TIVOs: Trusted Visual I/O Paths for Android

Earlence Fernandes
earlence@umich.edu

Qi Alfred Chen
alfchen@umich.edu

Georg Essl
gessl@umich.edu

J. Alex Halderman
jhalderm@umich.edu

Z. Morley Mao
zmao@umich.edu

Atul Prakash
aprakash@umich.edu

University of Michigan, Ann Arbor

University of Michigan, Ann Arbor Technical Report CSE-TR-586-14

Abstract

Stealthy pixel-perfect attacks on smartphone apps are a class of phishing attacks that rely on visual deception to trick users into entering sensitive information into trojan apps. We introduce an operating system abstraction called Trusted Visual I/O Paths (TIVOs) that enables a user to securely verify the app she is interacting with, only assuming that the operating system provides a trusted computing base. As proof of concept, we built a TIVO for Android, one that is activated any time a soft keyboard is used by an application (e.g., for password entry) so that the user can reliably determine the app that receives the user's keyboard input. We implemented TIVO by modifying Android's user-interface stack and evaluated the abstraction using a controlled user study where users had to decide whether to trust the login screen of four different applications that were randomly subjected to two forms of pixel-perfect attacks. The TIVO mechanism was found to significantly reduce the effectiveness of pixel-perfect attacks, with acceptable impact on overall usability and only modest performance overhead.

1. INTRODUCTION

We use our smartphones for a variety of tasks including sensitive actions such as logging into confidential email, entering sensitive personal information in tax filing apps and using passwords to authenticate ourselves to mobile banking apps. Phishing attacks, traditionally a problem on the web, is encroaching into the mobile space. There are several types of attacks that rely on visual deception and a lack of user knowledge falling under the umbrella term of phishing. Stealthy *pixel-perfect trojans* are a class of phishing attacks that imitate the look-and-feel of genuine apps, but only under certain conditions. For example, an interesting game could include code to generate a UI that imitates a banking app. This trojan is stealthy because it does not present the imitated UI immediately and blatantly, but utilizes techniques such as activity launch hijacking [4, 8] and UI state inference [7] to launch its fake UI windows when the user actually attempts to launch a genuine version of the target app. The fake UI windows may be presented out of context as well, borrowing techniques from clickjacking [16].

Phishing attacks often take advantage of UI preemption features of an operating system to achieve visual deception [24, 21, 1]. UI preemption is the process where the current foreground app is preempted by a background service and another app is brought to the foreground, without user intervention. This is a way of supporting common features such as alarms or reminders and is an embodiment of a command design pattern. Recent work by Chen et al. [7] shows that launching phishing attacks by taking advantage of UI preemption can be very stealthy.

Besides UI preemption, mobile UI phishing attacks are also achieved indirectly by hijacking the process which triggers screen redirection. On Android, the IPC delivery process (for interaction with Activities, Services or Broadcast Receivers) was found to be exploitable for hijacking [12, 11, 28]. Besides IPC, Z. Xu et al. [31] find that the Android notification delivery can also be exploited to allow an installed trojan application to launch phishing attacks or anonymously post spam notifications. These attacks depend on the triggering of certain conditions, for example IPC delivery, which may be limited in attack surface compared to direct UI preemption based phishing.

One solution, used on the web, is to display a security indicator, such as an HTTPS status indicator, security toolbars that make the current URL explicit, and secret images (e.g., SiteKey images that are commonly found on banking sites). Unfortunately, [22] found that both HTTPS toolbar as well as secret images at banking web sites were "not effective". [30] found that a security toolbar is often missed.

We conclude that designing tamper-resistant and effective security indicators is a difficult problem that has resisted a good solution. To get tamper-resistance, the OS could potentially reserve an area of a screen to provide trustworthy information about the app that a user interacts with, as has been suggested as far back as in the Terra system [15]. Unfortunately, this has not been deemed to be practical even on desktops. On smartphones, the available screen real estate is significantly smaller and many apps like to use the entire screen. Finally, as studies with browser-based security indicators suggest, a user evaluation is important to assess effectiveness.

Our solution does not require a reserved area on the screen for security indicators. This results in another challenge — a malicious app could attempt to fake the security indicators. For example, a browser's HTTPS security indicators are not truly tamper-resistant. A malicious app, even an unprivileged one, could potentially spoof a browser's look-

and-feel, including the security indicators. Similar challenge arises with designing tamper-resistant security indicators for mobile apps and we address that.

To address the problem of stealthy pixel-perfect phishing attacks and, more generally, the problem that a user lacks a robust way to determine the app he/she interacts with, we propose *Trusted Visual I/O paths (TIVOs)*. TIVOs enable users to reliably identify the app before providing input. The design further guarantees a race-free solution from time-of-check-time-of-use (TOCTOU) errors [29] in that another app cannot hijack the input by, for example, overlaying a transparent screen on top of the original app after the user identifies the app and before the user provides the input.

We implemented a concrete TIVO called *AuthAuth* on the Android platform that is based on the most widely used input method for userids, passwords, and credit card numbers — the soft keyboard. AuthAuth automatically launches a tamper-resistant security indicator, which we term as a *secure annotation*, that reliably identifies the foreground app and ensures that the keyboard input can only go to the identified app. Under normal usage of apps, when the keyboard is not being used, secure annotations need not be displayed, saving screen-estate. On Android, soft keyboards are themselves apps. AuthAuth is implemented inside the OS and is independent of the implementation of the soft keyboard that may be in use.

Contributions:

- We introduce and explore the Trusted Visual I/O operating system abstraction as a technique to assist users in identifying stealthy pixel-perfect phishing attacks (Section 4). We designed AuthAuth security indicators to be tamper-resistant (Section 4.2). In particular, AuthAuth security indicators are generated by code that is part of the trusted base and not under the control of the apps. When displayed, they are shown above any graphical layer that is produced by an app.
- On Android, we implemented a TIVO abstraction, called AuthAuth (Section 5), that presents a secure annotation of the current foreground app *automatically* whenever a soft keyboard is used for data input in *any* application without requiring existing Android applications to be modified.
- We compared the effectiveness of an AuthAuth-based system on a real device against standard Android in defending against pixel-perfect phishing attacks (Section 7) in a user study with 22 users. AuthAuth was found to significantly reduce the effectiveness of such attacks. With AuthAuth, users were almost always able to correctly determine when an app was being attacked whereas, on standard Android, users could not reliably identify when the app was being attacked. Furthermore, without AuthAuth, all users logged into fake apps multiple times, indicating trouble in identifying a phishing attack, whereas with AuthAuth, 19 out of 22 users were able to successfully identify all phishing attacks.

Anonymized TIVO video demos from our implementation illustrating the attacks and defenses can be found at `http://sites.google.com/site/authauthdemo/` [2].

