2015 IEEE Symposium on Security and Privacy

What the App is That? Deception and Countermeasures in the Android User Interface

Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
{*antoniob,jacopo,invernizzi,yanick,chris,vigna*}*@cs.ucsb.edu*

***Abstract*—Mobile applications are part of the everyday lives of billions of people, who often trust them with sensitive information. These users identify the currently focused app solely by its visual appearance, since the GUIs of the most popular mobile OSes do not show any trusted indication of the app origin.**

In this paper, we analyze in detail the many ways in which Android users can be confused into misidentifying an app, thus, for instance, being deceived into giving sensitive information to a malicious app. Our analysis of the Android platform APIs, assisted by an automated state-exploration tool, led us to identify and categorize a variety of attack vectors (some previously known, others novel, such as a non-escapable fullscreen overlay) that allow a malicious app to surreptitiously replace or mimic the GUI of other apps and mount phishing and click-jacking attacks. Limitations in the system GUI make these attacks significantly harder to notice than on a desktop machine, leaving users completely defenseless against them.

To mitigate GUI attacks, we have developed a two-layer defense. To detect malicious apps at the market level, we developed a tool that uses static analysis to identify code that could launch GUI confusion attacks. We show how this tool detects apps that might launch GUI attacks, such as ransomware programs. Since these attacks are meant to confuse humans, we have also designed and implemented an on-device defense that addresses the underlying issue of the lack of a security indicator in the Android GUI. We add such an indicator to the system navigation bar; this indicator securely informs users about the origin of the app with which they are interacting (e.g., the PayPal app is backed by "PayPal, Inc.").

We demonstrate the effectiveness of our attacks and the proposed on-device defense with a user study involving 308 human subjects, whose ability to detect the attacks increased significantly when using a system equipped with our defense.

I. Introduction

Today, smartphone and tablet usage is on the rise, becoming the primary way of accessing digital media in the US [1]. Many users now trust their mobile devices to perform tasks, such as mobile banking or shopping, through mobile applications, typically called "apps." This wealth of confidential data has not gone unnoticed by cybercriminals: over the last few years, mobile malware has grown at an alarming rate [2].

Popular mobile operating systems run multiple apps concurrently. For example, a user can run both her mobile banking application and a new game she is checking out. Obviously, a game should not receive financial information. As a consequence, the ability to tell the two apps apart is crucial. At the same time, it is important for these apps to have user-friendly interfaces that make the most of the limited space and interaction possibilities.

Let us assume that a victim user is playing the game, which is malicious. When this user switches to another app, the game will remain active in the background (to support background processing and event notifications). However, it will also silently wait for the user to login into her bank. When the malicious game detects that the user activates the banking app, it changes its own appearance to mimic the bank's user interface and instantly "steals the focus" to become the target with which the victim interacts. The user is oblivious to this switch of apps in the foreground, because she recognizes the graphical user interface (GUI) of the banking application. In fact, there have been no changes on the user's display throughout the attack at all, so it is impossible for her to detect it: she will then insert her personal banking credentials, which will then be collected by the author of the malicious app.

In this paper, we study this and a variety of other *GUI confusion* attacks. With this term, we denote attacks that exploit the user's inability to verify which app is, at any moment, drawing on the screen and receiving user inputs. GUI confusion attacks are similar to social engineering attacks such as phishing and click-jacking. As such, they are not fundamentally novel. However, we find that the combination of powerful app APIs and a limited user interface make these attacks much harder to detect on Android devices than their "cousins" launched on desktop machines, typically against web browsers.

The importance of GUI-related attacks on Android has been pointed out by several publications in the past, such as [3], [4] (with a focus on "tapjacking"), [5] (with a focus on phishing attacks deriving from control transfers), and [6] (with a focus on state disclosure through shared-memory counters). Our paper generalizes these previously-discovered techniques by systematizing existing exploits. Furthermore, we introduce a number of novel attacks. As an extreme example of a novel attack, we found that a malicious app has the ability to create a complete virtual environment that acts as a full Android interface, with complete control of all user interactions and inputs. This makes it very hard for a victim user to escape the grip of such a malicious application. Even though at the time of this writing the number of known samples performing GUI confusion attacks is limited, we believe (as we will show in this paper) that this is a real, currently unsolved, problem in the Android ecosystem.

This paper also introduces two novel approaches to defend

DOI 10.1109/SP.2015.62

against GUI confusion attacks. The first approach leverages static code analysis to automatically find apps that could abuse Android APIs for GUI confusion attacks. We envision that this defense could be deployed at the market level, identifying suspicious apps before they hit the users. Interestingly, we detected that many benign apps are using potentially-dangerous APIs, thus ruling out simple API modifications as a defense mechanism.

Our static analysis approach is effective in identifying potentially-malicious apps. More precisely, our technique detects apps that interfere with the UI in response to some action taken by the user (or another app). The apps that we detect in this fashion fulfill two necessary preconditions of GUI confusion attacks: They monitor the user and other apps, and they interfere with the UI (e.g., by stealing the focus and occupying the top position on the screen). However, these two conditions are not sufficient for GUI confusion attacks. It is possible that legitimate apps monitor other apps and interfere with the UI. As an example, consider an "app-locker" program, which restricts access to certain parts of the phone (and other apps). When looking at the code, both types of programs (that is, malicious apps that launch GUI confusion attacks as well as app-lockers) look very similar and make use of the same Android APIs. The difference is in the intention of the apps, as well as the content they display to users. Malicious apps will attempt to mimic legitimate programs to entice the user to enter sensitive data. App-lockers, on the other hand, will display a screen that allows a user to enter a PIN or a password to unlock the phone. These semantic differences are a fundamental limitation for detection approaches that are purely code-based.

To address the limitations of code-based detection, we devised a second, on-device defense. This approach relies on modifications to the Android UI to display a trusted indicator that allows users to determine which app and developer they are interacting with, attempting to reuse security habits and training users might already have. To this end, we designed a solution (exemplified in Figure 1) that follows two well-accepted paradigms in web security:

- the Extended Validation SSL/TLS certification and visualization (the current-best-practice solution used by critical businesses to be safely identified by their users)
- the use of a "secure-image" to established a shared secret between the user interface and the user (similarly to what is currently used in different websites [7], [8] and recently proposed for the Android keyboard [9])

We evaluate the effectiveness of our solution with a user study involving 308 human subjects. We provided users with a system that implements several of our proposed defense modifications, and verified that the success ratio of the (normally invisible) deception attacks significantly decreases.

To summarize, the main contributions of this paper are:

- We systematically study and categorize the different techniques an attacker can use to mount GUI deception attacks. We describe several new attack vectors that we found, and we introduce a tool to automatically explore reachable GUI states and identify the ones that can be used to mount an attack. This tool was able to automatically find two vulnerabilities in the Android framework that allow an app to gain full control of a device's UI.
- We study, using static analysis, how benign apps legitimately use API calls that render these attacks possible. Then, we develop a detection tool that can identify their malicious usage, so that suspicious apps can be detected at the market level.

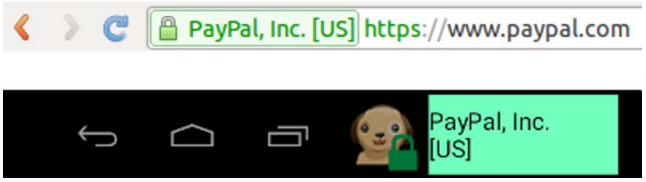


Fig. 1: Comparison between how SSL Extended Validation information is shown in a modern Browser (Chrome 33) and what our implemented defense mechanism shows on the navigation bar of an Android device.

- We propose an on-device defense that allows users to securely identify authors of the apps with which they interact. We compare our solution with the current state of the art, and we show that our solution has the highest coverage of possible attacks.
- In a user study with 308 subjects, we evaluate the effectiveness of these attack techniques, and show that our on-device defense helps users in identifying attacks.

For the source code of the proof-of-concept attacks we developed and the prototype of the proposed on-device defense, refer to our repository[1].

II. Background

To understand the attack and defense possibilities in the Android platform, it is necessary to introduce a few concepts and terms.

The Android platform is based on the Linux operating system and it has been designed mainly for touchscreen mobile devices. Unless otherwise noted, in this paper we will mainly focus on Android version 4.4. When relevant, we will also explain new features and differences introduced by Android 5.0 (the latest available version at the time of writing).

In an Android device, apps are normally pre-installed or downloaded from the Google Play Store or from another manufacturer-managed market, although manual offline installation and unofficial markets can also be used. Typically, each app runs isolated from others except for well-defined communication channels.

Every app is contained in an *apk* file. The content of this file is signed to guarantee that the app has not been tampered with and that it is coming from the developer that owns the corresponding private key. There is no central authority, however, to ensure that the information contained in the developer's signing certificate is indeed accurate. Once installed on a device, an app is identified by its *package name*. It is not possible to install apps with the same package name at the same time on a single device.

Apps are composed of different developer-defined components. Specifically, four types of components exist in Android: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. An *Activity* defines a graphical user interface and its interactions with user's actions. Differently, a *Service* is a component running in background, performing long-running operations. A *Broadcast Receiver* is a component that responds to specific system-wide messages. Finally, a *Content Provider* is used to manage data shared with other components (either within the same app or with external ones).

[1] https://github.com/ucsb-seclab/android_ui_deception

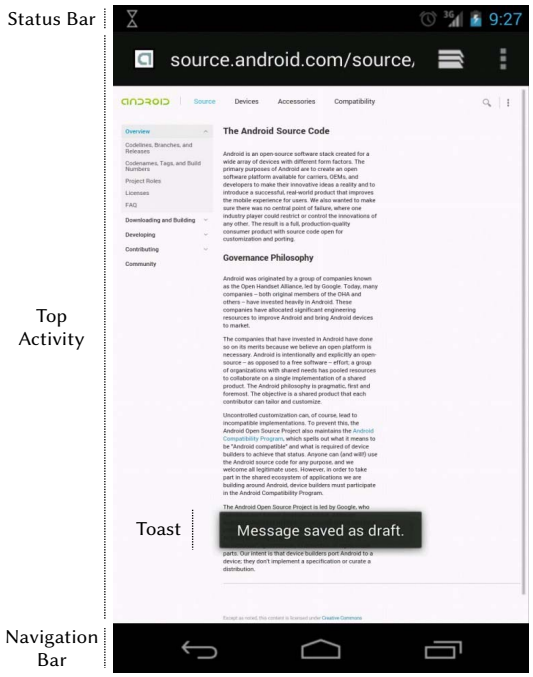


Fig. 2: Typical Android user interface appearance. The status bar is at the top of the screen, while the navigation bar occupies the bottom. A browser app is open, and its main Activity is shown in the remaining space.

