GUI-Squatting Attack: Automated Generation of Android Phishing Apps

Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu

Abstract—Mobile phishing attacks, such as mimic mobile browser pages, masquerade as legitimate applications by leveraging repackaging or clone techniques, have caused varied yet significant security concerns. Consequently, detection techniques have been receiving increasing attention. However, many such detection methods are not well tested and may therefore still be vulnerable to new types of phishing attacks. In this article, we propose a new attacking technique, named GUI-Squatting attack, which can generate phishing apps (phapps) automatically and effectively on the Android platform. Our method adopts image processing and deep learning algorithms, to enable powerful and large-scale attacks. We observe that a successful phishing attack requires two conditions, page confusion and logic deception during attacks synthesis. We directly optimize these two conditions to create a practical attack. Our experimental results reveal that existing phishing defenses are less effective against such emergent attacks and may, therefore, stimulate more efficient detection techniques. To further demonstrate that our generated `phapps` can not only bypass existing detection techniques, but also deceive real users, we conduct a human study and successfully steal users' login information. The human study also shows that different response messages (e.g., "Crash" and "Server failed") after pressing the login button mislead users to regard our phapps as functionality problems instead of security threats. Extensive experiments reveal that such newly proposed attacks still remain mostly undetected, and are worth further exploration.

Index Terms—Android phishing apps, android GUI attacks, android apps

1 Introduction

Due to the portability and convenience of mobile devices, mobile apps have surpassed traditional desktop applications, as the primary way of accessing the Internet. Many users heavily depend on their smartphones for daily tasks, such as shopping, payments, and chatting through mobile apps. This kind of popularity has attracted great attention from attackers with a growing number of malicious apps over the past few years. Among these malicious apps, phishing is the most popular and widely used strategy [58] involving the act of harvesting user names, passwords, and other sensitive information from a user. This identity theft poses a security threat for all mobile apps; however, the consequences are particularly severe for financial and social apps. It is reported that mobile phishing apps lead to the loss of billion dollars every year [1].

In traditional phishing attacks, attackers send SMS or emails containing malicious links to redirect the browser to external phishing web pages or inducing download activities to install malicious applications on users' devices [17]. Moreover, phishing attacks are not necessarily sent in bulks but can be highly targeted, such as credential spearphishing [39] and whaling attacks [40]. The effectiveness of such phishing methods have been reduced due to the increased public awareness of risk and a plethora of research about automatically detecting phishing web pages [73]. So attackers sought to propose more sophisticated methods, such as embedding attacks directly inside the apps. In particular, attacking the graphical user interface (GUI). For example, attackers will build a phishing app to masquerade as the original one by repackaging or cloning the original one to steal the private information entered in the login pages [15]. There are two challenges to perform this attack successfully. First, these methods require substantial effort and strong domain knowledge to carry out static program analysis to understand and mimic the logic of the original apps. Moreover, for cloning apps, the difficulty is increased when the UI pages in the original apps have dynamic loading areas which are not determined by the UI resources [21]. Second, the original apps may not be able to be replicated due to the development of app protection techniques (e.g., app packing [2] and code obfuscation [29]). In addition, the state-of-the-art defenses (e.g., fuzz hashing technique [78] and centroid-based approach [20]) can detect repackaging and cloning phishing attacks successfully and effectively. Hijacking existing original apps (e.g., window overlay and task hijacking) could also be detected and mitigated by state-of-the-art detection techniques [15], [35], [59], [60].

A Squatting attack [10] is a form of denial-of-service (DoS) attack where a program interferes with another program through the use of shared synchronization objects. There exist several attack derivatives for different scenarios, such as typo-squatting attack, skill-squatting attack, and

- *S. Chen, L. Fan, and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: {chensen, llfan, yangliu}@ntu.edu.sg.*
- *C. Chen is with Monash University, Clayton, VIC 3800, Australia. E-mail: chunyang.chen@monash.edu.*
- *M. Xue is with The University of Adelaide, Adelaide, SA 5005, Australia. E-mail: jason.xue@adelaide.edu.au.*
- *L. Xu is with New York University Shanghai, Shanghai 200122, China. E-mail: lihua.xu@nyu.edu.*

Manuscript received 9 Mar. 2019; revised 14 Oct. 2019; accepted 8 Nov. 2019. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Lingling Fan.)
Digital Object Identifier no. 10.1109/TDSC.2019.2956035

voice-squatting attack. In this paper, we propose "*GUI-Squatting Attack*", a new approach to automatically generate phishing apps effectively, within a few seconds, resulting in a powerful new attack for the real world. The generated phishing apps (called *phapps* in this paper) have very similar login-related UI pages corresponding to the original apps. Additionally, phapps have been encoded with deception code which can steal sensitive information secretly. We observe from the existing phishing techniques (e.g., repackaging and cloning phishing attacks, and zero-day phishing attacks [45]) that a successful phishing attack requires two conditions: *page confusion and logic deception*[1] (i.e., deceiving users with high similarity UI pages and stealing their information with deceptive UI responses after clicking the "login" button). Our GUI-Squatting attack optimizes these two conditions by leveraging image processing and deep learning methods, making a powerful attack, which can easily bypass state-of-the-art detection techniques.