The TIVO abstraction is flexible and is applicable to other scenarios besides soft keyboards. On Android phones, sensitive input may also occur through touch interactions. For example, the drug helper app WebMD presents a long list of drugs that the user can select from. An attacker interested in illegally collecting information about drug usage may launch a pixel-perfect phishing attack by presenting an identical UI using activity hijacking. To enable a user to protect herself in such situations, as proof-of-concept, we also describe a TIVO that activates on demand when the user presses a key escape combination. If the user presses the Power + Volume Up physical buttons, a secure annotation is displayed that identifies the app and permits input to only go to that app.

2. AN EXAMPLE ATTACK SCENARIO

Here we describe how a pixel-perfect phishing attack works on Android. Android allows apps to consist of services and activities. Services can run in the background while other apps run.

Consider a scenario where Alice finds a new and upcoming app called "Wave Guide" (written by Bob) that provides information on where to find the best surfing spots around a specified location. Alice downloads this app and uses it normally. Unknown to Alice, this app is a stealthy trojan and its real purpose is to launch phishing attacks.

When Alice logs into Facebook, the service belonging to Wave Guide notices this launch by continuously polling the top of Android's activity stack[1]. This service immediately launches its pixel-perfect copy of Facebook login and the unwary user sees something similar to Figure 1, a normal looking Facebook login screen. Alice proceeds to log in. Wave Guide receives the password, leaks it to Bob, and simply exits (or presents a login failure screen that appears to be from Facebook and then exits). To the user, it appears that login to Facebook did not succeed and retries. This time the Wave Guide app does not interfere.

Note that activity hijacking is not a requirement for the attack to succeed. Wave Guide's service could launch a look-alike of the Facebook login page on its own at an opportune time, e.g., when a device is woken up from sleep. If the user believes the page to be from Facebook, the attack may succeed.

3. THREAT MODEL AND TRUSTED BASE

Our trusted computing base consists of the Operating System of the phone, standard Android Framework services, and any alternate keyboards (e.g., Swype [26]) installed by the user. This work is designed to ensure that the user can reliably determine the app that receives user's input.

Apps on the phone are permitted to control the entire screen. We do not require any screen area to be reserved for the OS

[1]Scanning the top of the Activity stack is not the only method available to services on Android to monitor other app's activities. Other stealthier methods also exist that do not require any permissions at all [7].

to provide security indicators. Furthermore, the user may have installed one or more malicious apps that launch activity hijacking attacks on other apps, say, via UI preemption in order to steal information entered by the user. Our work does not protect against covert or side channels.

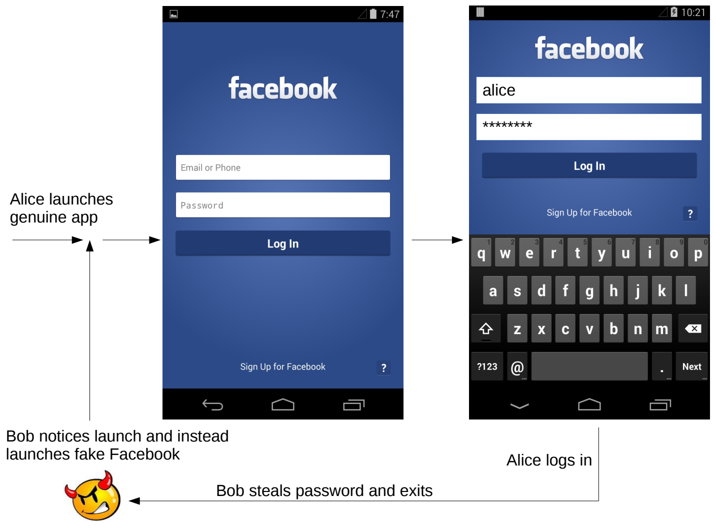

Figure 1: An attack scenario to steal passwords on Android. Bob provides a malicious app that is useful, but also contains a Trojan service to steal the user's Facebook credentials. It launches a Facebook-lookalike page or a transparent overlay at an opportune time, e.g., when the user launches the real Facebook app. Any input entered by the user is collected by the malicious app and leaked to Bob.

4. DESIGN

4.1 Security properties of Trusted Visual Paths

A trusted visual I/O path enables a user to securely determine the app that will receive input. A TIVO is established when the user requests the OS to bring up a secure annotation that contains information about the current foreground app. Once a TIVO is established, the security properties of a TIVO are activated. The properties listed below work together to ensure that a user will have reliable information about the foreground app.

1. *Annotation Verifiability:* The app identity information displayed in a secure annotation must be verifiable by the user. The user must be provided sufficient information in the secure annotation to identify the app as well as whether the annotation displayed is trustworthy. This also implies that the secure annotation must be both tamper-resistant (tampering should be detectable) and hard-to-forge (an app should not be able to forge an annotation).

2. *Atomicity of annotation display and subsequent user input for an app:* Once the system has displayed an annotation for the foreground app, it must also ensure that subsequent input by the user can only go that app (unless the user explicitly switches to another app).

The above requirements imply that once a secure annotation is displayed for an app, another app must not be able to surreptitiously become the foreground app and receive user's input.

As shown in Figure 2, a trusted visual I/O path (TIVO) is constructed out of two elements: a tamper-resistant and difficult-to-forge security-indicator, called *secure annotation* (SA) that the user can use to identify the foreground app, and an *execution monitor* within the operating system that activates the SA either automatically or upon user request and ensures Security Property 2. The execution monitor ensures that once an SA is shown (i.e., a TIVO is active), another app cannot become the the foreground app without an explicit user action; the foreground app will be the one to receive subsequent input.

Graphical subsystems use the notion of *Z-index* (also called Z-order) [14]. Graphical objects with higher Z-index appear on top of objects with lower Z-index. Our design ensures that the SA is displayed at a higher Z-index than any of the foreground app's graphical elements, ensuring that the SA is fully visible. This is done by maintaining an invariant within the rendering loop of the window manager. The secure annotation is always maintained at the highest possible Z value for the current rendering cycle. Furthermore, we display an SA *after* and app has gained focus but *before* the user provides input.

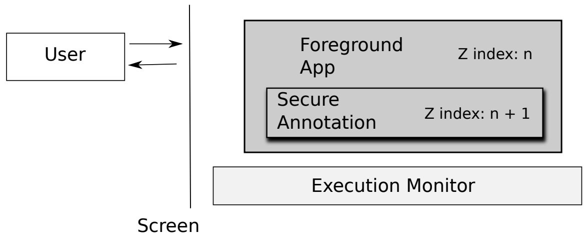

Figure 2: TIVO architectural overview

4.2 Design of the secure annotation

The secure annotation is a tamper-resistant and hard-to-forge security indicator whose purpose is to identify the current foreground app that will be receiving the user input. Clearly, for it be hard-to-forge, it has to be based on a shared secret between the user and the trusted computing base that the user can easily identify.