To perform sensitive operations (e.g., tasks that can cost money or access private user data), apps need specific permissions. All the permissions requested by a *non-system* app must be approved by the user during the app's installation: a user can either grant all requested permissions or abort the installation. Some operations require permissions that are only granted to *system* apps (typically pre-installed or manufacturer-signed). Required permissions, together with other properties (such as the package name and the list of the app's components), are defined in a manifest file (`AndroidManifest.xml`), stored in the app's apk file.

A. Android graphical elements

Figure 2 shows the typical appearance of the Android user interface on a smartphone. The small *status bar*, at the top, shows information about the device's state, such as the current network connectivity status or the battery level. At the bottom, the *navigation bar* shows three big buttons that allow the user to "navigate" among all currently running apps as well as within the focused app.

Details may vary depending on the manufacturer (some devices merge the status and navigation bars, for instance, and legacy devices may use hardware buttons for the navigation bar). In this work we will use as reference the current guidelines[2], as they represent a typical modern implementation; in general, our considerations can be adapted to any Android device with minor modifications.

[2]http://developer.android.com/design/handhelds/index.html , http://developer.android.com/design/patterns/compatibility.html

Apps draw graphical elements by instantiating system-provided components: Views, Windows, and Activities.

Views. A *View* is the basic UI building block in Android. Buttons, text fields, images, and OpenGL viewports are all examples of views. A collection of Views is itself a View, enabling hierarchical layouts.

Activities. An *Activity* can be described as a controller in a Model-View-Controller pattern. An Activity is usually associated with a View (for the graphical layout) and defines actions that happen when the View elements are activated (e.g., a button gets clicked).

Activities are organized in a global stack that is managed by the *ActivityManager* system Service. The Activity on top of the stack is shown to the user. We will call this the *top Activity* and the app controlling it the *top app*.

Activities are added and removed from the Activity stack in many situations. Each app can reorder the ones it owns, but separate permissions are required for global monitoring or manipulation. Users can request an Activity switch using the navigation bar buttons:

- The *Back* button (bottom left in Figure 2) removes the top Activity from the top of the stack, so that the one below is displayed. This default behavior can be overridden by the top Activity.
- The *Home* button lets the user return to the base "home" screen, usually managed by a system app. A normal app can only replace the home screen if the user specifically allows this.
- The *Recent* button (bottom right in Figure 2) shows the list of top Activities of the running apps, so the user can switch among them. Activities have the option not to be listed. In Android 5.0, applications can also decide to show different thumbnails on the Recent menu (for instance, a browser can show a different thumbnail in the Recent menu for each opened tab).

Windows. A *Window* is a lower-level concept: a virtual surface where graphical content is drawn as defined by the contained Views. In Figure 2, the Status Bar, the Navigation Bar and the top Activity are all drawn in separate Windows. Normally, apps do not explicitly create Windows; they just define and open Activities (which in turn define Views), and the content of the top Activity is drawn in the system-managed *top-activity Window*. Windows are normally managed automatically by the *WindowManager* system Service, although apps can also explicitly create Windows, as we will show later.

III. GUI CONFUSION ATTACKS

In this section, we discuss classes of GUI confusion attacks that allow for launching stealthy and effective phishing-style or click-jacking-style operations.

In our threat model, a malicious app is running on the victim's Android device, and it can only use APIs that are available to any benign non-system app. We will indicate when attacks require particular permissions. We also assume that the base Android operating system is not compromised, forming a Trusted Computing Base.

We have identified several Android functionalities (*Attack Vectors*, categorized in Table I) that a malicious app can use to mount GUI confusion attacks. We have also identified *Enhancing Techniques*: abilities (such as monitoring other apps) that do not present a GUI security risk in themselves, but can assist in making attacks more convincing or stealthier.

TABLE I: Attack vectors and enhancing techniques. We indicate with a dash attacks and techniques that, to the best of our knowledge, have not been already mentioned as useful in GUI confusion attacks.

Category	Attack vector	Mentioned in
Draw on top	UI-intercepting draw-over	[3], [5]
	Non-UI-intercepting draw-over	[3], [4], [5]
	Toast message	[3], [10]
App switch	*startActivity* API	[6]
	Screen pinning	—
	moveTaskTo APIs	—
	killBackgroundProcesses API	—
	Back / power button (passive)	—
	Sit and wait (passive)	—
Fullscreen	non-"immersive" fullscreen	—
	"immersive" fullscreen	—
	"inescapable" fullscreen	—
Enhancing techniques	*getRunningTask* API	[5]
	Reading the system log	[11]
	Accessing proc file system	[6], [12]
	App repackaging	[13], [14], [15]

A. Attack vectors

1) Draw on top: Attacks in this category aim to draw graphical elements over other apps. Typically, this is done by adding graphical elements in a Window placed over the top Activity. The Activity itself is not replaced, but malware can cover it either completely or partially and change the interpretation the user will give to certain elements.

Apps can explicitly open new Windows and draw content in them using the *addView* API exposed by the WindowManager Service. This API accepts several flags that determine how the new Window is shown (for a complete description, refer to the original documentation[3]). In particular, flags influence three different aspects of a Window:

- Whether it is intercepting user input or is letting it "pass through" to underlying Windows.
- Its *type*, which determines the Window's Z-order with respect to others.
- The region of the screen where it is drawn.

Non-system apps cannot open Windows of some types, while Windows with a higher Z-order than the top-activity Window require the *SYSTEM_ALERT_WINDOW* permission.

Windows used to display *toasts*, text messages shown for a limited amount of time, are an interesting exception. Intended to show small text messages even when unrelated apps control the main visualization, toast messages are usually created with specific APIs and placed by the system in Windows of type *TOAST*, drawn over the top-activity Window. No specific permission is necessary to show toast messages. Their malicious usage has been presented by previous research (refer to Table I).

Two other types of attack are possible:

- *UI-intercepting draw-over:* A Window spawned using, for instance, the *PRIORITY_PHONE* flag can not only overlay the top-activity Window with arbitrary content, but also directly steal information by intercepting user input.
- *Non UI-intercepting draw-over:* By forwarding all user input to the underlying Windows, classical "click-jacking" attacks are possible. In these attacks, users are lured to perform an unwanted action while thinking they are interacting with a different element.

2) App switch: Attacks that belong to this category aim to steal focus from the top app. This is achieved when the malicious app seizes the top Activity: that is, the malicious app replaces the legitimate top Activity with one of its own. The malicious app that we developed for our user study (Section VII) uses an attack in this category: it waits until the genuine Facebook app is the top app, and then triggers an app switch and changes its appearance to mimic the GUI of the original Facebook app.

Replacing the currently running app requires an *active* app switch. *Passive* app switches are also possible: in this case, the malicious application does not actively change the Activity stack, nor it shows new Windows, but it waits for specific user's input.

We have identified several attack vectors in this category:

***startActivity* API.** New Activities are opened using the *startActivity* API. Normally, the newly opened Activity does not appear on top of Activities of other apps. However, under particular conditions the spawned Activity will be drawn on top of all the existing ones (even if belonging to different apps) without requiring any permission. Three different aspects determine this behavior: the type of the Android component from which the *startActivity* API is called, the *launchMode* attribute of the opened Activity, and flags set when *startActivity* is called.

Given the thousands of different combinations influencing this behavior and the fact that the official documentation[4] does not state clearly when a newly Activity will be placed on top of other apps' Activities, we decided to develop a tool to systematically explore the conditions under which this happens.

Our tool determined that opening an Activity from a Service, a Broadcast Receiver, or a Content Provider will always place it on top of all the others, as long as the *NEW_TASK* flag is specified when the *startActivity* API is called. Alternatively, opening an Activity from another one will place the opened Activity on top of all the others if the *singleInstance* launch mode is specified. In addition, our tool found other, less common, situations in which an Activity is placed on top of all the others. For more details and a description of our tool, refer to Section IV-A.

***moveTaskTo* APIs.** Any app with the *REORDER_TASKS* permission can use the *moveTaskToFront* API to place Activities on top of the stack. We also found another API, *moveTaskToBack*, requiring the same permission, to remove another app from the top of the Activity stack.

Screen pinning. Android 5.0 introduces a new feature called "screen pinning" that locks the user interaction to a specific app. Specifically, while the screen is "pinned," there cannot be any switch

[3]http://developer.android.com/reference/android/view/WindowManager.LayoutParams.html

[4]http://developer.android.com/guide/components/tasks-and-back-stack.html

to a different application (the Home button, the Recent button, and the status bar are hidden). Screen pinning can be either manually enabled by a user or programmatically requested by an app. In the latter case, user confirmation is necessary, unless the app is registered as a "device admin" (which, again, requires specific user confirmation).

***killBackgroundProcesses* API.** This API (requiring the *KILL_BACKGROUND_PROCESSES* permission) allows killing the processes spawned by another app. It can be used maliciously to interfere with how benign apps work: besides mimicking their interface, a malicious app could also prevent them from interacting with the user. Android does not allow killing the app controlling the top Activity, but other attack vectors can be used to first remove it from the top of the stack.

Back/Power Button. A malicious app can also make the user believe that an app switch has happened when, in fact, it has not. For example, an app can intercept the actions associated with the *back* button. When the user presses the back button, she expects one of two things: either the current app terminates, or the previous Activity on the stack is shown. A malicious app could change its GUI to mimic its target (such as a login page) in response to the user pressing the back button, while at the same time disabling the normal functionality of the back button. This might make the user believe that an app switch has occurred, when, in fact, she is still interacting with the malicious app. A similar attack can be mounted when the user turns off the screen while the malicious app is the top app.

Sit and Wait. When a malicious app is in the background, it can change its GUI to that of a victim app, so that when the user switches between apps looking, for example, for the legitimate banking application, she could inadvertently switch to the malicious version instead. This type of attack is known in the browser world as *tabnabbing* [16].

3) Fullscreen: Android apps have the possibility to enter the so called *fullscreen mode*, through which they can draw on the device's entire screen area, including the area where the navigation bar is usually drawn. Without proper mitigations, this ability could be exploited by malicious apps, for example, to create a fake home screen including a fake status bar and a fake navigation bar. The malicious app would therefore give the user the impression she is interacting with the OS, whereas her inputs are still intercepted by the malicious app.

Android implements specific mitigations against this threat [17]: An app can draw an Activity on the entire screen, but in principle users always have an easy way to close it and switch to another app. Specifically, in Android versions up to 4.3, the navigation bar appears on top of a fullscreen Activity as soon as the user clicks on the device screen. Android 4.4 introduces a new "immersive" fullscreen mode in which an Activity remains in fullscreen mode during all interactions: in this case, the navigation bar is accessed by performing a specific "swipe" gesture.