To illustrate our phishing attack threat model, we follow the assumption made by [21], [60], we assume that Alice downloads a generated phishing banking app from an unreliable app market on her new smartphone. Installing the app does not raise any concerns of Alice as it only requires the permission to access the Internet. Launching the app does not raise any concerns either as the phishing app has a high similarity with the original app's UI pages. Alice clicks the "login" button after entering her personal banking credentials, and a dialog pops up, reminding Alice that the current banking app is out of date, and needs to be updated to the latest version. In parallel, her credentials have been recorded and transmitted to a remote server owned by the malicious app author. When Alice clicks "Update Now", Google Play is launched and redirected to the download page of the corresponding original app. Alice continues to use the original app without noticing that her sensitive information has already been stolen. Similar malicious apps by repackaging or cloning have been previously discovered [15], [21].

Motivated by the scenario above, we implement a new approach to automatically and effectively generating a new phishing app within a few seconds. Given only the login page(s) of an app, with no other requirements, we first extract all GUI components by adopting image processing techniques, next we obtain the component types through image classification. According to these identified components and their attributes in the original page, we generate the corresponding GUI code. Finally, we add deception code for the interactive GUI components to collect users' information and return a certain response to resolve the users' doubts about the phishing app. To increase the authenticity under real-world scenarios, we collected 10 types of responses following the "login" button from 50 real apps, to have our generated phapps randomly return one of these real responses.

Our approach is able to conduct a new powerful phishing attack in the real word due to the following three characteristics: (1) It is difficult for the generated app to be spotted as a phishing one. The generated login-related page (s) are very similar to those of the original app, with subsequent responses sourced from the original apps, mobile users cannot distinguish between the phapp and the original app (Section 5). In addition, the generated apps require very few permissions (only Internet access), and is therefore undetected by both users and existing malware detection techniques. (2) The generation process is fully automated without a need for humans to understand the complicated deception code of the app. Therefore, the attackers can easily generate a large number of phishing apps in a short amount of time (each new app takes 3 seconds on average) to launch large-scale attacks. (3) The generation method is platform-independent. Although the current implementation is based on the Android platform, it can be extended to other mobile platforms like iOS as long as we can collect data from those platforms. In addition, according to the recent news headlines [9], phishing attackers have started leveraging GDPR [5] as a themed (*bait*) in an attempt to steal users' information. Users usually receive scam emails with malicious links, showing that they should update their apps to comply with a new Privacy Policy, which reflects changes introduced by GDPR. Such hotspot can be used as an actual bait to make GUI-Squatting attacks possible in the real world. Android malware can be spread through a variety of techniques [37], [78], they can all be used to propagate and push the phapps to the users' mobile devices, which is out of scope of our research in this paper.

The experiments show that our method can accurately segment and classify most GUI components (83.2 percent accuracy) in the UI screenshot, and the generated login pages are on average 96 percent similar to the original page in a pixel comparison.[2] We then further demonstrate that the generated apps cannot only bypass existing malware or phishing app detection methods, but can also successfully capture mobile users' credentials without alerting users of the human study. The human study involved 20 real participants and 100 apps (50 original apps and 50 generated phapps). This study demonstrates that the different response messages, such as "Crash" or "Server failed" after pressing the "login" button, make users incorrectly regard the phapp as a functionality problem instead of a security threat. Our study also reveals insights that users care more about the security of financial apps than social ones, and that gender or profession does not result in much difference to the experimental results.

In summary, this paper makes the following contributions:

- We introduce a new approach for automated mobile phishing app (*phapp*) generation, which can be used on different mobile platforms, such as Android and iOS. The costless method enables a new powerful and large-scale attack ("*GUI-Squatting Attack*") to different apps in a short time (2.51 seconds for each app on average).

1. In this paper, logic deception refers to reasonable app responses (i.e., deception code) when clicking *interactive components* in login-related pages. Since our goal is to steal users' credentials, we do not attempt to generate the actual logic/back-end code that is similar to the original apps.

2. More results about the extracted components and the similarity comparison can be found on https://sites.google.com/view/gui-squattingattack/

- Our generated phishing apps can bypass the state-of-the-art anti-phishing techniques (e.g., DroidEagle [66] and WindowGuard [59]). Meanwhile, malware detection (e.g., Drebin) and anti-virus techniques (e.g., VirusTotal) are weak in identifying phapps.
- Our comprehensive experiments and human study also show the effectiveness and practicality of our generated phishing apps which successfully steal users' information imperceptibly in the real world. The analysis of users' feedback is also valuable to future research.

