Real-time detection and reaction to Activity hijacking attacks in Android smartphones

Anis Bkakria[1], Mariem Graa[1], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1] and Jean-Louis Lanet[2]
[1] IMT Atlantique, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
Email: {anis.bkakria, mariem.graa, nora.cuppens, frederic.cuppens}@imt-atlantique.fr
[2] Campus de beaulieu, 263 Avenue Général Leclerc, 35042 Rennes - France
Email: jean-louis.lanet@inria.fr

***Abstract*—Most Android users are required to communicate sensitive data (passwords, usernames, security codes, and credit card numbers) with applications. Hacker can launch phishing attacks to compromise user data confidentiality. He/She stealthily injects into the foreground a hijacking Activity at the right timing to acquire private information. In this paper, we propose an effective approach that uses the similarity between launched Activities in order to detect and reacts to hijacking attacks during runtime time. We demonstrate the effectiveness of our solution by quantifying the number of false positives that can be generated by our system. We observe that, in the worst case, our solution generates 4.2% of false positives and incurs only 0.39% performance overhead on a CPU-bound micro-benchmark.**

I. Introduction

Mobile technology has become integrated into nearly every aspect of our lives. In particular, smartphones are widely used for performing numerous important life Activities such as communication, shopping, shifting (GPS navigation), banking and web browsing. In terms of the smartphone operating system market, Android increased 16.6 % in the fourth quarter of 2015, to account for 80.7 percent of the global total [1]. In order to satisfy Android users Activities, the development of Android applications have been growing at a high rate. In May 2016, 65 billion apps had been downloaded from Google Play [2]. Most Android users Activities require to communicate sensitive data (passwords, usernames, security codes, and credit card numbers) with the application. Hacker can launch phishing attacks to compromise user confidentiality. He creates a similar application user interface (UI) to acquire private information. Phishing takes advantage of the user trust that this interface is real. Therefore, the hacker has the chance to gain the personal information of the targeted user. For example, Chen *et al.* [3] stealthily inject into the foreground a phishing Activity at the right timing and steal sensitive information a user enters. Phishing attacks [4] use a fake UI indistinguishable from the target user interface to steal a password or payment credentials data. We implement the Activity hijacking attack proposed in [3] and configure it to hijack the credit card payment Activity used by Google Play. A short video describing the attack is accessible at **https://youtu.be/q60Nhwgar0Y** [5] . Then, to evaluate the attack, we asked 50 Android users to use an Android device on which our attack app is installed to buy an item from Google Play without using their real credit card information. All the users who conducted the experiment did not realize that they were under a phishing attack. Many security techniques [6], [7] are used to detect phishing attacks in Android systems. They are based on analyzing application resource files (XML layout) to detect similarity of UI (fake and target one). However, the layout files can be obfuscated by an attacker without changing UI appearance. In addition, the interface component parameters can be modified in the java application code and the layout files remain unchanged. Thus, attackers can evade this detection mechanisms. In this paper, we modify the Android OS to detect phishing attacks that exploit visual similarity. We extract UI components parameters at application run time by instrumenting the Android operating system code. Then, we define a filter that compares similarity of the UIs using these parameters. To make our solution more robust, we use a taint mechanism that associates taint to UI components and we detect leakage of these information when they are sent to the attacker through the network. The rest of this paper is organized as follows: Section 2 categorizes application phishing attacks and analyzes possible countermeasures. Section 3 describes the proposed approach. Section 4 provides implementation details. Finally, section 5 concludes with an outline of future work.

II. Related work

In this section, we categorize application hijacking attacks in Android systems. We also discuss existing countermeasures.

A. Hijacking Attacks

Similarity attack [8],[9],[10] exploits the similarity between the legitimate and the phishing application to mislead the user. It uses a similar or identical name, icon, and UI.
Background attack [11],[4] exploits the Android ActivityManager, or a side-channel [12] to detect running of a legitimate application. This attack runs in the background and when a user app is launched, it moves to the foreground and displays a phishing screen.
The hijacking attacks [3] include similarity and background attacks. In this class of attacks, a malicious Activity is launched instead of the intended Activity. The hijacker can spoof the victim Activity's UI to steal user sensitive data. As the Android UI does not identify the currently running application, hijacking attacks can be implemented convincingly.