Images have been previously used as shared secrets in identifying web sites in the form of Site Authentication Images (SAI) for web banking [22]. There are two challenges in simply using SAIs as they are commonly used on bank web sites. First, as explained in [9], using stock images are easier to spoof and harder for users to remember. Second, in our case, the user also needs to be able to distinguish among apps. A different image per app could be used, but that also increases the workload for the user to create and remember the associations.

Our design extends the SAI concept in a manner that solves the above issues for mobile apps. We require the user to take a *single* personal image (taking advantage of the ubiquity of smartphone cameras) one-time to enable secure annotations, preferably when the phone is new or in factory-reset state when there is no malware on the device. This image should be of an object that is easy to identify for the user, but hard to guess and reproduce for apps. For example, it could be a photo of a personalized item with an easy-to-recognize background around one's home. A single image also frees the user from having to remember many standard images as with SAIs in online banking. Since the image is personalized, our hypothesis is that it will be easier for the user to remember and avoid the common pitfalls with SAIs for web sites. Our solution requires this image to be only shared with the OS on the device – we term it the *OS secret image*.

The image is phone-wide and exists to prove to the user that a secure annotation is generated by the OS. The image is not accessible to apps (see Section 5 on how it is protected from apps). The OS secret image forms a trust anchor and users may only trust the other information on a secure annotation when they have verified the existence of their secret image.

Contents of an SA. Figure 3 depicts the contents of the secure annotation in AuthAuth. The SA has two items whose contents must be verified by a user: the OS secret image that is in the middle and some app-identifying information. We chose both the app's familiar icon and its name as the app-identifying information; the app's icon is shown on the left of the OS secret image and its name is shown on the right.

The user has a simple verification procedure: she verifies that (1) the OS secret image matches the personalized image that was set and (2) the app's icon and name match what is expected for the foreground app (e.g., if displaying the Facebook's login screen, the icon and name correspond to Facebook).

Assuming that the (1) OS secret image can be protected from the apps and (2) the entire SA, when displayed, is on the top of any content that can be displayed by an app, the secure annotation becomes hard to forge. A malicious app will not be able to display, say, Facebook's SA since it would have to create a single image with both elements: the OS secret image as well as Facebook's icon and name. The information about Facebook's icon and name is public (so that part can be forged), but the OS secret image is not public information and hence cannot be forged.

Both conditions (1) and (2) in the previous paragraph are necessary conditions for security. Regarding (1), If the OS secret image can be captured by an app, it can obviously forge an SA for any other application. Regarding (2), the attack is more subtle. The malicious app could wait for its secure annotation to come up and then overwrite the left and right portions with another app's icon and name, preserving the middle part. Our implementation is designed to guarantees both conditions, as described in the Implementation section 5.

There are similarities in the above design of secure annotation to public key certificates. One can think of the OS secret image as certifying the remaining part of the secure annotation when they are displayed together as shown.

Secure annotation placement and activation. The placement of the secure annotation affects the usability of the system. One option is to always show the secure annotation near the top or lower portion of the screen. This is similar to web based approaches such as the HTTPS security indicator but had the disadvantage of taking up valuable screen real estate on small form factors. There is also the risk that users might just start ignoring it. Another risk is that full screen apps such as games will be affected as they receive lesser screen space.

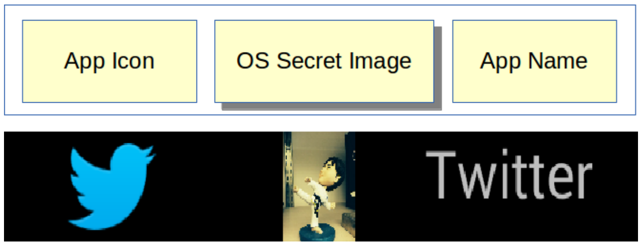

Figure 3: Secure annotation format

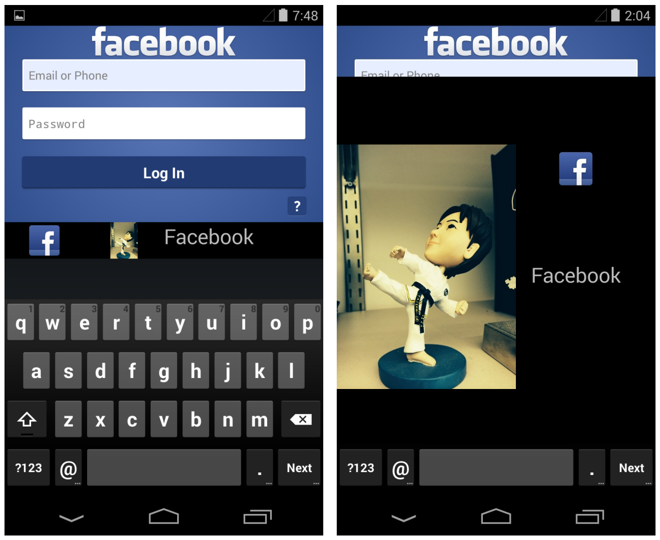

Figure 4: AuthAuth secure annotation for Facebook. The secure annotation is displayed just above the soft keyboard whenever the keyboard is displayed (left picture). If the user wishes to display a full-screen version of secure annotation, a simple tap on the secure annotation does that (right picture). Tapping the full-screen version returns the user back to the left picture. Note: these pictures are at approximately 50% width of real screens.

Our design uses an on-demand approach. Whenever the user is about to perform sensitive input via the soft keyboard, the OS automatically displays the secure annotation just above the keyboard and automatically pans the application UI upwards[2], as shown in Figure 4. Upon tapping the secure annotation, a larger version appears. In this way, we achieve

[2]Panning happens anyway when the keyboard is displayed. We are adding a little more panning to what the keyboard alone requires. Many applications provide a scrollbar or

a middle ground in screen space reservation – the secure annotation above the keyboard can be relatively modest in size, but a full-screen version is available if the user has trouble identifying the contents of the SA clearly.

The display life-cycle of the secure annotation matches the life-cycle of the keyboard. The advantage of this method is that we minimize the number of times a user has to verify the secure annotation contents and only show the secure annotation at the most appropriate time, i.e. when the user is about to perform possibly sensitive data input via a soft keyboard. The number of times the secure annotation is displayed may be further reduced as explained in Section 8.

4.3 The Execution Monitor

As stated in Section 4.1, to prevent time of check time of use errors, we require atomicity once the secure annotation is first rendered for a particular foreground app. Our design utilizes an execution monitor which is a set of hooks at locations crucial to the launch of application code. Considering Android specifically, the execution monitor is invoked when an Activity, Service or Broadcast Receiver is executed. The execution monitor consults the OS internal state to figure out whether the TIVO has been activated. As shown in Figure 5, the TIVO is activated when the SA is displayed as a consequence of the OS bringing up a keyboard. Any navigation within that app is treated as the same session until the user exits the app by pressing the HOME or BACK button or by tapping a notification on the status bar. Note that if a secure annotation is hidden, a TIVO still remains active. When a TIVO is active, the execution monitor converts any background app's attempt to display itself to a notification on the status bar of the phone; this is important to prevent TOCTOU errors and ensure atomicity (see Section 4.1). If the user wishes to launch this app, she will tap the notification and the OS moves the TIVO to an inactive state.