At a high level of this work, our experimental results reveal that phishing defenses should effectively respond to such newly proposed attacks. Our approach can aid the process to further understanding and to explore the characteristics of new mobile phishing apps.

2 Mobile Phishing Attack

In this section, we introduce the Android GUI framework and potential security threats arising due to consistent UI design principles. Additionally, we briefly introduce the types of mobile phishing attacks that have been exhibited.

2.1 Android GUI Framework

The Android GUI framework is famous for multi-interactive activities. The GUI is what the user can see and interact with. The Android GUI provides a variety of pre-built components, such as structured layout objects (e.g., LinearLayout) and components (e.g., Button and EditText). These elements allow developers to build the graphical user interfaces for the app. The layout structure uses a GUI-hierarchy to follow UI design principles.

The Android GUI framework is a reusable and extensible set of components with well-defined interfaces that can be specialized. However, the security of Android GUI framework remains an important yet under-scrutinized topic. The Android GUI framework does not fully consider security issues. For example, a weaker form of GUI confidentiality can be breached in the form of GUI state by a background app without requiring any permissions. The design of the GUI framework can potentially reveal each GUI state change through a newly-discovered public side channel – shared memory, giving a chance for attackers to steal sensitive user input [21]. The UI pages of Android apps are usually rendered by static XML files, which reduces the attack costs to control every pixel of the screen. If the attackers can extract the GUI components and their attributes, they can generate the corresponding GUI code smoothly.

Furthermore, when a user is interacting with the target GUI component like clicking or through voice controlling, it can actually trigger some other actions in the background such as tapjacking attack [61], which was not intended by the user. In fact, the Android platform has been plagued by various GUI attacks in recent years, such as phishing attacks, task hijacking [60], and the full screen attack [15]. Malware on the device that takes screenshots also breaches GUI confidentiality [46].

2.2 Existing Mobile Phishing Attacks

Phishing, as a type of social engineering attack [15], [58], is often used to steal user information, such as login credentials. It occurs when an attacker masquerades as a trusted entity (resembling the original web page or application) [43]. Web phishing attacks date back to 1995 [57], but recently, attackers have shifted their attention to mobile devices [37]. Due to the small screen size and lack of identity indicators of URLs seen next to online web sites, mobile users have become more vulnerable to phishing attacks. On mobile devices, 81 percent of phishing attacks are carried out using phishing apps, `SMS`, or web pages [71]. Mobile oriented phishing attacks are classified into two strategies: (1) masquerade as original apps; or (2) hijack existing original apps. Mobile phishing attacks can be classified into three types based on the above two strategies.

- *Similarity attacks (spoofing attacks)* analyze the GUI code of the original app and partially modify the GUI code. Attackers then add logic code to manipulate the original app logic [66]. For example, attackers can crack payment apps to bypass the payment functionality.
- *Window overlay attacks* render a window on top of mobile screen, either partially (e.g., Toast and Dialog) or completely (e.g., similar UI pages) overlapping the original app window [15], [21], [61]. For example, attackers choose a particular time to render the phishing UI pages by monitoring the occurrence of the original app's login activity. This attack usually leverages the flaws of design mechanism in mobile OS (e.g., using ActivityManager#getRunningTasks() to get "topActivity" before Android 5.1).
- *Task hijacking attacks* trick the system into modifying the app navigation behaviors or the tasks (back stacks) in the system [35], [60]. For example, The back button is popular with users because it allows users to navigate back through the history of activities. However, attackers may abuse the back button to mislead the user into a phishing activity (e.g., misusing "taskAffinity"). In short, attackers try to modify the tasks and back stack to execute phishing attacks.

2.3 Newly-Proposed Attack: GUI-Squatting Attack

We follow the assumption summarized by the existing mobile phishing attack techniques: a successful phishing app requires two conditions: *page confusion and logic deception*. In this paper, we propose a new powerful and large-scale attack (called "*GUI-Squatting Attack*") based on fully automated generation of phishing UI pages and apps. Moreover, our approach can generate similar UI pages for the phishing attacks mentioned above.

The following differences make the GUI-Squatting attack more threatening than previous attacks. (1) Only the login page(s) of an app is required and no other inputs are necessary, making a large-scale attack possible, regardless of platform limitations. (2) No requirements of domain knowledge and traditional attack techniques (e.g., repackaging and clone techniques) make the result harder to detect. (3) It can conduct a wide range of attacks due to the low cost of the generation process, and it can launch targeted attacks like

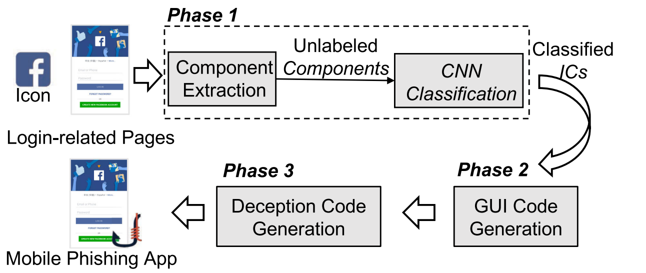


Fig. 1. Workflow of our approach (ICs is short for interactive components).