B. Hijacking Countermeasures

Code Analysis: Biahchi *et al.* [11] use static analysis to identify and categorize a variety of attack vectors that allow a malicious app to mimic the GUI of other apps and launch hijacking attacks. Also, they designed and implemented an on-device defense that informs users about the origin of the app with which they are interacting. The Google Bouncer system [14] uses patterns of system calls and permissions to detect phishing attacks. These signature-based malware detection techniques can not detect phishing attacks (similarity attacks) that do not require a specific permission such as the attack considered in this paper.

Personalized indicator: Marforio *et al.* [15] propose a personalized indicator to mitigate application phishing attacks in mobile platforms. This approach is based on checking the presence of the correct indicator. At install time, the user attaches an image to the application. At run time, when the user enters his private data, the application shows the image chosen at install time. A phishing attack is detected if the image is not displayed. Other applications cannot read the indicator since it is stored in application-specific storage. The drawback of this approach is that it requires extra user effort at install time and during application usage.

Visual similarity: Malisa *et al.*. [6] extract and compare the visual similarity of login interface screenshots. In order to do so, they fix a percentage (deception rate) of users that would confuse the examined screenshot with the reference application. Thus, these users are estimated to consider the hijacking app genuine. Malisa *et al.* approach incurs a significant runtime overhead. Sun *et al.* [7] propose the DroidEagle tool that detects similar Android apps by comparing visual characteristics and components in the layout resources. DroidEagle is used to identify repackaged apps and phishing malware. Sun *et al.* approach cannot detect obfuscation of layout files because UI appearance remains unchanged. Thus, attackers can evade this detection mechanisms. In this paper, we use visual similarity approach to detect hijacking attacks. Our approach extracts UI components parameters at application run time by instrumenting the Android operating system code. Therefore, we can detect obfuscation of layout files.

III. Our Approach

In this section, we describe the threat model we consider and we present our detection and reaction models.

A. Threat Model

First, we assume an attacker that controls an application running in the background on the victim device. In addition, we suppose that the attack application can use any existing Android permission to collect information about running applications in order to figure out which Activity is entering the foreground. Second, in order to overcome similarity detection by comparing visual characteristics and components in the layout resources [7], we assume that the attacker can create the hijacking screen dynamically (e.g., by creating UI elements at runtime). Third, in order to avoid the detection of exact similarity between the hijacked and the hijacking Activities, we assume an attacker that can change the visual characteristics (e.g., size, position, colour) of the UI elements in the hijacking Activity by exploiting the properties of human UI interfaces perception. Finally, we assume that the Android OS is not compromised by the attacker.

B. Detection Model

By knowing the moment in which a legitimate Activity is launched, the attacker exploits its plotting time to launch a similar Activity. We present the following definitions to compare UIs visual similarity and to detect Activity hijacking attacks.

Definition 1: **(plotting time).** Given a device D and an Activity A, a plotting time $\mathcal{P}_A^D$ represents the time needed for the device D to plot the Activity A on its screen.

The previous definition states that a plotting time $\mathcal{P}_A^D$ for a device *D* represents the elapsed time between an action that triggers the launch of an Activity *A* and the moment in which *A* appears on *D*'s screen.

Definition 2: **(k-size indistinguishability).** Given a device D having a screen size $s_D = (w_D, h_D)$ and two Activities A_1 and A_2 and their respective sizes $s_{A_1} = (w_{A_1}, h_{A_1})$ and $s_{A_2} = (w_{A_2}, h_{A_2})$. A_1 and A_2 are k-size indistinguishable on D iff the following condition holds:

$$1 - \frac{\|s_{A_1}, s_{A_2}\|}{\sqrt{w_D^2 + h_D^2}} = k$$

where $\|a, b\|$ denotes the euclidean distance between a and b.

On the same device, each activity has its own size that cannot be larger than the size of the device screen. Therefore, $k \in [0, 1]$. Actually, in the previous definition, k represents, for an Android user, the difficulty that he/she can distinguish A_1 and A_2 from their sizes. The more k will be close to 1, the more it is difficult to distinguish A_1 and A_2 based on their sizes. In fact, each Android Activity is composed of different type of user interface elements (e.g., TextView, TextEdit, Button, Spinner, etc.). In the reminder of this paper, we use $E = \{e_1, e_2, \cdots, e_n\}$ to denote all types of user interface elements that can be used in an Android Activity.