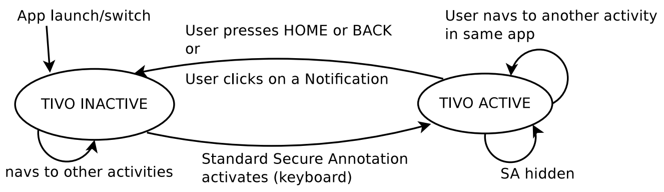
App launch/switch
User presses HOME or BACK or
User clicks on a Notification
User navs to another activity in same app
TIVO INACTIVE
TIVO ACTIVE
navs to other activities
Standard Secure Annotation activates (keyboard)
SA hidden

Figure 5: TIVO state machine

Going back to our Facebook example (Figure 1), Figure 6 shows the interaction. The user launches Facebook and the attacker notices the launch and displays a fake UI. The user then pulls up the keyboard, and alongwith it, a secure annotation. In this instance, the user notices that the OS secret image is valid and then notices a discrepancy in the app name/icon. Therefore, the user chooses to not log in.

One the other hand, if there was no attack, then the app icon/name indicates "Facebook" and the user would have logged in normally. At that point, the TIVO is active (see Figure 5). Now, it is still possible for the attacker to launch a hijacking attack if the user navigates to another activity within Facebook. Since the TIVO is active, the execution monitor notices the new launch attempt and pauses it. Then, a notification appears on the status bar telling the user that an app tried to preempt the UI. The other app's launch is only resumed when the user taps on the notification in the status bar at which point, the TIVO is moved to an inactive state by the OS. Then the OS will render a possibly different secure annotation when the keyboard is brought up again in the newly launched app.

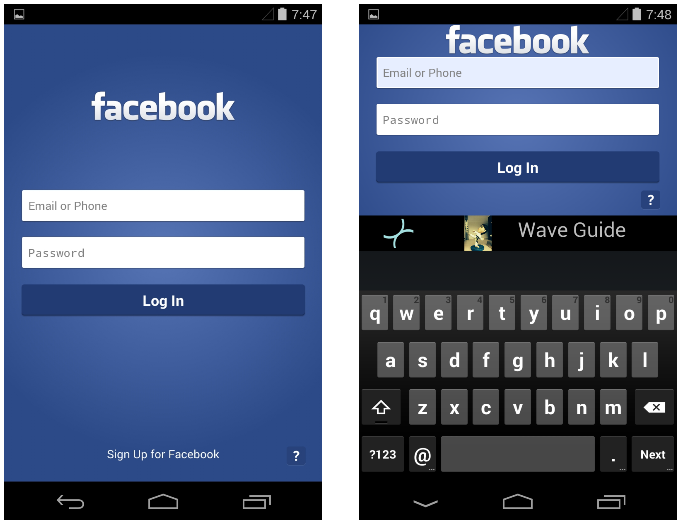

Figure 6: Keyboard-based TIVO in action for the attack scenario that was shown in Figure 1. When the user brings up the keyboard on the hijacked Facebook, secure annotation provides a reliable indicator regarding the application that is going to receive the input.

5. IMPLEMENTATION

We implemented the AuthAuth service for Android 4.4.2 (KitKat). We modified the *InputMethodManagerService* and *WindowManagerService* to notify AuthAuth of the changes in the display. Our implementation is approximately 1200 lines of code. Video demos in [2] show our implementation of AuthAuth and its security properties on a real device.

Figure 7 depicts the implementation architecture. We added a new system service to Android called the *AuthAuthService*. The *InputMethodManagerService* notifies the *AuthAuthService* when a soft keyboard is displayed on the screen. Android uses the Input Method Framework (IMF) and any generic soft keyboard implementation[3] extends the *InputMethodService* and the Android framework calls the methods in the class when it is time to display an input method. AuthAuth hooks at these points and calls into the *AuthAuthService* such that a secure annotation may be generated and rendered.

A modified *WindowManagerService* helps in panning the application UI when it is time to render a secure annotation.

Protection of the secure annotation. We ensure that a

leave sufficient space around the input area to handle the panning. Scaling, along with panning, would have been an alternative design option.

[3]either the system keyboard or a third party keyboard

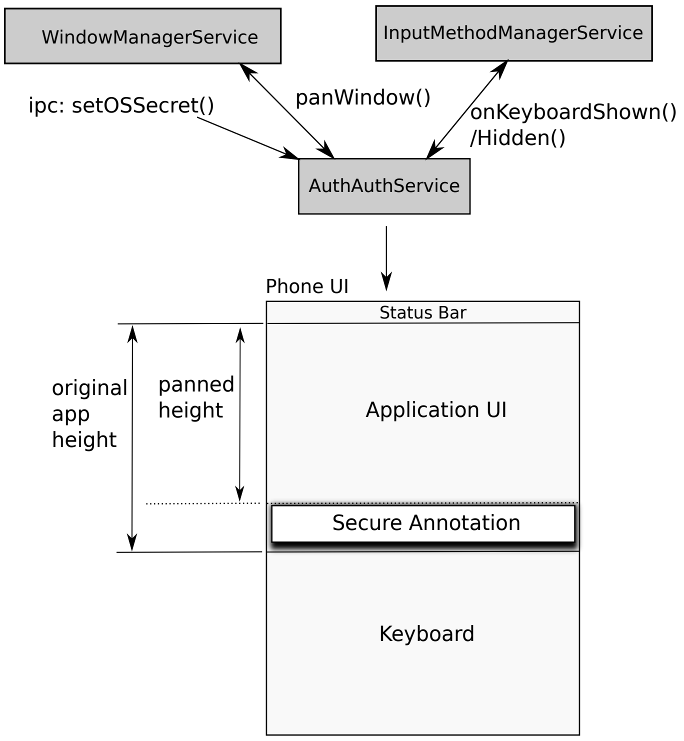

Figure 7: AuthAuth architecture displaying secure annotation when data is entered via a soft keyboard

screenshot of the secure annotation cannot be taken. We create the secure annotation in a separate Layer. A layer is a unit of management of the UI and several layers are composited together by a service called the *SurfaceFlinger*. We set the FLAG_SECURE parameter for the secure annotation's layer, guaranteeing that this layer is removed from screenshots and screen recordings. Even if an app manages to get the screenshot permission [18] (which is a signature only permission[4]), the system will prevent the secure annotation from appearing in a screenshot.

Execution monitor hooks. The execution monitor ensures that any background apps that try to become the foreground app are converted to a notification when the secure annotation is active. To implement this, for Activities and Toasts, hooks inside *ActivityStackSupervisor* notify *AuthAuthService* when an Activity is about to be launched and hooks inside *NotificationManagerService* notify us when a transient notification is about to be shown (Toast). The hooks save the state of the method call (launch or show Activity/-Toast) into a map and post a notification to the status bar. When the notification is tapped, the saved method is executed. We have similar hooks at other places that generate UI for display – For instance, `WindowManager.addView`.