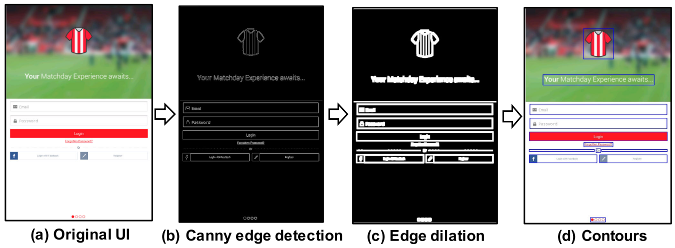


Fig. 2. Process of GUI component extraction.

credential spearphishing attacks [39]. Our generated phishing apps can successfully control every pixel of the screen and capture real users' credentials without raising the user's attention under practical GUI-Squatting attacks in the real world. We detail the new strategy in Section 3.

3 Our Approach

In this section, we first propose our threat model, and then introduce our new approach with three phases to automatically generate mobile phishing apps and UI pages.

3.1 Threat Model

We follow the assumption made in [60] that our generated phishing apps have been installed on the users' mobile devices. There are many propagation techniques capable of pushing malicious apps to user devices [37], which we consider beyond the scope of this paper. The generated apps only need the "INTERNET" permission, frequently requested by Android apps. Due to the high similarity between the original UI pages and the ones in our phapp, the app that the user does not realize is a phishing replica. The credentials will be collected and transmitted to a remote server after the user enters personal credentials and clicks the "login" button. At the same time, a response is shown (e.g., "update required" dialog, crash dialog, no response) to create a diversion so that the user does not suspect that their sensitive information has been stolen.

3.2 Approach Overview

The goal of our approach is to take in the login-related screenshots of a mobile app *lui*, the icon of a mobile app *icon*, and output a phapp that can collect user credentials. In order to generate phapps that are able to deceive users and successfully steal users' sensitive information imperceptibly, our approach needs to address two challenges: 1) To enable page confusion, the generated login-related UI pages should have a high similarity with the original ones. 2) To enable logic deception, deception responses need to be provided, especially for interactive components, including the functionality of interacting with other UI pages, hence corresponding deception code needs to be generated automatically.

To meet these conditions and successfully generate mobile phishing apps, we propose our approach to fully automate phishing app generation in Fig. 1. Our approach has three phases: (1) we extract the GUI components from the target UI screenshots by segmenting the components with image processing techniques (i.e., canny edge detection and edge dilation), and classify the types of GUI components with a deep learning algorithm (i.e., CNN); (2) we then assemble these components in assistance with the layout code snippet of each component along with their attributes, to generate layout code (i.e., XML file) for the imitation login page that is still highly similar to the original; (3) we further generate the deception code and assign responses for interactive components (ICs), such as ImageButton and EditText. The generated phishing apps can secretly collect users' credentials without causing users' awareness through these response messages.

3.3 Interactive Components Extraction (Phase 1)

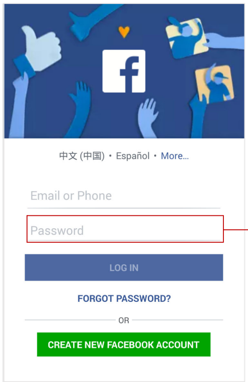
The extraction of interactive GUI components involves two steps: *component segmentation* and *component classification*.

GUI Component Segmentation. To segment the components from UI screenshots, we first detect the edges of all components in the screenshot through canny edge detection [3] which infers the edges by suppressing intensity gradients of the image. But the detected edges are too coarse to be used directly because this technique also detects the exact edges of each character and letter, which does not represent a full UI component. For example, the letters of "Password" in Fig. 2b are isolated from each other. Thus we merge adjacent elements by edge dilation [4], which gradually enlarges the boundaries of regions so that the holes within the regions become smaller or entirely disappear. As shown in Fig. 2c, the EditText with its hint texts and the background image have merged together.

We observe that although some UI components may use irregularly shaped elements, we opt to bound all components as rectangles to make the component identification and code generation process easier. Therefore we adopt contour detection to obtain the regions with an approximate rectangle border. Fig. 2d shows our detected GUI components with all components annotated with rectangular, blue bounding boxes. We crop these regions from the screenshots as images of the GUI components, and also record their coordinates and sizes for later use in the classification and generation process.