Definition 3: **(l-content indistinguishability).** Given a device D and two Activities A_1 and A_2. Let us suppose that the contents of A_1 and A_2 are represented respectively by $\mathcal{C}_{A_1} = \{e_1^{A_1}, e_2^{A_1}, \cdots, e_n^{A_1}\}$ and $\mathcal{C}_{A_2} = \{e_1^{A_2}, e_2^{A_2}, \cdots, e_n^{A_2}\}$ where each $e_j^{A_i}$ represents the set of user interface elements of type e_j that are contained in A_i. A_1 and A_2 are l-content indistinguishable on D iff the following condition holds:

$$\sum_{\substack{i=0 \\ e_i^{A_1} \neq \emptyset}} \left(\frac{\sum\limits_{c \in e_i^{A_1}} \left(\max\limits_{c' \in e_i^{A_2}} indist(c, c', D) \right)}{\mid e_i^{A_1} \mid \times \sum\limits_{\substack{i=0 \\ e_i^{A_1} \neq \emptyset}} 1} \right) = l$$

where $\mid e_i^{A_1} \mid$ denotes the cardinality of $e_i^{A_1}$, $indist(c, c', D) \in [0, 1]$ represents the level of indistinguishability between c and c' on D, and $l \in [0, 1]$.

In fact, to compute the indistinguishability of two user interface elements on a device, for each type of user interface, we use the size, the position and other specific visual attributes (e.g., text, colour, etc.).

Definition 4: **(visual attribute).** Given an attribute a of a UI element c, a is a visual attribute of c iff the modification of the value of a implies a modification of the appearance of c on any device screen.

Definition 5: Given a device D having a screen size $s_D = (w_D, h_D)$ and two UI elements c_1 and c_2 of the same type e. Indistinguishability is a value between 0 and 1 that has a component involving difference in size, difference in position, and the average similarity of all the components' visual attributes $a_1, \cdots, a_n$. The similarity of c_1 and c_2 in D is computed as following:

$$indist(c_1, c_2, D) = 1 - \frac{\|s_{c_1}, s_{c_2}\|^* \ + \ \|p_{c_1}, p_{c_2}\|^*}{2} + \sum_{i=1}^{n} \frac{sim(a_i^{c_1}, a_i^{c_2})}{n}$$

where:

- $s_{c_1}, s_{c_2}, p_{c_1}, p_{c_2}$ represent respectively the sizes and the positions of c_1, c_2 on D's screen.
- $\|a,b\|^* = \left\{ \begin{array}{ll} 1 & if \quad \frac{\|a,b\|}{\sqrt{w_D^2+h_D^2}} > 1 \quad (i) \\ \frac{\|a,b\|}{\sqrt{w_D^2+h_D^2}} & otherwise \end{array} \right\}$
- $sim : \mathcal{D} \rightarrow [0,1]$ such that :

$$\forall x_1, x_2 \in \mathcal{D}, \forall y, y' \in [0,1] : (sim(x_1, x_2) = y \wedge sim(x_1, x_2) = y') \rightarrow y = y'$$

In Android Activity, one can use *scrollview* to represent a content that is bigger than the size of the device screen. As a consequence, the euclidean distance between the sizes (resp. positions) of two UI elements can be bigger than the device screen. Therefore we use *(i)* to ensure that if the euclidean distance between the sizes (resp. positions) of two UI elements is bigger than the device screen size, the distinguishably of the two UI elements in terms of size (resp. position) is full.

Definition 6: Given two Android Activities A_1 and A_2 and an Android user u. A_1 and A_2 are said to be (k,l) indistinguishable to u iff the following conditions hold:

- A_1 and A_2 are k-size indistinguishable
- A_1 and A_2 are l-content indistinguishable
- u cannot distinguish A_1 from A_2.

We conducted a study to select the Activity indistinguishability threshold for a user (see Section IV-D). Based on previous definitions, the detection of an Activity hijacking attack is defined as following.