Given the large number of possible combinations of flags that apps are allowed to use to determine the appearance of a Window in Android, these safety functionalities are intrinsically difficult to implement. In fact, the implementation of the Android APIs in charge of the creation and display of Windows has thousands of lines of code, and bugs in this APIs are likely to enable GUI confusion attacks. Therefore, we used our API exploration tool to check if it is possible to create a Window that covers the entire device's screen area (including the navigation bar) without giving any possibility to the user to close it or to switch to another application. We call a Window with these properties an *"inescapable"* fullscreen Window.

Our tool works by spawning Windows with varying input values of GUI-related APIs and, after each invocation, determines whether an "inescapable" fullscreen mode is entered. By using it, several such combinations were found, thus leading to the discovery of vulnerabilities in different Android versions. Upon manual investigation, we found that Google committed a patch[5] to fix a bug present in Android 4.3; however, our tool pointed out that this fix does not cover all possible cases. In fact, we found a similar problem that affects Android versions 4.4 and 5.0. We notified Google's Security Team: a review is in progress at the time of this writing.

Section IV-B presents more technical details about the tool we developed and its findings.

There is effectively no limit to what a malicious programmer can achieve using an "inescapable" fullscreen app. For instance, one can create a full "fake" environment that retains full control (and observation powers) while giving the illusion of interacting with a regular device (either by "proxying" app Windows or by relaying the entire I/O to and from a separate physical device).

B. Enhancing techniques

Additional techniques can be used in conjunction with the aforementioned attack vectors to mount more effective attacks.

1) Techniques to detect how the user is currently interacting with the system: To use the described attack vectors more effectively, it is useful for an attacker to know how the user is currently interacting with the device.

For instance, suppose again that a malicious app wants to steal bank account credentials. The most effective way would be to wait until the user actually opens the specific login Activity in the original app and, immediately after, cover it with a fake one. To do so, it is necessary to know which Activity and which app the user is currently interacting with.

We have identified a number of ways to do so: some of them have been disabled in newer Android versions, but others can still be used in the latest available Android version.

Reading the system log. Android implements a system log where standard apps, as well as system Services, write logging and debugging information. This log is readable by any app having the relatively-common *READ_LOGS* permission (see Table IV in the next section). By reading messages written by the ActivityManager Service, an app can learn about the last Activity that has been drawn on the screen.

Moreover, apps can write arbitrary messages into the system log and this is a common channel used by developers to receive debug information. We have observed that this message logging is very commonly left enabled even when apps are released to the public, and this may help attackers time their actions, better reproduce the status of an app, or even directly gather sensitive information if debug messages contain confidential data items.

Given the possible malicious usage of this functionality, an app can only read log messages created by itself in Android version 4.1 and above.

[5] https://android.googlesource.com/platform/frameworks/base/+/b816bed

***getRunningTasks* API.** An app can get information about currently running apps by invoking the *getRunningTasks* API. In particular, it is possible to know which app is on top and the name of the top Activity. The relatively-common *GET_TASKS* permission is required to perform such queries.

The functionality of this API has been changed in Android 5.0, so that an app can only use it to get information about its own Activities. For this reason, in Android 5.0 this API cannot be used anymore to detect which application is currently on top.

Accessing the proc file system. It is possible to get similar information by reading data from the *proc* file system, as previous research [6], [12] studied in detail both in a generic Linux system and in the specific setup of an Android device.

For instance, an app can retrieve the list of running applications by listing the */proc* directory and reading the content of the file: */proc/<process_pid>/cmdline*. However, most of the apps have a process running in the background even when a user is not interacting with them, so this information cannot be used to detect the app showing the top Activity.

More interestingly, we have identified a technique to detect which is the app the user is currently interacting with. In particular, the content of the file */proc/<process_pid>/cgroups* changes (from "/apps/bg_non_interactive" to "/apps") when the app on top is run by the <process_pid>. This is due to the fact that Android (using Linux cgroups) uses the specific "/apps" scheduling category for the app showing the top activity. We have tested this technique in Android 5.0 and, to the best of our knowledge, we are the first one pointing out the usage of this technique for GUI-related attacks in Android.

Finally, as studied in [6], by reading the content of */proc/<process_pid>/statm*, an application can infer the graphical state of another app, and precisely identify the specific Activity with which a user is interacting.

2) Techniques to create graphical elements mimicking already existing ones: To effectively replace an Activity of a "victim app," a convincing copy is necessary. Of course, an attacker could develop a malicious app from scratch with the same graphical elements as the original one. However, it is also possible to take the original app, change its package name, and just add the attack and information-gathering code.

The procedure of modifying an existing app (called *repackaging*) is well-known in the Android ecosystem. In the context of this paper, repackaging is a useful technique to expedite development of interfaces that mimic those of other apps. Note, however, that the attacks described in this section are entirely possible without repackaging. Detecting and defending from repackaging is outside the scope of this paper.

C. Attack app examples

In practice, malicious apps can combine multiple attack vectors and enhancing techniques to mount stealthy attacks. For instance, the attack app we implemented for our user study portraits itself as a utility app. When launched, it starts to monitor other running apps, waiting until the user switches to (or launches) the Facebook app. When that happens, it uses the *startActivity* API to spawn a malicious app on top of the genuine Facebook app. The malicious app is a repackaged version of the actual Facebook app, with the additional functionality that it leaks any entered user credentials to a remote location. To be stealthier, it informs Android that it should not be listed in the Recent Apps view.

TABLE II: Component types, flags, and *launchMode* values tested by our tool

Component type	Activity, Service, Content Provider, Broadcast Receiver
launchMode attribute	standard, singleTop, singleTask, singleInstance
startActivity flags	MULTIPLE_TASK, NEW_TASK, CLEAR_TASK, CLEAR_TOP, PREVIOUS_IS_TOP, REORDER_TO_FRONT, SINGLE_TOP, TASK_ON_HOME

We also developed a proof-of-concept malicious app that covers and mimics the home screen of a device, and demonstration videos. The displayed attack uses the "immersive" fullscreen functionality, but it can be easily adapted to use the "inescapable" fullscreen mode described in Section III-A3.

IV. State exploration of the Android GUI API

We have developed a tool to study how the main Android GUI APIs can be used to mount a GUI confusion attack. The tool automatically performs a full state exploration of the parameters of the *startActivity* API, which can be used to open Activities on top of others (including Activities of different apps). Also, our tool systematically explores all Window-drawing possibilities, to check if it is possible to create Windows that:

1) entirely cover the device's screen;
2) leave the user no way to close them or access the navigation bar.

In the following two sections, we will explain our tool in detail, and we will show what it has automatically found.

A. Study of the startActivity API

First, using the documentation and the source code as references, we determined that three different aspects influence how a newly-started Activity is placed on the Activities' stack:

- The type of Android component calling *startActivity*.
- The *launchMode* attribute of the opened Activity.
- Flags passed to *startActivity*.

Table II lists the possible Android component types, all the relevant flags and *launchMode* values an app can use.

Our tool works by first opening a "victim" app that controls the top Activity. A different "attacker" app then opens a new Activity calling the *startActivity* API with every possible combination of the listed launch modes and flags. This API is called in four different code locations, corresponding to the four different types of Android components. Our tool then checks if the newly-opened Activity has been placed on top of the "victim" app, by taking a screenshot and analyzing the captured image.

Our tool found, in Android version 4.4, the following three conditions under which an Activity is drawn on top of every other:

1) The Activity is opened by calling the *startActivity* API from a Service, a Broadcast Receiver, or a Content Provider and the *NEW_TASK* flag is used.

TABLE III: Window types and flags. Flags in *italics* are only available starting from Android version 4.4, whereas TYPEs in **bold** require the SYSTEM_ALERT_WINDOW permission.

TYPEs	TOAST, **SYSTEM_ERROR**, **PHONE**, **PRIORITY_PHONE**, **SYSTEM_ALERT**, **SYSTEM_OVERLAY**
Layout flags	IN_SCREEN, NO_LIMITS,
System-UI Visibility flags	HIDE_NAVIGATION, FULLSCREEN, LAYOUT_HIDE_NAVIGATION, LAYOUT_FULLSCREEN, *IMMERSIVE*, *IMMERSIVE_STICKY*

2) The Activity is opened by calling the *startActivity* API from another Activity and it has the *singleInstance* launch mode.
3) The Activity is opened by calling the *startActivity* API from another Activity and one of the following combinations of launch modes and flags is used:
   - *NEW_TASK* and *CLEAR_TASK* flags.
   - *NEW_TASK* and *MULTIPLE_TASK* flags, and launch mode different from *singleTask*.
   - *CLEAR_TASK* flag and *singleTask* launch mode.

We are only aware of one previous paper [6] that (manually) studies the behavior of this API for different parameters and under different conditions. Interestingly, the authors do not find all the conditions that we discovered. This underlines how the complexity of the Android API and omissions in the official documentation are prone to creating unexpected behaviors that are triggered using undocumented combinations of flags and APIs. Such behaviors are hard to completely cover through manual investigation. Hence, our API exploration tool can effectively help Android developers to detect these situations. As one example, we will now discuss how our tool revealed the existence of an "inescapable" fullscreen possibility.

B. Study of "inescapable" fullscreen Windows

We first checked the documentation and source code to determine the three different ways in which an app can influence the appearance of a Window that are relevant to our analysis:

- Modifying the Window's *TYPE*.
- Specifying certain flags that determine the Window's layout.
- Calling the *setSystemUiVisibility* API with specific flags to influence the appearance and the behavior of the navigation bar and the status bar.

Table III lists all the relevant flags and Window types an app can use.

Our tool automatically spawns Windows with every possible combination of the listed types and flags. After spawning each Window, it injects user input that should close a fullscreen Window, according to the Android documentation (e.g., a "slide" touch from the top of the screen). It then checks if, after the injection of these events, the Window is still covering the entire screen, by taking a screenshot and analyzing the captured image.

Using our tool we were able to find ways to create an "inescapable" fullscreen Window in Android 4.3, 4.4 and 5.0, which we will now briefly describe.

In particular, a Window of type SYSTEM_ERROR created with the flag NO_LIMITS, can cover the device's entire screen in Android 4.3. To specifically address this problem, a patch has been committed in the Android code before the release of the version 4.4. This patch limits the position and the size of a Window (so that it cannot cover the navigation bar) if it has this specific combination of type and flag.

However, this patch does not cover all the cases. In fact, the "immersive" fullscreen mode introduced in Android 4.4 opens additional ways to create "inescapable" fullscreen Windows, such as using the SYSTEM_ERROR type and then calling the *setSystemUiVisibility* API to set the LAYOUT_HIDE_NAVIGATION, HIDE_NAVIGATION, LAYOUT_FULLSCREEN, and IMMERSIVE_STICKY flags. We verified that the same parameters create an "inescapable" fullscreen Window in Android 5.0 as well.