GUI Component Classification. We then classify the cropped images of these GUI components into different types such as Button and EditText. To carry out the GUI component classification, we adopt a Convolutional Neural Network (CNN), a state-of-the-art approach often used in computer vision applications.

The model takes as input the cropped images of GUI components and outputs an $\mathcal{N}$ dimensional vector where $\mathcal{N}$ is the number of classes that the program has to choose from. As we are only concerned about the




```
<RelativeLayout...
<ImageView
    android:layout_width="1080px"
    android:layout_height="1845px"
    android:scaleType="fitXY"
    android:background="@drawable/bg"/>
<ImageButton...
<EditText...
<EditTextandroid:id="@+id/pwd"
    android:layout_width="891px"
    android:layout_height="97px"
    android:layout_marginLeft="112px"
    android:layout_marginTop="959px"
    android:fontFamily="sans"
    android:background="#ffffff"
    android:hint="Password"
    android:paddingLeft="13px"
    android:textColorHint="#BEBEBE"
    android:inputType="textPassword"/>
<ImageButton...
<ImageButton...
<ImageButton...
</RelativeLayout>
```

Fig. 3. GUI code snippet of layout.xml file generated by our approach for phapp.

interactive GUI components which need extra GUI code in the login page and deception code, we consider the components of EditText, Button, ImageButton, TextView, and CheckBox. Note that Some TextViews contain clickable links and will be discussed later in Section 3.4. Other components, such as ImageView and Spinner, are put into one type called "Others." Thus, $\mathcal{N} = 6$ and our model is to classify a cropped component as one of these 6 types. Note that the output of the fully connected layer will be the probability of these 6 classes, where the sum of probabilities is 1.

3.4 GUI Code Generation (Phase 2)

In the second phase, we generate a GUI code snippet of the corresponding component based on the classified types of components, and embed their attributes collected from the component images, as shown in Fig. 3.

After obtaining a list of interactive GUI components, we generate the phapp following Algorithm 1. The inputs to our algorithm include lui as a list of UI screenshots of the Android app's login pages and $icon$ as the icon of the Android app. Note that one app may have several login UI pages. For example, it may require users to fill in the user name on the first page, and then fill in password in the next page. So we set the number of login UI pages as N ($N \geq 1$). We first obtain the list of GUI interactive components ordered from top to bottom, and from left to right on the original screenshot as ICs.

For each UI page, we separately generate GUI code and deception code since GUI code is usually maintained in an XML layout file, and the back-end code is usually maintained in one or more Java files. Apart from several interactive components for which we need to generate extra interaction code, most parts of the page do not need any change. Thus we put the original login UI screen(s) as the background canvas and add interactive components later. Specifically, for each UI page, we first initialize GUI code $code_{gui}$ as the code generated from the screenshot and leave deception code $code_{deception}[i]$ (i refers to the ith lui) empty (line 8) as the background canvas does not involve any deception code in apps. We then obtain attributes for each interactive component extracted from phase 1. For each component, we collect its cropped image, detailed coordinates with `getAttr()` in line 10. However, among the five interactive components, there is one special type, EditText. Apart from basic attributes, it may also contain text hints (reminder messages like "Email", "Password" as shown in Fig. 2) or drawable images (e.g., an email representation image or a password visibility toggle). Therefore, we check the existence of such hints and obtain their text by leveraging optical character recognition (OCR) techniques [8], and also extract drawable images from inside the EditText. Since EditText may also own a particular background color (e.g., white, blue), we take the most frequent pixel value to fill in the area of EditText. Fig. 3 shows the generated GUI code of one of these EditText components with detailed attributes.

The other special type of interactive component is TextView, many of which just display text without any interaction. However, some TextViews are special with clickable links, for example, an interactive TextView is used to assist a user in password recovery (i.e., "FORGOT PASSWORD?" as seen in Fig. 3). Therefore, to preserve this functionality, we also retrieve the text attributes of TextView through OCR, and treat them as an interactive component in the login-related pages if the text contains words that are matched with those in a keyword set (e.g., "sign up", "forget password" or related alias) with function isInteractive() in line 11. Otherwise, we ignore it both in GUI code and deception code (line 12).

Algorithm 1. Phapp Generation

Input: lui: a list of login-related UI pages
$icon$: icon of the Android app
Output: app: generated Android phishing app (phapp)
// GUI Code Generation
1: $N \leftarrow$ number of lui
2: $i \leftarrow 0$
3: $code_{gui} \leftarrow \varnothing$
4: $code_{deception} \leftarrow \varnothing$
5: $ICs \leftarrow$ getInteractiveComponents(lui)
6: **while** $i < N$ **do**
7: $code_{gui}$[i] $\leftarrow$ generateComponentUI(lui)
8: $code_{deception}$[i] $\leftarrow$ " "
9: **foreach** $ic \in ICs[i]$ **do**
10: $ic_{attr} \leftarrow$ getAttr(ic)
11: **if** ic == $TextView$ **and**!$isInteractive(ic)$ **then**
12: **continue**
13: $code_{gui}$[i] += generateComponentUI(ic_{attr})
// Deception Code Generation
14: $code_{deception}$[i] += generateComponentListener(ic_{attr})
15: $i = i + 1$
16: phapp $\leftarrow$ generateApp($code_{gui}$, $code_{deception}$, $icon$)
17: **return** phapp