Definition 7: **(Activity hijacking attack).** Given a device D, a user u, two Android Activities A_1 and A_2 launched on D at t_{A_1} and t_{A_2} by two processes p_{A_1} and p_{A_2}. Let us suppose that the plotting time of A_1 is $\mathcal{P}_{A_1}^D$. The Activity A_1 is successfully hijacked by A_2 iff the following conditions hold:

1) $p_{A_1} \neq p_{A_2}$
2) $t_{A_2} - t_{A_1} < \mathcal{P}_{A_1}^D$
3) $\exists k, l \in [0,1]$ such that A_1 and A_2 are (k,l) indistinguishable to u

The hacker must launch hijacking activity before the appearance of the victim activity in the screen (before the expiration of the plotting time $\mathcal{P}_{A_1}^D$ of $A1$). If he/she launchs the activity after, the user can observe the two activities (abnormal behavior).

C. Reaction Model

Our reaction model for Activity hijacking attacks is mainly based on the indistinguishability level between the attack and legitimate Activities. Two cases are considered.

1) Full indistinguishability: In our detection model, full indistinguishability between an attack and a legitimate Activities occurs when they are *1-size* and *1-content indistinguishable*. In this case, we react by performing two actions:

- Before entering the foreground, the attack Activity is blocked to prevent its usage by the user.
- The Android user is notified that a hijacking attack was detected and blocked. The notification contains the *pid* and the *name* of the attack application.

2) Partial indistinguishability: Partial indistinguishability between an attack and a legitimate Activities occurs when there exists $k \in]0,1[$ and $l \in]0,1[$ such that the two Activities are *k-size* and *l-content indistinguishable*. The reaction to this class of hijacking attacks is performed as following:

- In parallel with the entering of the attack Activity to the foreground, we notify the user that he/she is probably under a hijacking attack. We recommend him/her, before entering any input, to check whether a similar Activity (legitimate Activity) exists in the background.
- We associate taint to UI components. Thus, the private data entered by user will be tainted. We track propagation of tainted data in the Android system to detect leakage of these information. A notification appears when the private data are sent to the attacker through the network.

IV. Implementation

We have implemented our proposed approach in TaintDroid operating system [16]. To do so, first we instrumented *Window Manager*, *Activity Manager* and *View System* components of the Android os framework layer. Then, we developed and added *Activity Hijacking Protector* – a framework layer system service responsible for detecting and reacting to Activity hijacking attacks. The following sections explain the implementation of the components that are used to detect and react to Activity hijacking attacks.

A. Activity plotting time

Activity plotting time is a key parameter of our detection model, it depends mainly on the duration of the operations that need to be performed by both *onCreate()* and *onResume()* which, in many cases, depend on unknown parameters. For instance, the contents of some UI elements may have to be

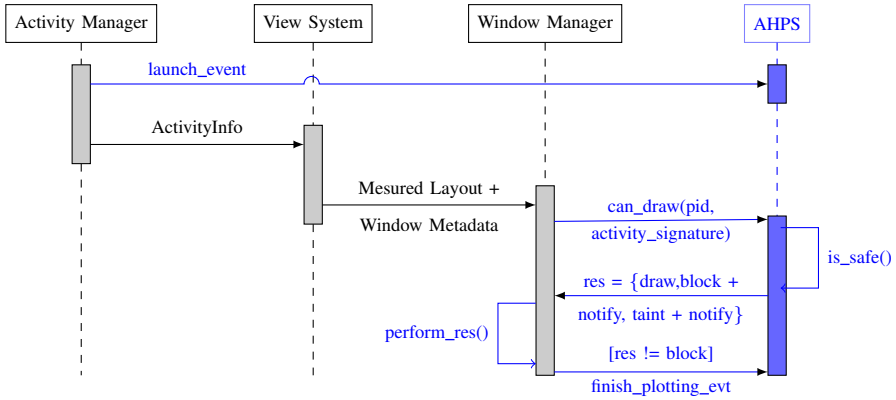


Fig. 1. AHPS interactions during Activity launching process

retrieved from a server, which makes the plotting time of the Activity depending on the time needed by the server to send those contents. Consequently, formally identifying the plotting time of an arbitrary Activity before or during its launching is not possible. Actually, our approach can still be valid if we can (1) figure out the launching time of an Activity and (2) at any instant, know whether the plotting of the Activity is finished. To meet (1), we instrumented the *Activity Manager* component by sending a *launch_evt* to the *Activity Hijacking Protector* at the beginning of *handleLaunchActivity()*(ActivityThread.java). To ensure (2), we instrumented the *Window Manager* service by sending, at the end of *finishDrawingWindow()*(WindowManagerService.java), a *finish_plotting_evt* to the *Activity Hijacking Protector*.