It is important to notice that all the ways we discovered to create "inescapable" fullscreen Windows require using the SYSTEM_ERROR type. To fully address this problem, we propose removing this type or restricting its usage only to system components.

V. Detection via static analysis

We developed a static analysis tool to explore how (and whether) real-world apps make use of the attack vectors and enhancing techniques that we previously explained in Section III. Our goals with this tool are two-fold:

1) Study if and how the techniques described in Section III are used by benign apps and/or by malicious apps, to guide our defense design.
2) Automatically detect potentially-malicious usage of such techniques.

A. Tool description

Our tool takes as input an app's *apk* file and outputs a summary of the potentially-malicious techniques that it uses. In addition, it flags an app as *potentially-malicious* if it detects that the analyzed app has the ability to perform GUI confusion attacks.

Specifically, it first checks which permissions the app requires in its manifest. It then extracts and parses the app's bytecode, and it identifies all the invocations to the APIs related to the previously-described attack techniques. Then, the tool applies backward program slicing techniques to check the possible values of the arguments for the identified API calls. The results of the static analyzer are then used to determine whether a particular technique (or a combination of them) is used by a given application. Finally, by analyzing the app's control flow, it decides whether to flag it as (potentially) malicious.

In this section, we will discuss the static analyzer, the attack techniques that we can automatically detect, and the results we obtained by running the tool on a test corpus of over two thousand apps. We would like to note that the implementation of the basic static analysis tool (namely, the backward program slicer) is not a contribution of this paper: We reused the one that Egele et al. developed for Cryptolint [18], whose source code was kindly shared with us.

1) Program slicer: The slicer first decompiles the Dalvik bytecode of a given app by using Androguard [19]. It then constructs an over-approximation of the application's call graph representing all possible method invocations among different methods in the analyzed app. Then, a backward slicing algorithm (based on [20]) is used to compute slices of the analyzed app. Given an instruction I and a register R, the slicer returns a set of instructions that

can possibly influence the value of R. The slice is computed by recursively following the def-use chain of instructions defining R, starting from instruction I. If the beginning of a method is reached, the previously-computed call graph is used to identify all possible calling locations of that method. Similarly, when a relevant register is the return value of another method call, the backward slicer recursively continues its analysis from the return instruction of the invoked method, according to the call graph.

As most of the static analysis tools focusing on Android, the slicer may return incomplete results if reflection, class loading, or native code are used. Dealing with such techniques is outside the scope of this project.

2) Detecting potential attack techniques: In the following, we describe how our tool identifies the different attack vectors and enhancing techniques.

Draw on top. We detect if the *addView* API, used to create custom Windows, is invoked with values of the *TYPE* parameter that give to the newly-created Window a Z-order higher than that of the top-activity Window.

In addition, to detect potentially-malicious usage of a toast message, we first look for all the code locations where a toast message is shown, and then we use the slicer to check if the *setView* API is used to customize the appearance of the message. Finally, we analyze the control flow graph of the method where the message is shown to detect if it is called in a loop. In fact, to create a toast message that appears as a persistent Window, it is necessary to call the *show* API repeatedly.

App Switch. Our tool checks if:

- The *startActivity* API is used to open an Activity that will be shown on top of others. As we already mentioned, three aspects influence this behavior: the type of the Android component from which the *startActivity* API is called, the *launchMode* attribute of the opened Activity, and flags set when *startActivity* is called. We determine the first aspect by analyzing the call graph of the app, the *launchMode* is read from the app's manifest file, whereas the used flags are detected by analyzing the slice of instructions influencing the call to the *startActivity* API.
- The *moveTaskToFront* API is used.
- The *killBackgroundProcesses* API is used.

We do not use as a feature the fact that an app is intercepting the back or power buttons, as these behaviors are too frequent in benign apps and, being *passive* methods, they have limited effectiveness compared to other techniques.

Fullscreen. Our tool checks if the *setUiVisibility* API is called with flags that cause it to hide the navigation bar.

Getting information about the device state. Our tool checks if:

- The *getRunningTasks* API is used.
- The app reads from the system log. Specifically, since the native utility *logcat* is normally used for this purpose, we check if the *Runtime.exec* API is called specifying the string "logcat" as parameter.
- The app accesses files in the */proc* file system. We detect this by looking for string constants starting with "/proc" within the app.

We did not use as a feature the fact that an app is a repackaged version of another, as its usage, even if popular among malware, is not necessary for GUI confusion attacks. If desired, our system can be completed with detection methods as those presented in [13], [14].

During our study, we found that some apps do not ask (on installation) for the permissions that would be necessary to call certain APIs for which we found calls in their code. For instance, we found some applications that contain calls to the *getRunningTask* API, without having the *GET_TASKS* permission. The reason behind this interesting behavior is that this API is called by library code that was included (but never used) in the app.

In the threat model we consider for this paper, we assume that the Android security mechanisms are not violated. So, calling an API that requires a specific permission will fail if the app does not have it. For this reason, we do not consider an app as using one of the analyzed techniques if it lacks the necessary permissions.

Since the version 5.0 of Android has been released too close to the time of the writing of this paper, we expect only a very limited (and not statistically significant) number of applications using techniques introduced in this version. For this reason, we decided not to implement the detection of the techniques only available in Android 5.0.

App classification. We classify an app as suspicious if the following three conditions hold:

1) The app uses a technique to get information about the device state.
2) The app uses an attack vector (any of the techniques in the Draw on top, App Switch, Fullscreen categories)
3) There is a path in the call graph of the app where Condition 1 (check on the running apps) happens, and then Condition 2 (the attack vector) happens.

Intuitively, the idea behind our classification approach is that, to perform an effective attack, a malicious app needs to decide when to attack (Condition 1) and then how to attack (Condition 2). Also, the check for when an attack should happen is expected to influence the actual launch of this attack (hence, there is a control-flow dependency of the attack on the preceding check, captured by Condition 3).

It is important to note that our tool (and the classification rules) are designed to identify the necessary conditions to perform a GUI confusion attack. That is, we expect our tool to detect any app that launches a GUI confusion attack. However, our classification rules are not sufficient for GUI confusion attacks. In particular, it is possible that our tool finds a *legitimate* app that fulfills our static analysis criteria for GUI confusion attacks. Consider, for example, applications of the "app-locker" category. These apps exhibit a behavior that is very similar to the attacks described in Section III. They can be configured to "securely lock" (that is, disable) certain other apps unless a user-defined password is inserted. To this end, they continuously monitor running applications to check if one of the "locked" apps is opened and, when this happens, they cover it with a screen asking for an unlock password. At the code level, there is no difference between such apps and malicious programs. The difference is in the intent of the program, and the content shown to users when the app takes control of the screen.

We envision that our tool can be used during the market-level vetting process to spot apps that need manual analysis since they could be performing GUI confusion attacks. App-lockers would definitely need this analysis to check whether they are behaving according to their specification. In the following evaluation, we do not count app-lockers and similar programs as false positives. Instead, our system

has properly detected an app that implements functionality that is similar to (and necessary for) GUI confusion attacks. The final decision about the presence of a GUI confusion attack has to be made by a human analyst. The reason is that static code analysis is fundamentally unable to match the general behavior of an app (and the content that it displays) to user expectations. Nonetheless, we consider our static analysis approach to be a powerful addition to the arsenal of tools that an app store can leverage. This is particularly true under the assumption that the number of legitimate apps that trigger our static detection is small. Fortunately, as shown in the next section, this assumption seems to hold, considering that only 0.4% of randomly chosen apps trigger our detection. Thus, our tool can help analysts to focus their efforts as part of the app store's manual vetting process.

One possibility to address the fundamental problem of static code analysis is to look at the app description in the market[6]. However, this approach is prone to miss malicious apps, as cybercriminals can deceive the detection system with a carefully-crafted description (i.e., disguising their password-stealer app as an app-locker).

A second possibility to address this fundamental problem is to devise a defense mechanism that empowers users to make proper decisions. One proposal for such a defense solution is based on the idea of a trusted indicator on the device that reliably and continuously informs a user about the application with which she is interacting. We will discuss the details of this solution in Section VI.

B. Results

We ran our tool on the following four sets of apps:

1) A set of 500 apps downloaded randomly from the Google Play Store (later called *benign1*).
2) A set of 500 apps downloaded from the "top free" category on the Google Play Store (later called *benign2*).
3) A set of 20 apps described as app-lockers in the Google Play Store (later called *app-locker*).
4) A set of 1,260 apps from the Android Malware Genome project [22] (later called *malicious*).

The top part of Table IV shows the usage of five key permissions that apps would need to request to carry out various GUI confusion attacks, for each of the four different data sets we used to evaluate our tool. From this data, it is clear that three out of five permissions are frequently used by benign applications. As a result, solely checking for permissions that are needed to launch attacks cannot serve as the basis for detection, since they are too common.

The bottom part of Table IV details how frequently apps call APIs associated with the different techniques. Again, just looking at API calls is not enough for detection. Consider a simplistic (*grep-style*) approach that detects an app as suspicious when it uses, at least once, an API to get information about the state of the device and one to perform an attack vector. This would result in an unacceptable number of incorrect detections. Specifically, this approach would result in classifying as suspicious 33 apps in the *benign1* (6.6%) set and 95 in the *benign2* set (19.0%).

On the *benign1* set, our tool flagged two apps as suspicious. Manual investigation revealed that these applications monitor the user's Activity and, under specific conditions, block normal user interaction with the device. Even though these samples do not perform a GUI confusion attack (since they do not mimic the appearance of another application), they are both app-lockers. Hence, we expect our tool to report them.

[6]A similar concept has been explored in Whyper [21], a tool to examine whether app descriptions indicate the reason why specific permissions are required.

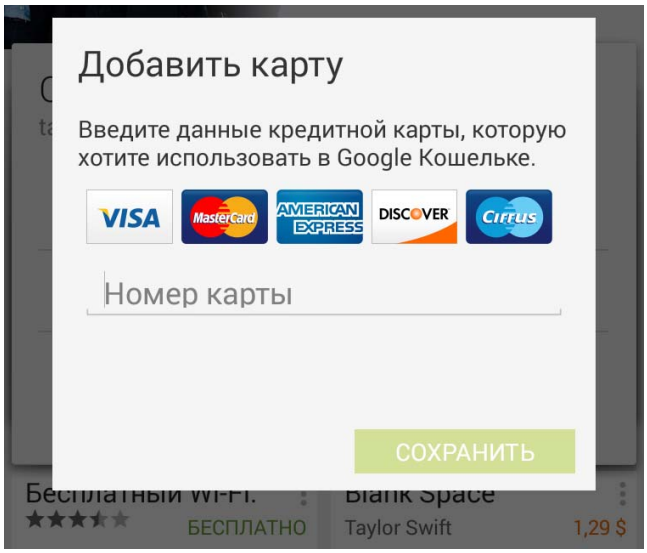


Fig. 3: A screenshot acquired while the sample of the *svpeng* malware family, detected by our tool, is attacking the user. The Activity shown in the picture (asking, in Russian, to insert credit card information) is spawned by the malware while the user is on the official Google Play Store. Data entered in this Activity is then sent to a malicious server.