We generate GUI code for every interactive component according to its attributes, and add the code into the overall linear layout of the GUI code file (line 13). For Button, ImageButton, and interactive TextView, we generate GUI code by utilizing ImageButton, i.e., cropped component images which can be clicked. For EditText, we obtain its GUI code by also considering any of its text hints, drawable images and background color (shown in Fig. 2).

TABLE 1
Interactive Components not Directly Associated With the Login Logic

Icon	Interactive Components	Functionalities	Deception Code Implementation
Keep username	Checkbox outside EditText	Remember user name	Saved in SharedPreference, SharedPreferences#getSharedPreferences
	Checkbox inside EditText	Display plaintext password	EditText#setTransformationMethod
OFF	Switch Button	Remember user credentials	Saved in SharedPreference
 SignUp	ImageButton	Login with third-parties (e.g. Facebook, Twitter); Sign up	Use the same response as the login button
Forgot password	Clickable TextView	Forget password	Use the same response as the login button

3.5 Deception Code Generation (Phase 3)

In the third phase, we generate the corresponding deception code snippets based on different types of components in the layout file, as well as different event listeners. We allocate different types of responses collected from real apps to the "Login" buttons. Meanwhile, we implement SSL/TLS authentication and user identity verification via HTTPS connection for each phapp to prevent being detected by traffic analysis tools. Additionally, to prevent being detected by control- or data-flow analysis, we create some widely-used activity transition relations for each phapp.

After generating the GUI code $code_{gui}$ for login images, we then generate the corresponding deception code $code_{deception}$ (line 14). Specifically, we set up listeners for different interactive components. Since our goal is to automatically generate phishing apps that can steal user credentials imperceptibly instead of cloning apps, we focus on generating the deception code of login-related pages of the original apps, and attempt to deceive users by displaying the highly similar login pages and showing plausible responses when clicking the "Login" button. According to our observation of login-related pages, we summarize two kinds of deception code that need to be generated based on different interactive components.

Interactive Components that are Directly Related to Basic Login Logic. (i.e., EditText for inputs and Button for submission). As users can enter their information including their user names and passwords in EditText, we add listeners to each EditText to collect users' credentials. For "login" buttons, we regard it as ImageButton in the GUI code, and add a listener (i.e., View#OnClickListener) to it. Once the submission component is clicked, the listeners for EditText will check whether there is content inside. If not, there will be a pop-up message reminding the user to"please fill in the account and password." Otherwise, the data collection program will be triggered, and the credentials are transmitted to a remote server via the "getText()" method.

Interactive Components that are Associated with other Functionalities or other UI Pages. As shown in Table 1, based on our observations of real apps, we summarize and demonstrate five kinds of interactive components that are most widely used. These interactive components may appear in the login-related pages; however, they are not directly associated with the login logic. For *Checkboxes outside EditText*, we use SharedPreference#getSharedPreferences to save the inputs of EditText to determine whether the Checkbox has been chosen or not. In addition, we use EditText#setTransformationMethod to control the plain-text display of the password in some cases. The implementation of a *Switch* is similar to Checkbox outside EditText. For *ImageButton* of third-party logins (e.g., Facebook and Twitter), the credentials are used via the interfaces from the corresponding parties, which are out of scope of our research in this paper though it could be possible to generate a phapp for the standardized Facebook or Google login page. Besides, the ImageButton of "Sign up" and interactive TextView of "Forgot password" will indicate that the current user does not have valid credentials; they are users who are not our phishing target, and thus it is meaningless to steal credentials from them. We therefore allocate the same response as clicking the "login" button to make them interactive. Note that, for ImageButton, Button, and interactive TextView, we treat them all as ImageButton in the GUI code, and add listeners for all of them.

We collected and identified 10 different types of responses for the "login" button. Among 37,251 Android apps automatically explored in Section 4.1, we randomly sample 50 of them which could not be logged in for a manual check. We check the screenshots of these apps after clicking the "login" button, and summarize the ten responses in Table 2. We find that 60 percent of the apps return "Invalid inputs", i.e., wrong user name or password. Other unsuccessful login pages include "Crash", "Server failed" (no connection to the remote server), "App update", "Network unavailable" (no connection to Internet), "Keep loading" (showing the progress bar), "Slow response" (delay of the app), "Google service update", "Force exit" (exit without notification), and "No response" (no feedback after the action). When generating the phishing apps, we randomly select one of these responses to camouflage our app as an original with functionality problems as shown in Fig. 4.