Protecting the OS Secret Image. The OS secret image is stored on the filesystem in a location that is only accessible to the "system" user. As each app on Android runs as a different UID, filesystem permissions prevent unauthorized accesses of the OS secret image. The image is set securely via an IPC from a system app we built. The system app is executed the first time the phone is booted or just after a factory reset during the phone customization process.

[4]Only code signed by the device vendor can obtain signature permissions.

Secure annotation's Z-index: To maintain the invariant that the secure annotation is above any graphical elements created by the app, we added code to reorder secure annotation to be above the apps' display buffers in `SurfaceFlinger::handleMessageRefresh`, which is the rendering cycle of the *SurfaceFlinger*. It is not possible for apps to gain a higher Z-index than the secure annotation because the apps are clients to the *SurfaceFlinger* and hand over their display buffers, which are then composited outside the app's control.

6. PERFORMANCE

Runtime Characteristics. All experiments are carried out on a LG Nexus 4 running a version of Android KitKat modified with TIVO support. AuthAuth, the soft keyboard TIVO, creates and adds a UI element to the view hierarchy of the system whenever a soft keyboard is displayed. We quantified the overhead of the extra code by launching the Messaging and Browser apps alternatively, and pulling up and hiding the soft keyboard. At every step the app process was terminated completely before a new run. We measured the time taken by the code to create and display the secure annotation using the *SystemClock.elapsedRealtime()* method. We averaged the results over 30 runs. The overhead caused by AuthAuth was found to be 11.3ms $\pm$ 2.2ms with 95% confidence. Given that this is only incurred when performing input, interactivity is not affected. The frame rate of graphics-intensive apps such as games are not affected since a keyboard is not used during game play.

App Compatibility. As expected, our keyboard TIVO (AuthAuth) was found to be backward compatible with existing apps. The display of the secure annotation is controlled by a system service and does not require any changes to app-facing APIs. We ran several standard and popular apps (Facebook, Twitter, Skype, Chase, Messaging, Browser, Email, Search, etc.) from the Google Play Store. All of them performed normally.

On a very careful look, we did notice that one app UI (Chase) when panned, was partially obscured by the keyboard (and SA) and did not scroll. Upon further investigation, we found the Chase app to have the same misbehavior on standard Android. We analyzed the decompiled app and found that it did not use the `windowSoftInputMode` XML tag as per Android developer guidelines [3]. Moreover, the other apps such as Facebook and Skype wrapped their layout in a `ScrollView`. We wrapped the decompiled Chase's UI in a `ScrollView` and the UI started scrolling freely.

7. USER STUDY

We conducted a user study to evaluate the effectiveness of TIVOs in enabling users to detect pixel-perfect phishing attacks. We recruited study participants[5] by posting to mailing lists as well as posting fliers. All participants who completed the study were entered into a lottery with a chance

[5]IRB Approval HUM00085708

to win a \$100 Amazon gift card. We received a total of 46 replies out of which 32 fit our requirements that the users own an Android phone. Out of the shortlisted candidates, we asked 3 users to participate in a pilot study to help us refine the study design. Out of the 29 participants, seven participants' data was discarded due to errors in the data collection process. We report on the results of 22 participants for whom data collection was error-free.

Our sample population consisted of 19 males and 3 females. 82% were graduate students, 14% were undergraduate students and 4% were not students. Out of the students, 72% were CS majors. 20 participants had used at least one app from our test set. Two users had not used any apps from our test set before.

Ethics. We obtained signed consent before inviting users to the lab for the study. We reminded users that no personal information would be used or reported. Additionally, we asked users to not use their real user names and passwords if they had accounts for the apps in our test set. We also assured users that if they mistakenly entered their real login information, that data would be immediately destroyed and the experiment be restarted.

7.1 Study Design

We conducted a controlled *within-subjects* experiment wherein each participant was asked to launch four apps (Chase, Skype, Facebook and Twitter) in a sequence and at each stage decide whether they would login to the app or not.

We educated the participants by explaining the nature of pixel-perfect phishing attacks, i.e., some apps could present the exact screens of another app. An example that was given was that a scenario where the user has both a mobile banking app and a game. When the user launches the mobile banking app, the game (malware) replaces the banking app's login screen with an identical-looking login screen and steals the password. The attack was not demonstrated to the users during training but we did tell the users that the attack can occur when launching the genuine versions of the apps. To create baseline data, we then instructed each participant to launch the four apps four times yielding 16 data points on a stock Android phone (without TIVO defenses installed). The data points count how many times users logged into each app. In the background, two out of the four launches of each app was attacked at random by executing an activity hijacking attack, in which another app detected the launch of the genuine app and launched an identical looking UI quickly. This gave us baseline data on the susceptibility of users to fall for a phishing attack on a normal phone, even when they were aware that such attacks could occur.

The second stage measures the ability of our keyboard based TIVO (AuthAuth) to help users in detecting pixel-perfect phishing attacks. We helped each user setup an OS secret image by taking a picture of their choice from the phone's camera. We explained how the AuthAuth mechanism worked using the Messaging app and the Chrome browser app – two apps on which they were not going to be evaluated. Specifically, we launched the messaging app and activated the AuthAuth TIVO by tapping on a text field. We then explained the verification procedure telling the users that they must first verify the central OS secret image. Once that is confirmed to be their own, they must look at the app icon/name and then co-relate that with the UI in the foreground. If the two correspond, then verification is complete, otherwise, it is not and they cannot trust what the foreground app has displayed. There was no attack carried out during this process. The users were then allowed to use the phone with AuthAuth installed on their own without our intervention and free from attacks, trying out any applications they wished. Finally, to collect the data for the second stage, we instructed users to launch the four test apps in a sequence of their choice and attacked two out of four launches of each app at random using activity hijacking from a background app. This gave us comparative data on susceptibility of users to phishing attacks via activity hijacking when AuthAuth is deployed. The second stage of the experiment yields 16 data points corresponding to the 16 data points from the first stage.

We then carried out further experiments in which we evaluated whether users pay attention to the secure OS image, which forms the anchor for verifying the secure annotation. We describe further details of this in Section 7.3.

Subjects took a post study questionnaire (4 questions) at the conclusion of the study, and then we took any questions.

7.2 Result Analysis

Our *null hypothesis* is the number of times users log in to fake and genuine versions of the apps is similar, i.e. AuthAuth does not increase the user's ability to detect the pixel-perfect attacks.

We used a Wilcoxon signed rank test[6] to compare for differences in the number of times users logged in to fake apps on Android (median = 4.5) and a system with AuthAuth (median = 0), with the null hypothesis that the median difference among the pairs is 0. The test showed a significant effect (W = 253, Z = 4.2, $p < 0.001$), thus rejecting the null hypothesis.