On the *benign2* set, the tool detected 26 applications. When reviewing these apps, we found that two of them are app-lockers, ten of them are chat or VOIP apps, which display custom notifications using a separate mechanism than the status bar (such as stealing focus on an incoming phone call), four are games with disruptive ads, and four are "performance enhancers" (which monitor and kill the background running apps and keep a persistent icon on the screen). We also detected two anti-virus programs (which jump on top when a malicious app is detected) and one (annoying) keyboard app that jumps on top to offer a paid upgrade. We also had three false positives; two apps that could be used to take pictures, and one browser. These three apps satisfy the three conditions used to flag an app as potentially-malicious, but they do not interfere with the device's GUI.

The difference between results on sets *benign2* and *benign1* is due to the fact that popular apps are significantly bigger and more complex than the randomly-selected ones. In general, they do more and call a larger variety of APIs. Nonetheless, the total number of apps that would need to be manually analyzed is small, especially considering the set of random apps. Hence, an app store could use our system to perform a pre-filtering to check for apps that can potentially launch GUI confusion attacks, and then use manual analysis to confirm (or refute) this hypothesis.

To evaluate the detection capabilities (and false negative rate) of our tool, we randomly downloaded from the Google Play Store a set of 20 apps (called *app-locker*), described as app-lockers on the store. Since, as previously explained, this category of applications exhibits a behavior that is very similar to the attacks described in Section III, we expected our tool to detect them all. Our tool detected 18

TABLE IV: Number of apps requesting permissions used by GUI confusion attacks and number of apps using each detected technique in the analyzed data sets

permission name	*benign1* set		*benign2* set		*malicious* set		*app-locker* set	
GET_TASKS	32	6.4%	80	16.0%	217	17.2%	19	95.0%
READ_LOGS	9	1.8%	35	7.0%	240	19.1%	13	65.0%
KILL_BACKGROUND_PROCESSES	3	0.6%	13	2.6%	13	1.0%	5	25.0%
SYSTEM_ALERT_WINDOW	1	0.2%	34	6.8%	3	0.2%	10	50.0%
REORDER_TASKS	0	0.0%	4	0.8%	2	0.2%	2	10.0%
technique	***benign1* set**		***benign2* set**		***malicious* set**		***app-locker* set**	
startActivity API	53	10.6%	135	27.0%	751	59.6%	20	100.0%
killBackgroundProcesses API	1	0.2%	8	1.6%	6	0.5%	4	20.0%
fullscreen	0	0.0%	22	4.4%	0	0.0%	1	5.0%
moveToFront API	0	0.0%	0	0.0%	1	0.1%	1	5.0%
draw over using *addView* API	0	0.0%	9	1.8%	0	0.0%	3	15.0%
custom toast message	0	0.0%	1	0.2%	0	0.0%	1	5.0%
getRunningTasks API	23	4.6%	68	13.6%	147	11.7%	19	95.0%
reading from the system log	8	1.6%	18	3.6%	28	2.2%	8	40.0%
reading from *proc* file system	3	0.6%	26	5.2%	43	3.4%	4	20.0%

TABLE V: Detection of potential GUI confusion attacks.

Dataset	Total	Detected	Correctly Detected	Notes
benign1 set	500	2	2	The detected apps are both app-lockers.
benign2 set	500	26	23	10 chat/voip app (jumping on top on an incoming phone call/message), 4 games (with disruptive ads), 4 enhancers (background apps monitoring and killing, persistent on-screen icon over any app), 2 anti-virus programs (jumping on top when a malicious app is detected), 2 app-lockers, and 1 keyboard (jumping on top to offer a paid upgrade).
app-locker set	20	18	18	Of the two we are not detecting, one is currently inoperable, and the other has a data dependency between checking the running apps and launching the attack (we only check for dependency in the control flow).
malicious set	1,260	25	21	21 of the detected apps belong to the *DroidKungFu* malware family, which aggressively displays an Activity on top of any other.

out of 20 samples. Manual investigation revealed that of the two undetected samples, one is currently inoperable and the other has a data dependency between checking the running apps and launching the attack (we only check for dependency in the control flow).

Finally, we tested our tool on the *malicious* set of 1,260 apps from the Android Malware Genome project [22]. Overall, most current Android malware is trying to surreptitiously steal and exfiltrate data, trying hard to remain unnoticed. Hence, we would not expect many samples to trigger our detection. In this set, we detected 25 apps as suspicious. Upon manual review, we found that 21 of the detected samples belong to the malware family *DroidKungFu*. These samples aggressively display an Activity on top of any other, asking to the user to either grant them "superuser" privileges or enable the "USB debugging" functionality (so that the root exploit they use can work). Due to code obfuscation, we could not confirm whether the other four samples were correct detections or not. To be on the safe side, we count them as incorrect detections.

We also ran our tool on a sample of the *svpeng* [23] malware family. To the best of our knowledge, this is the only Android malware family that currently performs GUI confusion attacks. Specifically, this sample detects when the official Google Play Store is opened. At this point, as shown in Figure 3, the malicious sample spawns an Activity, mimicking the original "Enter card details" Activity. As expected, our tool was able to detect this malicious sample. Furthermore, we tested our tool on an Android ransomware sample known to interfere with the GUI (*Android.Fakedefender*). As expected, our tool correctly flagged the app as suspicious, since it uses an enhancing technique (detecting if the user is trying to uninstall it) and an attack vector (going on top of the uninstall Activity to prevent users from using it).

Finally, we used our tool to check for the "inescapable" fullscreen technique. Our tool did not find evidence of its usage in any of the analyzed sets. This suggests that removing the possibility of using this very specific functionality (as we will propose in the next section) will not break compatibility with existing applications.

VI. UI Defense mechanism

As mentioned, we complete our defense approach with a system designed to inform users and leave the final decision to them, exploiting the fact that the Android system is not being fooled by GUI attacks: Recall from Section II-A that all user-visible elements are created and managed via explicit app-OS interactions.

What compromises user security (and we consider the root cause of our attacks) is that there is simply no way for the user to know with which application she is actually interacting. To rectify this situation, we propose a set of simple modifications to the Android system to establish a trusted path to inform the user without compromising UI functionality.

TABLE VI: Examples of deception methods and whether defense systems protect against them.

	Fernandes et al. [9]	**Chen et al. [6]**	**Our on-device defense**
Keyboard input to the wrong app	✓	✗	✓
Custom input method to the wrong app (i.e., Google Wallet's PIN entry), on-screen info from the wrong app	Off by default, requires user interaction: The protection is activated only if the user presses a specific key combination.	✗	✓
Covert app switch	Keyboard only	✓ (animation)	✓
Faked app switch (through the back or power button)	Keyboard only	✗	✓
"Sit and Wait" (passive appearance change)	Keyboard only	✗	✓
Similar-looking app icon and name, installed through the market	✗ (the security indicator displays the similar-looking app icon and name. No verification of the author of the app happens.)	✗	✓
Side-loaded app, with the same app icon and name (possibly, through repackaging)	✗ (the security indicator displays the original app icon and name. No verification of the author of the app happens.)	✗	✓
Confusing GUI elements added by other apps (intercepting or non-intercepting draw-over, toast messages)	Off by default, requires user interaction	✗	✓ (yellow lock)
Presenting deceptive elements in non-immersive fullscreen mode	Off by default, requires user interaction	✗	✓
Presenting deceptive elements in immersive fullscreen mode	Off by default, requires user interaction	✗	✓ ("secret image")

In particular, our proposed modifications need to address three different challenges:

1) Understanding with which app the user is actually interacting.
2) Understanding who the real author of that app is.
3) Showing this information to the user in an unobtrusive but reliable and non-manipulable way.

Three independent components address these challenges. The combination of the states of components one and two determines the information presented to the user by component three.

Overall, two principles guided our choices:

- Offering security guarantees comparable with how a modern browser presents a critical (i.e., banking) website, identifying it during the entire interaction and presenting standard and recognizable visual elements.
- Allowing benign apps to continue functioning as if our defense were not in place, and not burdening the user with extra operations such as continuously using extra button combinations or requiring specific hardware modifications.

In particular, we wish to present security-conscious users with a familiar environment consistent with their training, using the same principles that brought different browser manufacturers to present similar elements for HTTPS-protected sites without hiding them behind browser-specific interactions.

An overview of the possible cases, how our system behaves for each of them, and the analogy with the web browser world that inspired our choices is presented in Table VII, while a more detailed description of each of our three components will be presented in the following sections.

Our implementation will be briefly described in Section VI-D, whereas Table VI exemplifies deception methods and recaps how users are defended by our system and those described in [9] and [6], which target attacks similar to the ones we described (Section VIII provides more details).

A. *Which app is the user interacting with?*

Normally, the top Activity (and, therefore, the top app) is the target of user interaction, with two important exceptions:

1) Utility components such as the navigation bar and the status bar (Section II-A) are drawn separately by the system in specific Windows.
2) An app, even if not currently on top of the Activity stack, can direct a separate Window to be drawn over the top-activity Window.

Interactions with utility components are very common and directly mediated by the system. Thus, we can safely assume that no cross-app interference can be created (the "Back" button in the navigation bar, for instance, is exclusively controlled by the top Activity) and we don't need to consider them (Point 1) in our defense.

However, as exemplified in Section III, Windows shown by different apps (Point 2) can interfere with the ability of a user to interact correctly with the top app.

While we could prohibit their creation (and thus remove row 3 of Table VII), the ability to create "always-visible" Windows is used by common benign apps: for instance, the "Facebook Messenger" app provides the ability to chat while using other apps and it is currently the most popular free app on the Google Play Store. Therefore, we have decided to simply alert users of the fact that a second app is drawing on top of the current top app, and leave them free to decide whether they want this cross-app interaction or not.

