not detected or only rarely detected
Malware overlaying active System app while user performing input	Prevented	Not detected

attack using our newly discovered side channel, as shown earlier (Section 4), the security indicator does not change to yellow 92% of the time.

Malware overlaying system app during user input. In a similar vein, if an overlay mutex is active during user input into a system app, the pre-emption is converted to a safe notification. However, *WhatTheApp* does not recognize any overlays on top of system apps and the security indicator is green. Therefore, *WhatTheApp* does not detect such attacks. A fix that we made to *WhatTheApp* in this special case is to check the window stack just like it does for non-system apps. If this attack were applied to the patched-*WhatTheApp*, then the security indicator would display yellow and the attack will be user-detectable.

Microbenchmark Performance. Our overlay mutex solution changes window creation to make two IPC calls and one lock/unlock sequence. We quantified the overhead of these extra operations to determine the impact on interactivity using a UI bench that created and removed a 200x200 pixel window, 300 times. We ran our benchmark on a Nexus 4 running stock Android 4.4 and Android 4.4 with the overlay mutex's secure mode active. We observed that, on average, it took 17.7 microsec for a window to display on stock Android compared to 35.3 microsec on Android with our defense—a modest increase in window display time. For comparison, launching an Android activity takes several hundred milliseconds.

App Compatibility. We manually tested popular apps by exercising common functionality associated with each app on our overlay mutex prototype, and observed whether any functionality failed. We used the following set and did not observe any issues: standard (Messaging, Browser, Email, Search, Contacts), social networking (Facebook, Twitter), communication and business (Skype, TaxACT), banking (Chase, Bank of America), and top 2 free games (Angry Birds, Trivia Crack). For instance, we brought up the keyboard in Facebook (thereby gaining the Overlay Mutex), and initiated a Skype call from another device. The overlay mutex caught the Skype-initiated pre-emption for an incoming call, and later, when we clicked on the notification, the incoming call window pre-empted Facebook. The caller side experienced a delayed call answer.

7 Discussion

The attacks in this paper concerned stealing key input, such as passwords, credit card numbers, etc. However, clickjacking has a broader meaning—an attacker could display malicious windows strategically placed over UI elements to confuse the user. For example, a window placed over a permission listing screen can serve

misinformation to the user about the permissions a particular app requests, or windows placed over "OK" and "Cancel" buttons can reverse their meaning. Such attacks are possible in limited situations on *WhatTheApp*. In the case of overlaying a system window, such attacks will go unnoticed. In the general case of overlaying any window, a flickering effect will occur since the attacker has to add and remove the attack window so that the security check does not turn yellow. In the case of our overlay mutex, such attacks will be blocked, and converted to notifications.

The iOS platform too has been a victim of recent UI deception attacks [34,6]. These attacks exploit the lack of UI provenance. For instance, the common iCloud popup password boxes are indistinguishable from fake ones. The recent Android Marshmallow update does not offer any UI provenance or overlay mutex mechanisms—UI deception attacks are still possible.

A limitation of our work is the lack of a usability study to assess effectiveness of the defenses in practice and to determine whether users require training to make effective use of the defenses. Another limitation is that the overlay mutex mechanism could cause side-effects on functionality of existing apps or certain use cases. While we did test a small set of apps manually, as future work, we plan systematic testing of the mechanism with a larger set of apps [19].

8 Related Work

UI deception, or phishing, has a long history of attacks and defenses, initially occurring in the context of browsers [15,26,14,32,31]. The fundamental issue that attackers exploit is lack of provenance. Even when provenance exists, practice has shown that users' lack of understanding of security indicators, visually similar UIs, and lack of user attention conspire to make phishing attacks very successful [15]. UI deception attacks also use other attack vectors as enhancements, e.g., phishing emails are a common way to get users to visit phishing pages [18].

A large body of previous work has focused on various forms of UI attacks on smartphone operating systems. Niemietz *et al.* introduced UI redressing attacks [25], Felt *et al.* discuss phishing on mobile devices [16] and Chen *et al.* [11] introduce a memory-statistics side channel that is used to enhance classic activity hijacking attacks [3], [4], [12]. `svpeng` demonstrates that UI deception occurs in the wild and uses the activity hijacking technique [21]. All these serve to motivate the necessity for a technical solution to UI deception.