Figure 8 summarizes our findings. We conclude that AuthAuth has a favorable and significant effect in steering users away from phishing apps. While 21 out of 22 users had a 100% accuracy (zero fake app logins) in identifying phishing apps using AuthAuth, one user missed attacks on the first two instances and then started giving correct responses subsequently, perhaps figuring out how to use the secure annotation effectively.

Some keen users in our study later reported noticing a slight flickering when the real screen was overlaid by the activity hijacker. When users detected flickering, they were able to take advantage of that to detect an attack even without AuthAuth. This would make the results only more conservative as it would bias the results in favor of the baseline. We ourselves were not aware of the flickering prior to the study. Even users who were looking for flickering were not able to reliably detect all activity hijacking attacks. No users had zero errors in the baseline.

[6]We could not confirm that the data fit a normal distribution based on a histogram and Shapiro-Wilk test.

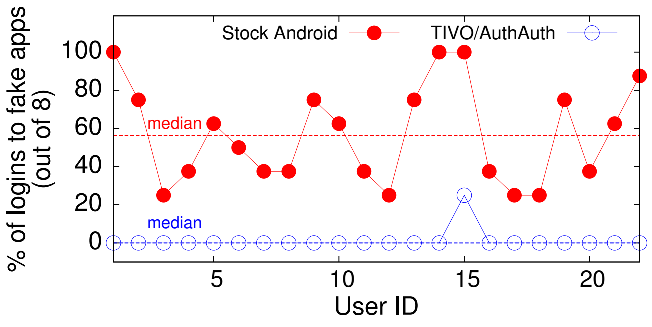

Figure 8: For 8 fake apps presented per user, logins to fake apps in standard Android (out of 8: median = 4.5, mean = 4.5, SD = 2.1) compared with logins to fake apps in TIVO/AuthAuth-based Android (median = 0, mean = 0.1, SD = 0.4) for n = 22 users. *Note:* The graph shows the y-axis as a percentage.

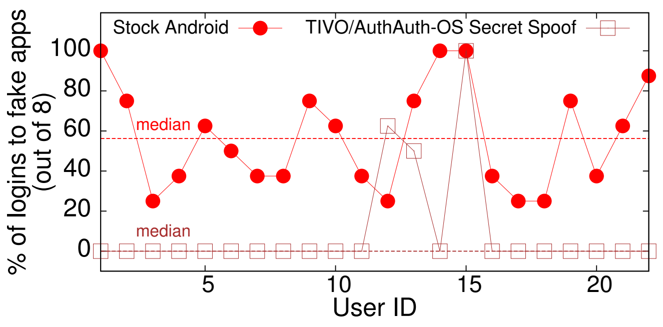

Figure 9: For 8 fake apps presented per user, logins to fake apps in standard Android (out of 8: median = 4.5, mean = 4.5, SD = 2.1) compared with logins to fake apps in AuthAuth-based Android, when the fake app carries out a pixel-perfect attack with a spoofed version of secure annotation (median = 0, mean = 0.8, SD = 2.1) for n = 22 users. *Note:* The graph shows the y-axis as a percentage.

7.3 Spoofing secure annotations

We next evaluated if users could identify a future activity hijacking app that attempts to overcome AuthAuth by not using the system's soft keyboard at all (thus preventing the secure annotation from coming up) and spoofing the secure annotation with its own version. Users have to pay attention to the OS secret image since it forms the trust anchor that enables the user to verify the rest of the information on the secure annotation. We simulated a version of this attack by creating a keyboard UI and a fake secure annotation on the malicious app's window. We used a pre-selected image as the OS secret image. The app identity information was identical to genuine apps. The only difference is the OS secret image.

We instructed users to launch the four test apps two more times in a sequence of their choice while we executed the attack. This gave us an additional 8 data points counting the number of times the users are tricked by the spoofed secure annotation attack. A Wilcoxon signed rank test of the differences in the number of times users logged into fake apps on stock Android (median = 4.5, see Figure 9) and on an AuthAuth system with the OS image attack (median = 0) showed a significant effect ($W = 223$, $Z = 3.8$, $p < 0.001$), rejecting the null hypothesis that the median of the differences was 0. Note that fake apps were shown exactly 8 times to each user on both stock Android and with the spoofed secure annotation.

With AuthAuth deployed, 19 out of 22 users did not log into any of the fake apps under when the secure annotation was spoofed by malware. This is desired behavior because the OS secret image is the trust anchor and users are expected to verify its presence before trusting any other information on the secure annotation. However, two users did not notice the replaced OS secret image at first, each logging into the fake versions 5 and 4 times, respectively, before realizing the change. One user never noticed the change at all and logged into all the fake versions.

User confidence while logging in. We counted the number of times users logged into to the genuine versions of the apps in our test set. On standard Android, 10 users logged into the genuine versions of apps all the time (2 times for each genuine app, giving a count of 8 logins to genuine versions). With AuthAuth-based Android, 20 out of 22 users logged in correctly all the time.

7.4 Post Study Analysis

At the end of the main study, we conducted a post study questionnaire. The first two questions were 5-point Likert scale questions on whether the secure annotation hindered the user's workflow and whether the subjects would use the AuthAuth mechanism if it were available on Android.

For the first question (1 - Strongly Agree that the secure annotation disrupts workflow, 5 - Strongly Disagree), the average response was 2.9 (SD = 1.0) and the most common response was 2. This suggests that most users found the secure annotation to be a slight hindrance. This could be due to users not being to used to using the secure annotations. Interestingly, on the second question (1 - Strongly Disagree and would not use the mechanism, 5 - Strongly Agree and would use the mechanism if available), the average response was 3.7 (SD = 0.8) with the most common response being 4. This means that users are willing to accept the secure annotations in exchange for better security after realizing the damage such attacks cause. We summarize the results of the Likert scale analysis in Figure 10.

We also asked the users what was the most important section of the secure annotation to them when deciding the authenticity of an app. Since the OS secret image forms the trust anchor, users have to learn to verify that image always. As depicted in Figure 11, 10 people preferred to look at all the three indicators, 8 people preferred only the OS secret image and the app name, and one preferred to look at the OS secret image and the app's icon. This shows that most users indeed considered the OS secret image and at least one piece of the app's identifying information. Interestingly, two out of three users who did not notice the spoofed version of secure annotation stated that they paid attention to *only* the app label.

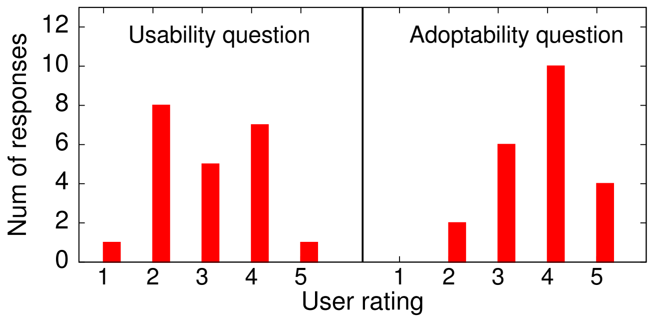

Figure 10: User choices for Usability (left): whether the secure annotation hindered the user's workflow (1 - Strongly Agree, 5 - Strongly Disagree and Adoptability (right): whether the subjects would use the AuthAuth mechanism if it were available on Android (1 - Strongly Disagree, 5 - Strongly Agree).