The official Android system also provides a limited defense mechanism:

1) As mentioned, a specific permission is necessary to create always-visible custom Windows. If it is granted during installation, no other checks are performed. It is impossible for the top app to prevent extraneous content from being drawn over its own Activities. *Toasts* are handled separately and do not require extra permissions.
2) The top app can use the *filterTouchesWhenObscured* API on its Views (or override the *onFilterTouchEventForSecurity* method)

TABLE VII: Possible screen states and how they are visualized.

if	*then*			
	Resulting UI state	**Visualization**	**Equivalent in browsers**	**Visualization in browsers**
no domain specified in the manifest	Apps not associated with any organization	Regular black navigation bar	Regular HTTP pages	no lock icon
Domain specified in the manifest, successful verification, *no* visible Windows from other apps	Sure interaction with a verified app	Green lock and company name	HTTPS verified page	Green lock, domain name, and (optionally) company name
Domain specified in the manifest, successful verification, visible Windows from other apps	Likely interaction with a verified app, but external elements are present	Yellow half-open lock	Mixed HTTP and HTTPS content	Varies with browsers, a yellow warning sign is common
Domain specified in the manifest, unknown validity,	Incomplete verification (networking issues)	Red warning page, user allowed to proceed	Self-signed or missing CA certificate	Usually, red warning page, user allowed to proceed
(other cases)	Failed verification	Red error page	Failed verification	Red error page

to prevent user input when content from other apps is present at the click location.

Given the attack possibilities, however, these defenses are not exhaustive for our purposes if not supplemented by the extra visualization we propose, as they still allow any extraneous content to be present over the top Activity. Moreover, the protection API can create surprising incompatibilities with benign apps (such as "screen darkeners") that use semi-transparent Windows, and does not prevent other apps' Windows from intercepting interactions (that is, it can protect only from Windows that "pass through" input).

The Android API could also be extended to provide more information and leave developers responsible to defend their own apps, but providing a defense mechanism at the operating system level makes secure app development much easier and encourages consistency among different apps.

B. *Who is the real author of a given app?*

In order to communicate to the user the fact that she is interacting with a certain app, we need to turn its unique identifier (the package name, as explained in Section II) into a message suitable for screen presentation. This message must also provide sufficient information for the user to decide whether to trust it with sensitive information or not.

To this aim, we decided to show to the user the app's developer name and to rely on the Extended-Validation [24] HTTPS infrastructure to validate it, since Extended-Validation represents the current best-practice solution used by critical business entities (such as banks offering online services) to be safely identified by their users. As we will discuss in the following paragraphs, other solutions could be used, but they are either unpractical or unsafe.

As a first example, the most obvious solution to identify an application would be to show the app's name as it appears in the market, but we would need to rely on the market to enforce uniqueness and trustworthiness of the names, something that the current Android markets do not readily provide. The existence of multiple official and unofficial markets and the possibility of installing apps via an *apk* archive (completely bypassing the markets and their possible security checks), make this a complex task. In fact, we observed several cases in which apps mimic the name and the icon of other apps, even in the official Google Play market: as an example, Figure 4 shows how a search for the popular "2048" game returns dozens of apps with very similar names and icons. For this reason, establishing a root of trust to app names and icons (such as in [9]) is fundamentally unreliable, as these are easily spoofed, even on the official market.

The only known type of vetting on the Google Play market involves a staff-selected app collection represented on the market with the "Top Developer" badge [25]. This is, to our knowledge, the only case where market-provided names can be reasonably trusted. Unfortunately, this validation is currently performed on a limited amount of developers. Moreover, no public API exists to retrieve this information. When an official method to automatically and securely obtain this information is released, our system could be easily adapted to show names retrieved from the market for certified developers, automatically protecting many well-known apps.

Relying on market operators is not, however, the only possible solution. The existing HTTPS infrastructure can be easily used for the same effect. This system also allows users to transfer their training from the browser to the mobile world: using this scheme, the same name will be displayed for their bank, for instance, whether they use an Android app or a traditional web browser.

As far as identifying the developer to the user, two main choices are possible in the current HTTPS ecosystem. The first one simply associates apps with domain names. We need to point out, however, that domain names are not specifically designed to resist spoofing and the lack of an official vetting process can be troublesome.

On the other hand, Extended-Validation (EV) certificates are provided only to legally-established names (e.g., "PayPal, Inc."), relying on existing legal mechanisms to protect against would-be fraudsters, thus preventing a malicious developer to use a name mimicking the one of another (e.g., using the name "Facebuuk" instead of "Facebook"). Extended-Validation certificate are the current mechanism in use by web browsers to safely identify the owner of a domain and they are available for less than $150 per year: in general, a substantially lower cost than the one involved in developing and maintaining any non-trivial application.

Concretely, to re-use a suitable HTTPS EV certification with our protection mechanism, the developer simply needs to provide a domain name (e.g., `example.com`) in a new specific field in the app's manifest file, and make a `/app_signers.txt` file available on the website containing the authorized public keys. During installation (and periodically, to check for revocations), this file will be checked to ensure that the developer who signed

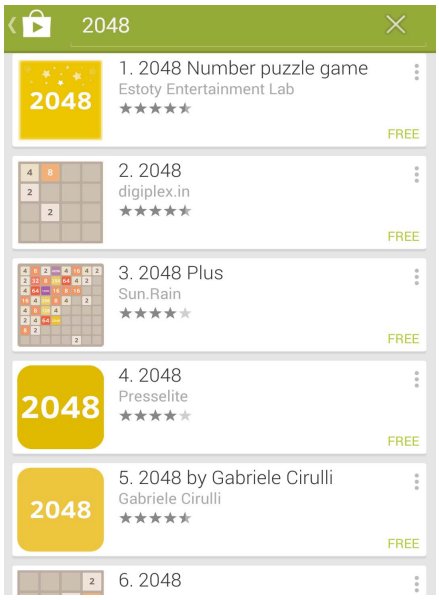


Fig. 4: A search for the popular "2048" game, returning several "clones." The app developed by the inventor of the game is listed in fifth position.

the app[7] is indeed associated with the organization that controls `example.com`. If desired, developers can also "pin" the site certificate in the app's manifest.

It should be noted that several issues have been raised on the overall structure of the PKI and HTTPS infrastructure (for a summary see, for instance, [26]). Our defense does not specifically depend on it: in fact, it should be kept in line with the best practices in how secure sites and browsers interact.

C. Conveying trust information to the user

The two components we have described so far determine the possible statuses of the screen, summarized in the first two columns of Table VII. The three right columns of Table VII present our choices, modeled after the user knowledge, training, and habit obtained through web browsers, since the mobile environment shares with them important characteristics:

- The main content can be untrusted and interaction with it can be unsafe.
- It is possible for untrusted content to purport to be from reputable sources and request sensitive user information.
- Cross-entity communications must be restricted and controlled appropriately.

Browsers convey trust-related information to the user mainly via the URL bar. Details vary among implementations, but it is generally a user element that is always visible (except when the user or an authorized page requests a fullscreen view) and that shows the main "trusted" information on the current tab.

For a web site, the main trust information is the base domain name and whether the page shown can actually be trusted to be from that domain (determined by the usage of HTTPS, and shown by a "closed lock" icon). A different element is shown when "mixed" trusted-untrusted information is present. Also, the user is warned that an attack may be in effect if the validation fails.

Most importantly, information presented in the URL bar is directly connected to the page it refers to (pages cannot directly draw on the URL bar, nor can they cause the browser to switch to another tab without also changing information shown on the URL bar).

On the Android platform, we choose the navigation bar as the "trusted" position that will behave like the URL bar. As browsers display different URL bars for different tabs, we also dynamically change information shown on the navigation bar: at every instant in time, we make sure it matches the currently visible status (e.g., the bar changes as Activities are moved on top of the stack, no matter how the transition was triggered). In other words, the security indicators are always shown as long as the navigation bar is.

The navigation bar is in many ways a natural choice as a "trusted" GUI in the Android interface, as apps cannot directly modify its appearance and its functionality is vital to ensure correct user interaction with the system (e.g., the ability for a user to go back to the "home" page or close an app).

Fullscreen apps. To ensure our defense reliability and visibility, our defense mechanism needs to deal with scenarios in which an application hides the content of the navigation bar (on which we show our security indicator) by showing a fullscreen Activity. This allows a malicious application to render a fake navigation bar in place of the original one.

For this reason, to further prove the authenticity of the information shown by our defense system, we complemented our system by using a "secret image" (also called security companion). This image is chosen by the user among a hundred different possibilities (images designed to be recognizable at a small size) and it is displayed together with our lock indicator (see Figure 1) making it impossible to correctly spoof it. In fact, a malicious application has no way to know which is the secret image selected by the user.

This system is similar to the "SiteKey" or "Sign-in Seal" mechanisms used by several websites to protect their login pages (i.e., [7], [8]), with the considerable advantage that users are constantly exposed to the same security companion whenever they interact with verified apps or with the base system.

The user has the opportunity to select the secret image during the device's first-boot or by using a dedicated system application. After that a secret image is selected, its functionality is briefly explained to the user. To prevent a malicious application from inferring the image chosen by the user, we store it in a location unreadable by non-system applications.

In addition, we modify the system so that the chosen image will not appear in screenshots (note that the Android screenshot functionality is mediated by the operating system). Also note that non-system applications cannot automatically take screenshots without explicit user collaboration.

We also propose the introduction of a fullscreen mode which still shows security indicators (but not the rest of the navigation bar), in case apps designed for fullscreen operation wish to show their credentials on some of their Activities.

Finally, we prevent applications from creating "inescapable" fullscreen Windows, by simply removing the possibility to use the

[7]Recall that all *apk* archives must contain a valid developer signature, whose public key must match the one used to sign the previous version during app updates.

specific Window's type that makes it possible (refer to Section IV-B for the technical details). As pointed out in Section V-B, we do not expect this change in the current Android API to interfere with any existing benign application.

D. Implementation

Our prototype is based on the Android Open Source Project (AOSP) version of Android (tag *android-4.4_r1.2*). Some components are implemented from scratch, others as modifications of existing system Services.

The proposed modifications can be easily incorporated into every modern Android version, since they are built on top of standard, already existing, user-space Android components. Their footprint is around 600 LOCs, and we ported them from Android 4.2 to 4.4 without significant changes.

Interaction-target app detection. This component retrieves the current state of the Activity stack and identifies the top app, by accessing information about the Activity stack (stored in the ActivityManager Service).

We also check (via the WindowManager Service) if each Window currently drawn on the device respects at least one of the following three properties:

1) The Window has been generated by a system app.
2) The Window has been generated by the top app.
3) The Window has not been created with flags that assign it a Z-order higher than that of the top-activity Window.

If all the drawn Windows satisfy this requirement, we can be sure that user interaction can only happen with the top app or with trusted system components. This distinguishes the second and third row of Table VII.

Database and author verification Service. A constantly-active system Service stores information about the currently installed apps that purport to be associated with a domain name. This Service authenticates the other components described in this section and securely responds to requests from them.

This Service also performs the HTTPS-based author verification as described previously[8]. The PackageManager system Service notifies this component whenever a new app is installed.

User interaction modification. The navigation bar behavior is modified to dynamically show information about the Activity with which the user is interacting, as described in Table VII. We also added a check in the ActivityManager Service to block apps from starting when necessary (cases listed in the fourth and fifth rows of Table VII).