TABLE 2
Response Types Extracted From Real Apps

Types of Response	Description	#
Invalid inputs	Wrong user name or password	30
Crash	Unfortunately, the app has stopped	6
Server failed	Can not connect with remote server	4
Update app	Update the latest version from market	2
Update Google service	Update Google service from market	2
Network unavailable	Check your network connection	2
Keep loading	Keep showing the loading status	2
Slow response	Simulate system delay	2
Force exit	Exit app directly	1
No response	No feedback after the action	1

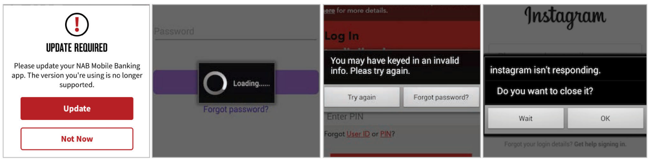


Fig. 4. Response examples after clicking "login" buttons.

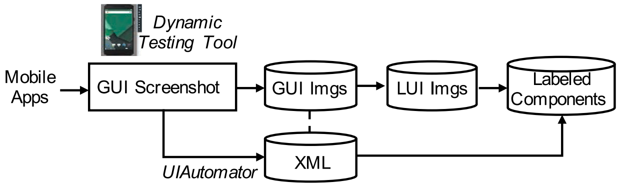


Fig. 5. Training data collection.

Listing 1. Simplified Code Snippet of Server Authentication in Phapps

```
1  // Phapp server authentication
2  X509TrustManager trustManager = new
   X509TrustManager(){
3   // Certificate verification
4   public void checkServerTrusted(...){
5    for (X509Certificate cert : chain){
6     // Is it expired
7     cert.checkValidity();
8     // Certificate public key string
9     cert.verify(ca.getPublicKey());
10 }}}
11 // Hostname verification
12 final HostnameVerifier hostnameVerifier=
   new HostnameVerifier(){
13  public boolean verify(...){
14   if(URL.equals(hostname)){
15    return true;
16 }};
```

With the help of Socket or HTTP/HTTPS connections, our remote server (i.e., webpage) will receive users' credentials after users enter their information and click the submit or login button. Such one-way communication may be vulnerable to detection through traffic analysis, which tracks network traffic from the client to the server by using a simple pattern-based approach. To avoid being detected, we implement *server authentication* and *user identity verification* for each phapp. (1) We implement SSL/TLS authentication (the core simplified code snippet is shown in Listing 1) when the client side (i.e., phapp) sends network requests to mimic the real communication between the client and server sides. Specifically, we first generate the server certificate using `keytool` (i.e., *keytool -genkey -alias phapp -validity 3560 -keystore phapp.keystore*), which is later imported at the server side. After that, we also use `keytool` to export *public key string* of the server certificate, which is used to verify the server certificate at the client side. Server authentication contains two phases: server certificate verification (Lines 2-10) and server hostname verification (Lines 12-16). ① For the verification of the server certificate, we use *checkValidity()* to verify whether the certificate is expired or not, and use *verify()* (Line 9) and *getPublicKey()* to verify the *public key string* of the server certificate. ② For the verification of the server hostname, we just verify the domain name address. Moreover, we dynamically compose the server URL (Line 14) using separate strings to evade the black-list matching strategy. (2) We implement user identity verification via HTTPS for each phapp by returning an always-true result. Before pushing different types of responses for the "login" button, the server will check the validity of the token sent from the phapp, and the client side also will parse the received token no matter what data is sent from the client side (the core simplified code snippet is shown in Listing 2). Note that, a true result will be returned from the server side, indicating that the user is valid. Then, the response will be pushed to users, and the response about the functionality problem will be displayed on the top of the screen to distract users so that they do not regard the phapp as a phishing app.

Listing 2. Simplified Code Snippet of User Identity Verification in Phapp

```
public void send(...) {
 new Thread(new Runnable()) {
    // Send the login data to server
       Request req = new Request.Builder().url
 (URL).post(login_data);
    OkhttpClient client = new OkHttpClient();
    // Check the login data and receive response
     Response res = client.newCall(req).exectue
();
    receivedDataParsing(res);
}}
```

Some control- or data-flow analysis methods [22], [54] analyze the transitions between activities, it would raise suspicion if there is no transition between the login activity and other activities in an app. To evade it, we create many templates of activities that are widely used to interact with the login activity, such as register activity, main activity, and setting activity. To set up the transitions between them, we leverage the API `StartActivity()` provided by Android system to enable the activity transition from activity A to activity B. Such activity transitions help address the doubts of flow-based analysis. In fact, the users would not observe the existence of these activities since the app would encounter functional problems after users click the "Login" button.