B. Activity UI Signature extraction

To get the UI elements features of the Activity to be launched, we instrumented the *View System* component by adding, in each UI element class (e.g., TextView.java, etc.), a function that collects the values of the UI element visual attributes. To compute the UI signature of the Activity, we create a function (*contructSignature()*) that traverses the UI layout tree of the Activity, collects all the leaf nodes in the layout tree, and extracts the features that dominate the UI visual appearance. The *contructSignature()* function is called at the beginning of *performDraw()*(ViewRootImpl.java).

```
1  Procedure handle_launch_evt (activity_name)
2      pid = Binder.getCallingPid()
3      add_to_launched_activity(pid, activity_name)
4      return
5  Procedure handle_finish_plotting_evt (activity_pid,
     activity_name)
6      pid = Binder.getCallingPid()
7      if pid! = WINDOW_MANAGER_PID then
8          return
9      end
10     remove_from_launched_activity(activiy_pid, activity_name)
11     return
```

Algorithm 1: Handling $launch\_event$ and $finish\_plotting\_evt$ events by (AHPS)

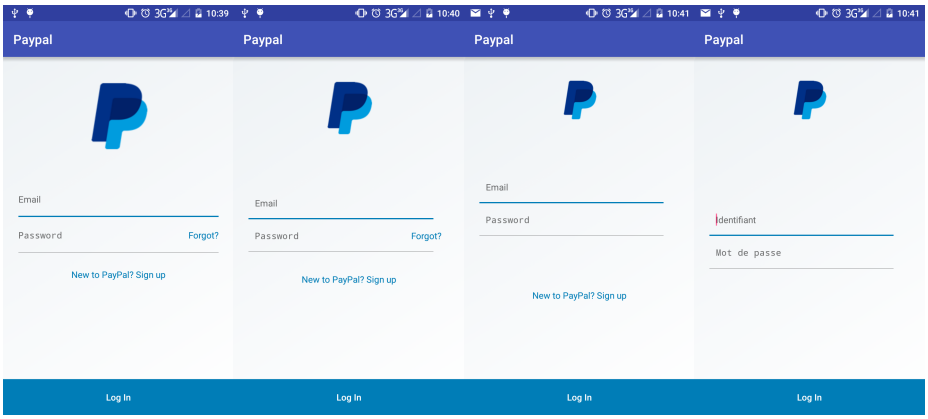
C. Activity Hijacking Protection Service (AHPS)

AHPS represents the main component in the implementation of our approach. Figure 1 illustrates the sequence of exchanged information and events between *Activity Manager*, *View System*, *Window Manager*, and *AHPS* during the launch of an Activity. The modifications we introduced to the initial process of Activity launching are colored in blue.

```
1  Procedure can_draw()
2      pid = Binder.getCallingPid()
3      result = AHPS.is_safe(pid, activity_Signature)
4      perform_res(result)
5      AHPS.handle_finish_plotting_evt(pid,
         activity_Signature.get_name())
6      return
7  Procedure perform_res(result)
8      if result = DRAW then
9          finishDrawingWindow()
10     end
11     if result = TAINT then
12         taint(activity_ui_objects)
13         notify_user("You are probably under an
             attack!")
14         finishDrawingWindow()
15     end
16     if result = BLOCK then
17         activity.finish() /* stop the activity thread */
18         notify_user("An attack from 'pid' is blocked")
19     end
20     return
```