Tong and Evans introduced GuarDroid, a trusted path for password entry that uses a trusted keyboard to encrypt sensitive information before delivering to apps and then automatically decrypts the data upon network communication [28]. However, endpoints of this system are still vulnerable to overlay attacks. Therefore, Bianchi *et al.* propose a two-layered defense comprising a market-level static analysis tool to catch particularly malicious patterns of UI deception. This tool is augmented with a runtime defense modeled after the well-known HTTPS lock icon and EV infrastructure [7]. We regard this work as the state-of-the-art and presented its detailed analysis in Section 3, along

with a clickjacking vulnerability and a newly discovered side channel that makes clickjacking very effective.

Recently, Android 5.0 introduced the screen-pinning feature that locks the user into a single app and prevents navigation away from that app unless the owner explicitly exits screen-pinning using an unlock code [27]. However, screen-pinning is fundamentally different from overlay mutexes as they are not intended as a UI deception defense mechanism. We verified that direct phishing attacks and clickjacking attacks are possible on screen-pinned apps.

Huang *et al.* proposed InContext, a suite of defenses targetting visual and temporal integrity of browser UI elements [20]. Their solution requires introducing a delay of 250ms or higher between the time the user issues an input command and the time the command is delivered to the browser UI element. Besides the potential impact on interactivity, subsequent work has demonstrated that defenses relying on time delays are still vulnerable to clickjacking [5]. Our work does not rely on delays and instead locks out window transitions while the user is performing secure input, converting attempted overlays to notifications.

9 Conclusion

Android is vulnerable to a wide range of UI attacks. The standard solution to handling UI attacks is to use security indicators to assist users in identifying the attacks and recent solutions have proposed security indicators inspired by HTTPS lock icons for Android [7]. We studied the security properties of a recent system by Bianchi *et al.* and determined that it remains vulnerable to side-channel-enhanced clickjacking attacks. Our work introduced the Binder statistics channel and demonstrated how to leverage it to elude security checks. We proposed the overlay mutex mechanism that guarantees temporal integrity of security indicators by converting window pre-emption attempts to safe notifications. We evaluated our defense against known UI deception as well as the new side channel attack introduced in this paper and found that the defense is effective against these attacks.

Acknowledgements. We thank the reviewers for their insightful feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1318722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Android UI Deception PoC Code. https://github.com/earlence/AndroidUIDeceptionRevisitedFC16, Accessed: Oct 2015
2. Apple XCodeGhost Attack. http://www.apple.com/cn/xcodeghost/#english, Accessed: Oct 2015
3. Activity hijacking pattern for Android. http://capec.mitre.org/data/definitions/501.html, Accessed: Oct 2015

4. Android Touch-Event Hijacking. https://blog.lookout.com/blog/2010/12/09/android-touch-event-hijacking/, Accessed: Oct 2015
5. Akhawe, D., He, W., Li, Z., Moazzezi, R., Song, D.: Clickjacking revisited: A perceptual view of ui security. In: Proceedings of the 8th USENIX Conference on Offensive Technologies. pp. 1–1. WOOT'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2671293.2671294
6. Ben Lovejoy: Beware authentication popups in iOS Mail: bug allows convincing-looking phishing attacks. http://9to5mac.com/2015/06/10/ios-mail-phishing-popup/, Accessed: Dec 2015
7. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the App is That? Deception and Countermeasures in the Android User Interface. In: Proceedings of the IEEE Symposium on Security and Privacy (SP). San Jose, CA (May 2015)
8. Castillo, C.: McAfee Labs. Phishing Attack replaces Banking app with malware. Published June 2013. http://blogs.mcafee.com/mcafee-labs/phishing-attack-replaces-android-banking-apps-with-malware, Accessed: Oct 2015
9. Chebyshev, V., Unuchek, R.: Mobile malware evolution in 2013. http://securelist.com/analysis/kaspersky-security-bulletin/58335/mobile-malware-evolution-2013/, Accessed: Oct 2015
10. Chen, J., Chen, H., Bauman, E., Lin, Z., Zang, B., Guan, H.: You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 657–690. USENIX Association, Washington, D.C. (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-jin
11. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In: Proceedings of the 23rd USENIX Security Symposium (2014)
12. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-application Communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. pp. 239–252. MobiSys '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1999995.2000018
13. Clickjacking SideChannel Demonstration videos. https://sites.google.com/site/clickjackingsidechannels/, Accessed: Oct 2015
14. Dhamija, R., Tygar, J.D.: The Battle Against Phishing: Dynamic Security Skins. In: Proceedings of the 2005 Symposium on Usable Privacy and Security. pp. 77–88. SOUPS '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1073001.1073009
15. Dhamija, R., Tygar, J.D., Hearst, M.: Why Phishing Works. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 581–590. CHI '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1124772.1124861
16. Felt, A.P., Wagner, D.: Phishing on Mobile Devices. In: In W2SP (2011)
17. Fernandes, E., Chen, Q., Essl, G., Halderman, J.A., Mao, Z.M., Prakash, A.: TIVOs: Trusted Visual I/O Paths for Android. Tech. Rep. Technical Report CSE-TR-586-14, CSE Department, University of Michigan, Ann Arbor (2014)
18. Fette, I., Sadeh, N., Tomasic, A.: Learning to detect phishing emails. In: Proceedings of the 16th International Conference on World Wide Web. pp. 649–656. WWW '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1242572.1242660