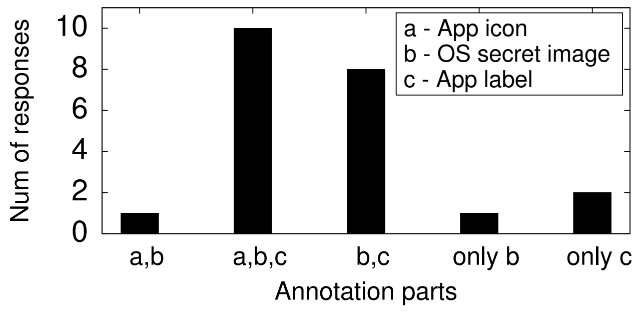

Figure 11: Number of responses for which part of secure annotation is used most.

7.5 Study Limitations

Here we briefly explain the limitations of our study. Some of our users were not familiar with some of the apps that were used in the experiment, though all the apps were reasonably popular. This may have caused the users to trust an app they would have otherwise not trusted even though we gave time to the users to explore the phone and the apps on it before the experiment started.

For privacy reasons, we asked the users to not enter their real passwords and user names. Users might have been more careless in our study since there was no real danger of information theft.

People who volunteered to participate in our study were generally students (undergraduates and graduates) and many of them from computer science. The results could differ if a study were carried out with other populations of users.

We educated the participants on both the nature of pixel-perfect attacks as well as on interpreting a secure annotation. Past work [22] has investigated the effects of security priming, and did not find a significant effect for their population. Nevertheless, without education of participants, results could have been different.

We only had the opportunity to observe each user for a limited time. It is possible that the results would differ if users could be observed using AuthAuth over a longer time horizon. For example, users could become less careful over time or become better at using secure annotations over time.

8. DISCUSSION

TIVOs provide a significant advantage over stock Android in that users can associate a specific app to a phishing attempt. They can then delete the app, thus making repeated phishing attacks by an app more difficult.

8.1 Generalization: non-keyboard TIVOs

This paper primarily focused on automatically creating a trusted input path for soft keyboards, a common mechanism for providing sensitive input such as user ids and passwords. In general, sensitive input may also be in other forms, e.g., selecting privacy-related items via tapping touch screen UI elements. An example is a list of drugs in drug helper app WebMD asking for the user's selection. We implemented a proof-of-concept implementation to show that TIVOs can be generalized to other forms of inputs. To enable the user to confirm the source of the popup box, we introduced a Secure Annotation Key Escape (SAKE), which is a physical key combination. The user presses the Power + Volume Up key, a secure annotation renders on the screen that displays information about the current foreground app and provides the same guarantees for subsequent inputs going to that app, as we provided with soft keyboards. In this instance, we overlaid the secure annotation above the app window. A video demo for SAKE is in [2]. Further investigating the design of such TIVOs, the best way to display their security indicators, and evaluating them with users is future work.

8.2 App-requested TIVO

For some apps, developers may want an OS mechanism that automatically displays a secure annotation all the time in a reserved portion of the application window. We prototyped a TIVO that automatically resizes an application window and renders a secure annotation when the developer requests the OS for it. Note that the developer does not have to explicitly reserve screen space in her app layout. We implemented a preliminary version of selective resizing of app windows by modifying `SurfaceFlinger::doComposeSurfaces()` inside Android's *SurfaceFlinger*. We added code to perform a scaling transformation in the Y direction.

8.3 TIVO improvements

Some users requested that the AuthAuth based TIVO be displayed only for apps of their choosing. This can be done easily by implementing a policy within the *AuthAuthService* and we defer its implementation to future work.

8.4 Limitations of TIVOs

Web site spoofing within the browser app: Our solution does not address the problem of identifying a web site to a user. When using the Google Chrome app, a user could use AuthAuth to ensure that any keyboard input is received by Chrome, and not by another app. But that does not guarantee that Chrome sends the input to the intended web site.

Display icon/name similarity attacks: A malicious app could spoof another app, e.g., Facebook, if it has a similar-looking icon and name as Facebook on the App Store. TIVOs do not prevent such attacks – the burden of distinguishing the real Facebook versus a fake Facebook app on the App Store remains with the user or with the App Store operator. We did ask the 22 users in our study about what their actions would be if they were to notice two similarly-named apps (and possibly with similar icons) on their device. 17 out of 22 users (77%) stated that they would be conservative and uninstall both the apps. Out of the 17, one person gave an interesting response "I would uninstall both apps to which the icons belong and re-install from play store – the one which has highest number of downloads". Four people said that they would find out which is the older one and uninstall the newer one, presumably assuming that the older one was more likely to be authentic.

Privilege escalation attacks: An attacker with **root capability** on the OS will succeed in negating the security guarantees of a TIVO. We note that at this point, all security guarantees made by the OS are nullified.

Side channel attacks: Side channels may exist that reveal contents of the secure annotation to a malicious app, leaking the OS secret image. Our solution does not rule out covert or side channel attacks.

9. RELATED WORK

Security Indicators. The concept of rendering security information on a display dates back to the Compartmented Mode Workstation [6] and Terra [15]. The concept has been applied on the web but designing reliable security indicators has remained a challenge. Nitpicker [13], a kernel based OS window manager, proposes a floating security indicator wherein all windows not in focus are dimmed out and security information is displayed in a reserved area of the in-focus window, possibly covering contents of the window. We achieve a verifiable security indicator without obscuring window contents, without reserving screen space and importantly prevent time of check, time of use errors [16].

Web browsers today render URL information and HTTPS connection status. Users click the HTTPS lock icon to pull up information about certificate of the website and use it to confirm authenticity. Several security toolbars (SpoofGuard [25]) make the current URLs explicit. Unfortunately, this solution is not adequate against pixel-perfect attacks. Nothing prevents an unprivileged application from using the OS windowing primitives to create a full screen web page that also creates a spoofed version of the HTTPS security indicator.

The use of secret images is popular among online banking websites and are known as Site Authentication Images (SAIs). Schechter et al. [22] performed an empirical study on the effectiveness of SAIs and found that 23 of 25 participants decided to log-on in spite of the images being replaced with something else. Furthermore, an SAI is susceptible to man-in-the-middle attacks [23] because all that is required to retrieve an SAI is the user name and sometimes response to a challenge question. Our solution avoids these pitfalls of SAIs that are used on the web, taking advantage of the different context of use. We incorporate the recommendations of [9] by using personalized images as the OS secret image instead of standard graphics or predefined text strings.