Algorithm 2: Requesting the permission to draw an activity

As presented in Algorithm 1, *AHPS* handles the two events *launch_event* and *finish_plotting_evt* that are sent respectively from the *Activity Manager* component and the *Window Manager* service. Based on those two events, *AHPS* will be able of know, at any instant, (i) if there exists one or more Activities that are launched, but not yet plotted. When a new Activity is going to be plotted by the *Window Manager*, this last queries *AHPS* for the permission to plot the Activity (Algorithm 2 line 3). The permission is computed by the *AHPS*, in Algorithm 3, based on (i), the *pid* that is launching the Activity, the Activity signature, and the user Activity distinguishability threshold



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 2. The appearance of Paypal login Activity in the 4 steps of our experiment app

```
1  Procedure is_safe(pid, act_sig)
2      k,l = 0, safe = true
3      foreach (prev_pid, activity_name) in
         launched_activity do
4          if pid != prev_pid then
5              safe = false
6              prev_act_sig = get_signature(prev_pid)
7              (k', l') =
                 compute_similarity(prev_act_sig,act_sig)
8              if k' + l' > k + l then
9                  (k, l) = (k', l')
10             end
11         end
12     endfch
13     if safe then
14         return DRAW
15     end
16     if k = 1 and l = 1 then
17         return BLOCK
18     end
19     if (k, l) > user_distinguishability_threshold then
20         return TAINT
21     end
22     return
```

Algorithm 3: Computing Activity plotting permission

(Section IV-D). Three permission values can be returned from *AHPS* as a response to the query of the *Window Manager*. The response *"DRAW"* means that the *Window Manager* is authorized to plot the Activity since no attack has been detected from *AHPS*. The response *TAINT* means that the Activity that is going to be launched is probably an Activity hijacking attack. In this case, the *Window Manager* taints all the UI objects contained in the Activity, notifies the user about a probable Activity hijacking attack, and plot the Activity (Algorithm 2 lines 12 to 14). Finally, the response *BLOCK* means that *AHPS* is almost sure that the Activity that is going to be launched is an Activity hijacking attack. In this case, the *Window Manager* blocks the plotting of the Activity and notifies the use (Algorithm 2 lines 17 and 18).

D. Activity indistinguishability threshold selection

It has been demonstrated in [17], [18] that user's perception of an interface can be influenced by many factors (e.g, gender, culture, age, etc.). However, the results of those studies are too generic to be used to select an Activity indistinguishability threshold for a user. We conducted a study on the distinguishability perception of mobile app interfaces. The study is performed as an Android app composed of four steps. In the first step, the participant is asked to use a set of Activities belonging to different apps (e.g., Paypal login Activity, Google Play credit card paying Activity, etc.). The purpose of this step is to help the participant to become familiar with the used Activities. Then, for each subsequent step, we change the order in which the Activities is shown to the participant. Moreover, for each Activity, we increasingly modify its appearance (i.e., change the size, position, visual attributes of UI elements) compared to the one shown during the first step. Fig. 2 shows, in each step, the appearance of Paypal login Activity shown to the user. The picture (a) of the Fig.2 represents a slightly different Paypal login activity in which we imperceptibly reduce the size of *EditText*s compared to the original Paypal login activity. In the picture (b) of Fig.2, we increase the appearance difference by reducing the size of the Paypal icon used in the activity. In the step 3 in Fig.2, we remove the *TextView* containing the "forgot?" link and we decrease more the size of the Paypal icon. Finally, in the step 4, we remove the *TextView* containing the "New to PayPal? Sign up" link and we modify "TextEdit"s' hints. The purpose of the developed app is to know in which step the participant will be able to distinguish the modified Activity from the original one (Step 1). This knowledge allows us to select the Activity indistinguishability threshold (*ind_thld*) for the participant as following:

$$ind\_thld = \frac{k+l}{2}$$

Where k and l are respectively the size and content indistinguishability between the original Activity and the distin-

	Gender		Age						Usage of mobile payment	
	Male	Female	≤ 20	[20-30]	[30-40]	[40-50]	[50-60]	≥ 60	yes	no
Percentage of participants	60,6 %	39,4 %	10,1 %	23,5 %	29,2 %	16,8 %	14,6 %	5,7 %	22,4 %	77,5 %
w_ind_thld	0,48	0,45	0,51	0,55	0,54	0,5	0,48	0,45	0.6	0.45