VII. Evaluation

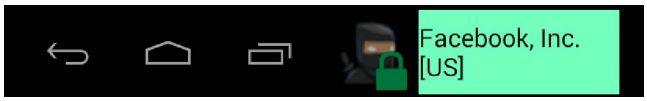
We performed an experiment to evaluate:

- The effectiveness of GUI confusion attacks: do users notice any difference or glitch when a malicious app performs a GUI confusion attack?
- How helpful our proposed defense mechanism is in making the users aware that the top Activity spawned by the attack is not the original one.

[8]For our evaluation prototype, static trust information was used to demonstrate attacks and defense on popular apps without requiring cooperation from their developers.

(a) Task B_1 and Task B_2 (real Facebook app)

(b) Task A_{std} (non-fullscreen attack app)



(c) Task A_{full} (fullscreen, defense-aware, attack app)

Fig. 5: Appearance of the navigation bar for subjects using our defense (Group 2 and Group 3), assuming they chose the dog as their security companion. Note that a non-fullscreen app *cannot* control the navigation bar: only a fullscreen app can try to spoof it. In all attacks, the malicious application was pixel-perfect identical to the real Facebook app.

We recruited human subjects via Amazon Mechanical Turk[9], a crowd-sourced Internet service that allows for hiring humans to perform computer-based tasks. We chose it to get wide, diversified subjects. Previous research has shown that it can be used effectively for performing surveys in research [27]. IRB approval was obtained by our institution.

We divided the test subjects into three groups. Subjects in *Group 1* used an unmodified Android system, to assess how effective GUI confusion attacks are on stock Android. Subjects in *Group 2* had our on-device defense active, but were not given any additional explanation of how it works, or any hint that their mobile device would be under attack. This second group is meant to assess the behavior of "normal" users who just begin using the defense system, without any additional training. To avoid influencing subjects of the first two groups, we advertised the test as a generic Android "performance test" without mentioning security implications. Finally, subjects in *Group 3*, in addition to using a system with our on-device defense, were also given an explanation of how it works and the indication that there might be attacks during the test. This last group is meant to show how "power users" perform when given a short training on the purpose of our defense.

Subjects interacted through their browser[10] with a hardware-accelerated emulated Android 4.4 system, mimicking a Nexus 4 device. For subjects in Group 2 and Group 3, we used a modified Android version in which the defense mechanisms explained in Section VI had been implemented.

A. Experiment procedure

The test starts with two general questions, asking the subjects i) their age and ii) if they own an Android device. These questions are repeated, in a different wording, at the end of the test. We use these

[9]https://www.mturk.com
[10]We used the noVNC client, http://kanaka.github.io/noVNC

TABLE VIII: Results of the experiment with Amazon Turk users.
Percentages are computed with respect to the number of *Valid Subjects*.

	Group 1: Stock Android	**Group 2**: Defense active. Subjects not aware of the possibility of attacks	**Group 3**: Defense active, briefly explained. Subjects aware of the possibility of attacks
Total Subjects	113	102	132
Valid Subjects	99	93	116
Subjects answering correctly to Tasks:			
B_1 and B_2	67 (67.68%)	70 (75.27%)	85 (73.28%)
A_{std}	19 (19.19%)	60 (64.52%)	80 (68.97%)
A_{full}	17 (17.17%)	71 (76.34%)	86 (74.14%)
A_{std} and A_{full}	8 (8.08%)	55 (59.14%)	67 (57.76%)
A_{std} and B_1 and B_2	4 (4.04%)	51 (54.84%)	73 (62.93%)
A_{full} and B_1 and B_2	6 (6.06%)	63 (67.74%)	76 (65.52%)
A_{std} and A_{full} and B_1 and B_2	2 (2.02%)	50 (53.76%)	66 (56.90%)

questions to filter out subjects that are just answering randomly (once given, each answer is final and cannot be reviewed or modified).

Then, subjects in Group 2 and Group 3 are asked to choose their "security companion" in the emulator (which is, for example, the image of the dog in Figure 1), picking among several choices of images as they would be asked to do at the device's first boot to set up our defense. The selected image will be then shown in our defense widget on the navigation bar.

Then, subjects are instructed to open the Facebook app in the emulator. We chose this particular app because it is currently the second most popular free app, and it asks for credentials to access sensitive information. The survey explains to our subjects that the screen of a real Nexus 4 device is being streamed to their browser, and that the application they just opened is the real one. We have included this step because, in a previous run of our experiment, a sizable amount of our subjects did not believe that the phone was "real," and so they did not considered as "legitimate" *any* interaction they had with it.

Subjects are then instructed to open the Facebook app in the emulator several times, leaving them free to log in if they want to. After a few seconds, we hide the emulator and ask our subjects about their interaction. Specifically, we ask if they think they interacted with the original Facebook application as they did at the very beginning. Subjects had to respond both in a closed yes-no form and by providing a textual explanation. We used the closed answers to quantitatively evaluate the subjects' answers and the open ones to get insights about subjects' reasoning process and to spot problems they may have had with our infrastructure.

We decided against evaluating the effectiveness of our defense by checking if users have logged in. This is because, in previous experiments, we noticed that security-conscious users would avoid surrendering their personal credentials in an online survey (regardless of any security indicator), but would not be careful if provided with fake credentials. Instead, we decided to ask the subjects to perform four different tasks: B_1, B_2, A_{std}, and A_{full}.

During Task B_1 and Task B_2, subjects are directed to open the Facebook app. In these two tasks, this will simply result in opening the real Facebook app.

In Task A_{std} we deliver the attack described in Section III-C while the subjects are opening Facebook. As a result, the device will still open the real Facebook app, but on top of it there will be an Activity that (even though it looks just like the real Facebook login screen) actually belongs to our malicious app. In Groups 2 and Group 3, which have our defense active, our widget in the navigation bar will show that the running app is not certified, by showing no security indicator on the navigation bar. Therefore, subjects in Group 2 and 3 may detect the attack by noticing the missing widget.

Differently, in Task A_{full}, we simulate a fullscreen attack. In this case, our malicious app will take control of the whole screen. The malicious app can mimic perfectly the look and feel of anything that would be shown on the screen, but it cannot display the correct security companion (because it does not know which one it is). The fullscreen attack app must then mimic to its best the look of our defense widget, but it will show a different security companion, hoping that the user will not notice. For this reason, subjects in Group 2 and Group 3 can detect the attack if (and only if) they notice that our widget is not showing the "correct" security companion they had chosen. Note that this puts our defense in its worst-case scenario, with pixel-perfect reproduction of the original app and the defense widget except for the user-selected secret image.

Note that for subjects in Group 1 this task looks exactly the same as Task A_{std}: if the navigation bar never shows security indicators, we assume it would be counterproductive for an attacker to drastically alter it by showing a "spoofed" security indicator.

The four tasks are presented in a randomized order. This prevents biasing results in case performing a task during a specific step of the experiment (e.g., at the beginning) could "train" subjects to answer better in subsequent tasks.

Figure 5 summarizes what has been shown on the navigation bar to the subjects in Group 2 and Group 3 during the execution of the different tasks.

B. Results

In total, 347 subjects performed and finished our test. However, we removed 39 subjects because the control questions were inconsis-

tent (e.g., How old are you? More than 40. What's your age? 21.), the same person tried to retake the test, or the subject encountered technical problems during the test. This left us with 308 valid subjects in total. The results of the experiment are shown in Table VIII.

The vast majority of subjects in Group 1, using stock Android, were not able to correctly identify attacks and often noticed no difference (typically, answering that they were using the real Facebook in all tasks) or reported minimal animation differences due to the reduced frame rate and emulator speed (unrelated to the attacks). This corroborates our opinion that these attacks are extremely difficult to identify. In particular, only 8.08% of the subjects detected both attacks and only 2.02% of the subjects answered all questions correctly. Manual review of the textual answers revealed that this happened randomly (that is, the subjects did not notice any relevant graphical difference among the different tasks).

Comparing results for Group 1 and Group 2, it is clear that the defense helped subjects in detecting the attacks. Specifically, the percentage of correct detections increased from 19.19% to 64.52% for Task A_{std} ($\chi^2 = 40.68$, $p < 0.0001$)[11] and from 17.17% to 76.34% ($\chi^2 = 67.63$, $p < 0.0001$) for Task A_{full}. Also, the number of subjects able to answer correctly all times increased from 2.02% to 53.76% ($p < 0.0001$, applying Fisher's exact test).

Comparing detection results of the two attacks, we found that the detection rate for the fullscreen attack is slightly better than the one for the non-fullscreen one. However, this difference is not statistically significant. In particular, considering Group 2 and Group 3 together, 66.99% of the subjects answered correctly during Task A_{std} and 75.12% answered correctly during Task A_{full} ($\chi^2 = 3.36$, $p = 0.0668$).

We also noticed that the number of subjects answering correctly during the non-attack tasks (Tasks B_1 and B_2) did not increase when our defense was active. In other words, we did not find any statistical evidence that our defense leads to false positives.

Finally, results for Group 2 and Group 3 are generally very similar, with just a slight (not statistically significant) improvement for subjects in Group 3 in the ability to answer correctly all questions ($\chi^2 = 0.21$, $p = 0.6506$). This may hint to the fact that our additional explanation was not very effective, or simply to how the mere introduction of a security companion and defense widget puts users "on guard," even without specific warnings.

C. Limitations

As mentioned, we took precaution not to influence users' choices during the experiment. In particular, subjects in Group 2 used a system with our defense in place, but without receiving any training about it before. Nonetheless, they had to set up their security companion prior to starting the experiment, as this step is integral to our defense and cannot be skipped when acquiring a new device. We designed our experiment to simulate, as accurately as possible, the first-use scenario of a device where our proposed defense is in place. In this scenario, users would be prompted to choose a security companion during the device's first boot. We acknowledge, however, that this step may have increased the alertness of our subjects so that our results may not be completely representative of the effect that our defense widget has on users, especially over a long period of time.

[11] We evaluate results using 95% confidence intervals. Applying the Bonferroni correction, this means that the null hypothesis is rejected if $p < 0.01$.

Similarly, the fact that subjects, at the beginning of the experiment, were made to interact with the original Facebook application may have helped them in answering to the different tasks. However, we assume it is unlikely that users are being attacked by a malicious app performing a GUI confusion attack during the very first usage of their device.

It is also possible that the usage of an emulator, accessed using a web browser, may have had a negative impact on the subjects' ability to detect our attacks. It should be noted, however, that the usage of an x86 hardware-accelerated emulator (and VNC) resulted in a good-performance, to the point we would recommend this setup to future experimenters (unless, of course, they have the time and resources to gather enough participants and use real devices).