19. Hao, S., Liu, B., Nath, S., Halfond, W.G., Govindan, R.: Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services. pp. 204–217. MobiSys '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2594368.2594390
20. Huang, L.S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: Attacks and defenses. In: Proceedings of the 21st USENIX Conference on Security Symposium. pp. 22–22. Security'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2362793.2362815
21. Kaspersky: Svpeng android malware targets banking apps. http://www.kaspersky.com/about/news/virus/2014/Kaspersky-Lab-detects-mobile-Trojan-Svpeng-Financial-malware-with-ransomware-capabilities-now-targeting-US-users, Accessed: Oct 2015
22. Kelly, M.: Badlepricon: Bitcoin gets the mobile malware treatment in Google Play. https://blog.lookout.com/blog/2014/04/24/badlepricon-bitcoin/, Accessed: Oct 2015
23. Liu, B., Nath, S., Govindan, R., Liu, J.: DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps (2014)
24. Liu, D., Cuervo, E., Pistol, V., Scudellari, R., Cox, L.P.: ScreenPass: Secure Password Entry on Touchscreen Devices. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services. pp. 291–304. MobiSys '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2462456.2465425
25. Niemietz, M., Schwenk, J.: UI Redressing Attacks on Android Devices. In: Proceedings of BlackHat Abu Dhabi.) (2012)
26. Schechter, S.E., Dhamija, R., Ozment, A., Fischer, I.: The Emperor's New Security Indicators. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy. pp. 51–65. SP '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/SP.2007.35
27. Android 5.0 Screen Pinning. https://support.google.com/nexus/answer/6118421?hl=en, Accessed: Oct 2015
28. Tong, T., Evans, D.: GuarDroid: A Trusted Path for Password Entry. In: Proceedings of Mobile Security Technologies (MoST) (2013)
29. TrendMicro: Mobile phishing attacks ask for government ids. http://blog.trendmicro.com/trendlabs-security-intelligence/mobile-phishing-attack-asks-for-users-government-ids/, Accessed: Oct 2015
30. Unuchek, R.: Svpeng android malware targets google play with fake credit card window. http://securelist.com/blog/incidents/63746/latest-version-of-svpeng-targets-users-in-us/, Accessed: Oct 2015
31. Whittaker, C., Ryner, B., Nazif, M.: Large-scale automatic classification of phishing pages. In: NDSS (2010)
32. Wu, M., Miller, R.C., Garfinkel, S.L.: Do Security Toolbars Actually Prevent Phishing Attacks? In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 601–610. CHI '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1124772.1124863
33. Zhang, Y., Xue, H., Wei, T.: Occupy your icons silently on android. http://www.fireeye.com/blog/technical/2014/04/occupy_your_icons_silently_on_android.html, Accessed: Oct 2015
34. Zhaofeng Chen, Tao Wei, Hui Xue, Yulong Zhang: Three New Masque Attacks against iOS: Demolishing, Breaking and Hijacking. https://www.fireeye.com/blog/threat-research/2015/06/three_new_masqueatt.html, Accessed: Dec 2015