TABLE I
CLASSIFICATION OF PARTICIPANTS THAT COMPLETED OUR EXPERIMENT

guish one. Our study was conducted in a public square in Rennes (France). We randomly select 89 participants for our experiment. Table I classifies them based on their information (i.e., gender, age, and the usage of mobile payment methods) collected before the usage of our experiment app. In addition, it provides the percentage of participants belonging to each class and shows the worst activity indistinguishability threshold (*w_ind_thld*) for each class (i.e., the lowest level of distinguishability that allows all members of a class of participants to distinguish two activity based on their appearance on a device screen). The usage of the minimum *w_ind_thld* as the Activity indistinguishability threshold for Android users allows to reduce to the best the number of false negatives. However, it also increases the number of generated false positives. To overcome this limitation, we classify the user according to their gender, age, and their usage of mobile payment. Then, the user' Activity indistinguishability threshold is computed as the maximum *w_ind_thld* of the classes in which the user belongs.

We evaluate the effectiveness of our proposed solution. Our approach generates 4.2% of false positives in the case of partial indistinguishability and $10^{-3}\%$ in the case of full indistinguishability. In addition, we evaluate the performance of our approach. We found that our solution incurs only 0.39% performance overhead on a CPU-bound micro-benchmark. Due to space limitations, the evaluation details has not been included.

V. CONCLUSION

In this paper, we propose a solution to detect and prevent Activity hijacking attack based on the indistinguishability level between the attack and legitimate Activities. To do so, we modify the Android system to extract and compare UI elements features of the legitimate and the phishing interfaces. We implement an Activity hijacking protection service that allows plotting Activities based on the pid that launched the Activity, the Activity signature, and the user Activity distinguishability threshold. We define a reaction mechanism that blocks the attack Activity before entering the foreground to prevent it to be used in the case of full indistinguishability. In the case of partial indistinguishability, we notify Android user and we associate taint to sensitive data to detect leakage of these information through the network. Our approach generates 4.2% of false positives in the worst case and does not really affect the performance of the system (0.39% overhead).

REFERENCES

[1] Egham, "Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015," 2016, http://www.gartner.com/newsroom/id/3215217.

[2] "Cumulative number of apps downloaded from the google play as of may 2016 (in billions)," http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

[3] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 1037–1052.

[4] A. P. Felt and D. Wagner, *Phishing on mobile devices*. na, 2011.

[5] "Video demo for activity hijacking attack on google play," http://conferences.telecom-bretagne.eu/fps/as/.

[6] L. Malisa, K. Kostiainen, and S. Capkun, "Detecting mobile application spoofing attacks by leveraging user visual similarity perception."

[7] M. Sun, M. Li, and J. Lui, "Droideagle: seamless detection of visually similar android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 9.

[8] D. Trends, "Do not use imessage chat for android, it's not safe," 2013, http://www.digitaltrends.com/mobile/imessage-chat-android-security-flaw/.

[9] F-secure, "Warning on possible android mobile trojans," 2010, https://www.f-secure.com/weblog/archives/00001852.html.

[10] Forbes, "Alleged "nazi" android fbi ransomware mastermind arrested in russia." 2015. [Online]. Available: http://www.forbes.com/sites/thomasbrewster/2015/04/13/alleged-nazi-android-fbi-ransomware-mastermind-arrested-in-russia/47fd631360a9

[11] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 931–948.

[12] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets." in *NDSS*, 2014.

[13] Z. Xu and S. Zhu, "Abusing notification services on smartphones for phishing and spamming." in *WOOT*, 2012, pp. 1–11.

[14] G. Inc, "Android and security," 2012. [Online]. Available: http://googlemobile.blogspot.fr/2012/02/android-and-security.html

[15] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Capkun, "Personalized security indicators to detect application phishing attacks in mobile platforms," *arXiv preprint arXiv:1502.06824*, 2015.

[16] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[17] S. Kitayama, S. Duffy, T. Kawamura, and J. T. Larsen, "Perceiving an object and its context in different cultures a cultural look at new look," *Psychological Science*, vol. 14, no. 3, pp. 201–206, 2003.

[18] S. J. Simon, "The impact of culture and gender on web sites: an empirical study," *DATA BASE*, vol. 32, no. 1, pp. 18–37, 2001. [Online]. Available: http://doi.acm.org/10.1145/506740.506744