Finally, there is a possibility that the subject's network was introducing delays. From the network's point of view, the emulation appears as a continuous VNC session from the beginning to the end. This setup should not specifically affect individual tasks, but may have caused some jitter for subjects.

VIII. Related work

As mentioned in the introduction, previous papers have already shed some light on the problem of GUI confusion attacks in Android. In particular, [3] describes tapjacking attacks in general, whereas [4] focuses on tapjacking attacks against WebViews (graphical elements used in Android to display Web content). Felt et al. [5] focus on phishing attacks on mobile devices deriving from control transfers (comparable to the "App Switching" attacks we described), whereas Chen et al. [6] describe a technique to infer the UI state from an unprivileged app and present attack examples. Our paper generalizes these previously-discovered techniques by systematizing existing exploits and introducing additional attack vectors. We also confirmed the effectiveness of these attacks through a user study. More importantly, we additionally proposed two general defense mechanisms and evaluated their effectiveness.

Fernandes et al. present a GUI defense focusing on keyboard input in [9]: the "AuthAuth" system augments the system keyboard by presenting a user-defined image and the app name and icon. Our proposed defense system uses the same "UI-user shared secret" mechanism: in both cases, users must first choose an image that will be known only by the OS and the user, making it unspoofable for an attacking app.

However our works significantly differ in how this mechanism is used and what is presented to the user. For instance, as we have shown before (e.g., see Figure 4), app names and icons are not valid or reliable roots of trust, as they are easy to spoof. Apps with similar-looking name and icons are commonly present in Android markets, and fake apps with the same name and icon can be side-loaded on the device. Our work, instead, establishes a root of trust to the author of the app, and extends the covered attack surface by considering more attack scenarios and methods. In particular, we opted to secure all the user interactions instead of focusing only on the keyboard, because users interact with apps in a variety of ways. For instance, some payment apps (e.g., Google Wallet) use custom PIN-entry forms, while others get sensitive input such as health-related information through multiple-choice buttons or other touch-friendly methods.

Other research efforts focus on the analysis of Android malware. Zhou et al. performed a systematic study of the current status of malware [22], whereas other studies focus on the specific techniques

that current malicious applications use to perform unwanted activities. A frequently-used technique is repackaging [14], [15]. In this case, malware authors can effectively deceive users by injecting malicious functionality in well-known, benign-looking Android applications. As previously mentioned in Section III-B2, this technique can be used in combination with our attack vectors to make it easy for attackers to mimic the GUI of victim apps.

Roesner et al. [28] studied the problem of embedded user interfaces in Android and its security implications. Specifically, they focus on the common practice of embedding in an app graphical elements, created by included libraries. The problem they solve is related and complementary to the one we focus on. Specifically they focus on how users interact with different elements within the same app, whereas we focus on how users interact with different apps.

Felt et al. performed a usability study to evaluate how users understand permission information shown during the installation process of an app [29]. They showed that current permission warnings are not helpful for most users and presented recommendations for improving user attention. Possible modifications to how permissions are shown to users and enforced have been also studied in Aurasium [30]. Our work has in common with these the fact that it proposes a set of modifications to give users more information on the current status of the system, although we address a different threat.

Many studies investigated how to show security-related information and error messages in browsers, both from a general prospective [31]–[33] and specifically for HTTPS [34]–[38]. Akhawe et al. [38] showed that proper HTTPS security warning messages are effective in preventing users from interacting with malicious websites. The knowledge presented by these works has been used as a baseline for our proposed defense mechanism. It should be noted, however, that other studies have shown that indicators are not always effective. In fact, over the years, the situation has significantly improved in browsers: compare, for instance, the almost-hidden yellow lock on the status bar of Internet Explorer 6 from [37] with Figure 1. We believe that our solution may also have benefited from the EV-style presentation of a name in addition to a lock and the consequent increase in screen area. In general, effectively communicating the full security status of user interactions is an open problem.

Phishing protection has been extensively studied in a web browser context (e.g., in [39]–[41]) and is commonly implemented using, for example, blacklists such as Google's SafeBrowsing [42]. Our work is complementary to these approaches and explores GUI confusion attacks that are not possible in web browsers.

Finally, the problem of presenting a trustworthy GUI has been studied and implemented in desktop operating systems, either by using a special key combination [43] or decorations around windows [44]. Given the limited amount of screen space and controls, applying these solutions in mobile devices would be impossible in an unobtrusive way.

IX. Conclusion

In this paper, we analyzed in detail the many ways in which Android users can be confused into misidentifying an app. We categorized known attacks, and disclose novel ones, that can be used to confuse the user's perception and mount stealthy phishing and privacy-invading attacks.

We have developed a tool to study how the main Android GUI APIs can be used to mount such an attack, performing a full state exploration of the parameters of these APIs, and detecting problematic cases.

Moreover, we developed a two-layered defense. To prevent such attacks at the market level, we have developed another tool that uses static analysis to identify code in apps that could be leveraged to launch GUI confusion attacks, and we have evaluated its effectiveness by analyzing both malicious applications and popular benign ones.

To address the underlying user interface limitations, we have presented an on-device defense system designed to improve the ability of users to judge the impact of their actions, while maintaining full app functionality. Using analogies with how web browsers present page security information, we associate reliable author names to apps and present them in a familiar way.

Finally, we have performed a user study demonstrating that our on-device defense improves the ability of users to notice attacks.

Acknowledgments

We would like to thank all the participants in our user study that provided useful and detailed feedback.

This material is based upon work supported by DHS under Award No. 2009-ST-061-CI0001, by NSF under Award No. CNS-1408632, and by Secure Business Austria. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of DHS, NSF, or Secure Business Austria.

This material is also based on research sponsored by DARPA under agreement number FA8750-12-2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] comScore, "The U.S. Mobile App Report," http://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report, 2014.

[2] ESET, "Trends for 2013," http://www.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf.

[3] M. Niemietz and J. Schwenk, "UI Redressing Attacks on Android Devices," *Black Hat Abu Dhabi*, 2012.

[4] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking Attacks on Web in Android, iOS, and Windows Phone," in *Proceedings of the 5th International Conference on Foundations and Practice of Security (FPS)*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 227–243.

[5] A. P. Felt and D. Wagner, "Phishing on mobile devices," *Web 2.0 Security and Privacy*, 2011.

[6] 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 the 23rd USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2014, pp. 1037–1052.

[7] Bank of America, "SiteKey Security," https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go.

[8] Yahoo, "Yahoo Personalized Sign-In Seal," https://protect.login.yahoo.com.

[9] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "TIVOs: Trusted Visual I/O Paths for Android," University of Michigan CSE Technical Report CSE-TR-586-14, 2014.

[10] TrendLabs, "Tapjacking: An Untapped Threat in Android," http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/, December 2012.

[11] TrendLabs, "Bypassing Android Permissions: What You Need to Know," http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/, November 2012.

[12] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, May 2012, pp. 143–157.

[13] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 62–81.

[14] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY)*. New York, NY, USA: ACM, 2012, pp. 317–326.

[15] W. Zhou, X. Zhang, and X. Jiang, "AppInk: Watermarking Android Apps for Repackaging Deterrence," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*. New York, NY, USA: ACM, 2013, pp. 1–12.

[16] P. De Ryck, N. Nikiforakis, L. Desmet, and W. Joosen, "TabShots: Client-side Detection of Tabnabbing Attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*. New York, NY, USA: ACM, 2013, pp. 447–456.

[17] Google, "Using Immersive Full-Screen Mode," https://developer.android.com/training/system-ui/immersive.html.

[18] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2013, pp. 73–84.

[19] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Black Hat Abu Dhabi*, 2011.

[20] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[21] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards Automating Risk Assessment of Mobile Applications," in *Proceedings of the 22nd USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2013, pp. 527–542.

[22] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, May 2012, pp. 95–109.

[23] R. Unuchek, "The Android Trojan Svpeng Now Capable of Mobile Phishing," http://securelist.com/blog/research/57301/the-android-trojan-svpeng-now-capable-of-mobile-phishing/, November 2013.

[24] CA/Browser Forum, "Guidelines For The Issuance And Management Of Extended Validation Certificates," https://cabforum.org/wp-content/uploads/Guidelines_v1_4_3.pdf, 2013.

[25] Google, "Featured, Staff Picks, Collections, and Badges," https://developer.android.com/distribute/googleplay/about.html#featured-staff-picks.

[26] J. Clark and P. van Oorschot, "SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 511–525.

[27] A. Kittur, E. H. Chi, and B. Suh, "Crowdsourcing User Studies with Mechanical Turk," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2008, pp. 453–456.

[28] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *Proceedings of the 22nd USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2013, pp. 97–112.

[29] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the Eighth Symposium On Usable Privacy and Security (SOUPS)*. New York, NY, USA: ACM, 2012, pp. 3:1–3:14.

[30] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21st USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2012, pp. 27–27.

[31] Z. E. Ye and S. Smith, "Trusted Paths for Browsers," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 263–279.

[32] A. Neupane, N. Saxena, K. Kuruvilla, M. Georgescu, and R. Kana, "Neural Signatures of User-Centered Security: An fMRI Study of Phishing and Malware Warnings," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[33] Y. Niu, F. Hsu, and H. Chen, "iPhish: Phishing Vulnerabilities on Consumer Electronics." in *Proceedings of the 1st Conference on Usability, Psychology, and Security (UPSEC)*, 2008.

[34] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," in *Proceedings of the 18th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2009, pp. 399–416.

[35] J. Lee, L. Bauer, and M. L. Mazurek, "The Effectiveness of Security Images in Internet Banking," *Internet Computing, IEEE*, vol. 19, no. 1, pp. 54–62, Jan 2015.

[36] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2012, pp. 50–61.

[37] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, May 2007, pp. 51–65.

[38] D. Akhawe and A. P. Felt, "Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness," in *Proceedings of the 22nd USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2013, pp. 257–272.

[39] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell, "Client-side defense against web-based identity theft," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[40] R. Dhamija and J. D. Tygar, "The Battle Against Phishing: Dynamic Security Skins," in *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*. New York, NY, USA: ACM, 2005, pp. 77–88.

[41] E. Kirda and C. Kruegel, "Protecting users against phishing attacks with AntiPhish," in *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, vol. 1, July 2005, pp. 517–524 Vol. 2.

[42] Google, "Safe Browsing," http://www.google.com/transparencyreport/safebrowsing/.

[43] D. Clercq and Grillenmeie, *Microsoft Windows Security Fundamentals*. (Chapter 5.2.1), Connecticut, USA: Digital Press, October 2006.

[44] J. Rutkowska, "Qubes OS Architecture (Section 5.3)," http://files.qubes-os.org/files/doc/arch-spec-0.3.pdf, January 2010.