Akhawe et al. [5] measured the click-through (of field data) rates of active security indicators such as the SSL warning messages and phishing warnings. The study found that click-through rates for phishing warnings were between 9.1% and 18.0% for Mozilla Firefox and Google Chrome respectively. Other studies [30, 22, 10] have made similar measurements of warning messages. The TIVO mechanism is not a warning mechanism but a reliable information tool. It is active only when the system detects that a user is about to perform input of data and it always shows up on keyboard input, unless the app is under attack, providing a consistent experience.

Security indicators on smartphones. Reserving screen space [11, 17] at the OS level is one method of securely displaying information about the UI currently in the foreground. Crossover [17] reserves a portion of the screen space at the framebuffer level to display the currently active virtual machine in a multi-VM phone. This solution permanently reduces available screen real estate as well as risks users missing the information in the reserved portion because its always there.

Password theft prevention. GuarDroid [27] uses a trusted system keyboard to encrypt any passwords typed. The system asks users to set a "secure string" at boot time by selecting words from a predefined set. The purpose is to prove to the user that the soft keyboard is indeed system generated. GuarDroid then intercepts outbound network traffic and rewrites packets to replace the encrypted form of the password with the decrypted form. ScreenPass [20] is a similar system that uses OCR to determine whether a keyboard is rendered on screen and then verifies that they keyboard is indeed the trusted keyboard. ScreenPass then performs heavyweight taint tracking to ensure that passwords are not leaked. In contrast, TIVOs do not require OCR or taint tracking and are not restricted to password data.

Trusted Path. Historically, Linux and Windows have included support for a session login trusted path which is a key combination that brings up a trusted login dialog. On Linux it is called the Secure Attention Key [19] and on Windows its the Ctrl-Alt-Del combination. TIVOs are not restricted to login prompts, and function with any app window requiring user input. Zhou et al. [32] address a different threat model and aim to provide trusted communication between devices and a process when the OS is compromised. They require a trusted hypervisor and did not discuss phishing and pixel-perfect attacks.

10. CONCLUSION

We introduced an operating system abstraction called Trusted Visual I/O Paths (TIVOs) that enables a user to securely verify the app she is interacting with and described an implementation for Android. The TIVO in the Android implementation is activated any time a soft keyboard is used by an application (e.g., for password entry) so that the user can reliably determine the app that will be receiving the user's keyboard input. The TIVO is designed to be a tamper-resistant, difficult-to-forge security indicator and does not require a reserved area on the screen. In a controlled user study, the soft keyboard TIVO was found to make a significant improvement in users being able to identify pixel-perfect attacks. We also describe other forms of TIVOs that can be launched on demand by users or by apps.

11. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0916126 and 1318722 Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

12. REFERENCES

[1] Android Touch-Event Hijacking. `https://blog.lookout.com/blog/2010/12/09/android-touch-event-hijacking`.

[2] AuthAuth Video Demo. `http://sites.google.com/site/authauthdemo/`.

[3] Handling Input Method Visibility. Android Authors. `http://developer.android.com/training/keyboard-input/visibility.html`.

[4] Activity hijacking pattern for Android. `http://capec.mitre.org/data/definitions/501.html`.

[5] Akhawe, D., and Felt, A. P. Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 257–272.

[6] Berger, J. L., Picciotto, J., Woodward, J. P. L., and Cummings, P. T. Compartmented mode workstation: Prototype highlights. *IEEE Trans. Softw. Eng. 16*, 6 (June 1990), 608–618.

[7] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into Your App without Actually Seeing it: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[8] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.

[9] Dhamija, R., and Tygar, J. D. The Battle Against Phishing: Dynamic Security Skins. In *Proceedings of the 2005 Symposium on Usable Privacy and Security* (New York, NY, USA, 2005), SOUPS '05, ACM, pp. 77–88.

[10] Dhamija, R., Tygar, J. D., and Hearst, M. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2006), CHI '06, ACM, pp. 581–590.

[11] Felt, A. P., and Wagner, D. Phishing on Mobile Devices. In *In W2SP* (2011).

[12] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 22–22.

[13] Feske, N., and Helmuth, C. A Nitpicker's Guide to a Minimal-complexity Secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), ACSAC '05, IEEE Computer Society, pp. 85–94.

[14] Foley, J., van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principle and Practice.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[15] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 193–206.

[16] Huang, L.-S., Moshchuk, A., Wang, H. J., Schechter, S., and Jackson, C. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 22–22.

[17] Lange, M., and Liebergeld, S. Crossover: Secure and Usable User Interface for Mobile Devices with Multiple Isolated OS Personalities. In *Proceedings of the 29th Annual Computer Security Applications*

Conference (New York, NY, USA, 2013), ACSAC '13, ACM, pp. 249–257.

[18] Lin, C.-C., Li, H., Zhou, X., and Wang, X. Screenmilker: How to Milk Your Android Screen for Secrets. In *NDSS* (2014).

[19] Linux Secure Attention Key. `https://www.kernel.org/doc/Documentation/SAK.txt`.

[20] Liu, D., Cuervo, E., Pistol, V., Scudellari, R., and Cox, L. P. ScreenPass: Secure Password Entry on Touchscreen Devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 291–304.

[21] Niemietz, M., and Schwenk, J. UI Redressing Attacks on Android Devices. In *Proceedings of BlackHat Abu Dhabi.)* (2012).

[22] Schechter, S. E., Dhamija, R., Ozment, A., and Fischer, I. The Emperor's New Security Indicators. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), SP '07, IEEE Computer Society, pp. 51–65.

[23] Fraud Vulnerabilities in SiteKey Security at Bank of America. Jim Youll. `http://www.cr-labs.com/publications/SiteKey-20060718.pdf`.

[24] Phishing Attack replaces Android banking apps with malware. `http://blogs.mcafee.com/mcafee-labs/phishing-attack-replaces-android-banking-apps-with-malware`.

[25] SpoofGuard. `http://crypto.stanford.edu/SpoofGuard/`.

[26] Swype Keyboard for Android. `https://play.google.com/store/apps/details?id=com.nuance.swype.dtc`.

[27] Tong, T., and Evans, D. GuarDroid: A Trusted Path for Password Entry. In *Proceedings of Mobile Security Technologies (MoST)* (2013).

[28] Wang, R., Xing, L., Wang, X., and Chen, S. Unauthorized Origin Crossing on Mobile Platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 635–646.

[29] Watson, R. N. M. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies* (Berkeley, CA, USA, 2007), WOOT '07, USENIX Association, pp. 2:1–2:8.

[30] Wu, M., Miller, R. C., and Garfinkel, S. L. Do Security Toolbars Actually Prevent Phishing Attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2006), CHI '06, ACM, pp. 601–610.

[31] Xu, Z., and Zhu, S. Abusing Notification Services on Smartphones for Phishing and Spamming. In *WOOT* (2012), pp. 1–11.

[32] Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 616–630.