In addition to event handler generation, we further bind the GUI code and deception code via findViewById(), which identifies the corresponding component from the layout file (i.e., GUI code) and binds it with the deception code. To avoid being detected by other anti-phishing techniques based on screenshots, we prohibit our apps from having screenshots taken by other third-party apps by setting the flag (WindowManager.LayoutParams.FLAG_SECURE = TRUE) on the login page. With the app icon, and the generated GUI code, deception code, we finally build the phapp (line 15).

4 Implementation

4.1 GUI Component Collection

Fig. 5 shows the training data collection process. We crawled 37,251 unique Android apps with the highest installation numbers from Google Play Store. These apps belong to 30

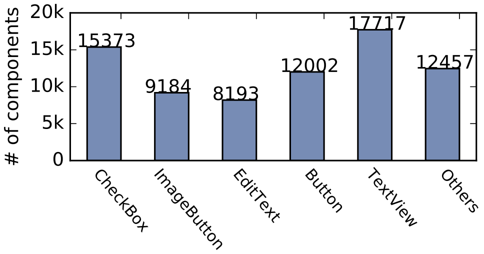


Fig. 6. Number of labeled GUI components.

categories, including finance (e.g., Bank of America), social (e.g., Facebook), news (e.g., BBC News), etc. Game apps have been excluded due to lack of standard GUI components that can be automatically extracted. We obtain billions of original UI screenshots in assistance with dynamic Android testing tools (e.g., UIAutomator [12] and Stoat [65]). These tools are configured with the default setting and run on Android emulators (Android 4.3) on Ubuntu 14.04. At the same time, we use UIAutomator to extract component information (i.e., component types and coordinate positions) for the explored app screens. We note that not every app was successfully launched on the emulator due to version update warnings, Google service update warnings, lack of third-party library support, etc. Our goal of this large-scale component analysis is to ensure we obtain multiple sets of screenshots and components, rather than completely explore each app and obtain all components in each screenshot. Although the layout information from UIAutomator does not include all components and may contain minor errors, it would not affect the collection of our training set. Finally, the result data set contains 1,842,580 unique screenshots based on pixel comparisons, which is by far the largest raw data set of UI screenshots to our knowledge.

Since we only focus on login-related pages and generate corresponding code for phapps, we extract login-related screenshots or closely related login screenshots (e.g., related with register, transfer, and submission) by (1) using keyword filtering (i.e., login, sign, regist, transfer, submit), and (2) ensuring the screenshots to contain the component types of EditText, TextView, and Button. We finally obtain 4,420 login-related screenshots, from which we extract 57,209 labeled cropped GUI component images (6 types) shown in Fig. 6. Note that since we only managed to collect 697 CheckBox components in the login-related screenshots, we extend it with 14,676 CheckBox components from the other unique screenshots we collected. We place other components that appear infrequently into the “Others” category (for 12,457 in total), including ToggleButton, RadioButton, ImageView, etc., since we do not need to handle all component types. This part differs from the state-of-the-art GUI code generation tools [14], [19]. Meanwhile, we disregard the components that do not appear in login-related pages, such as Spinner, RatingBar, and SeekBar.

4.2 Approach Implementation

Our approach is implemented in Python 2 (3K+ Lines of Code), and leverages several open source libraries (e.g., OpenCV, Tesseract) to automatically generate phapps. Specifically, we use CV (i.e., OpenCV [7]) and OCR techniques (i.e., Tesseract [11]) to extract components and their attributes (e.g., coordination positions, width, height, color, texts) from the screenshots of UI pages. Meanwhile, we use Tesseract#makebox to extract the coordinate of each letter.

To classify the types of segmented components within the UI screenshots, we adopt the CNN model as discussed in Section 3.3. Our model contains three convolutional layers, three pooling layers, and two fully-connected layers. Within the convolutional layer, we set the filter size as 3, the stride as 1, and padding size as 1. The same setting also applies to the pooling layer. For two fully-connected layers, both have 128 neurons. We implement our network with the Tensorflow framework written in Python. The model is trained for roughly 2 hours on a CPU, RAM, and Nvidia Tesla P40 GPU card (24G memory) over 10 epochs.

From the classified interactive components and their attributes, we generate the login GUI code for the given UI screenshot. For each component, we use two layout attributes (i.e., android:layout_marginLeft and android:layout_-marginTop) to identify their coordinates. In addition to the basic attribute settings, we also transfer attributes of the component to corresponding layout code (e.g., android:textColor, android:inputType). After implementing the UI login code, we implement 10 types of responses from Table 2 when interactive components are clicked, each component has a different response attached within the deception code. As for the response to login actions, we randomly choose one response to be attached to the “login” button. Our implementation runs on a 64-bit Ubuntu 16.04 machine with 12 cores (3.50 GHz Intel CPU and 32 GB RAM.)

5 Experimental Evaluation

In this section, we conduct extensive experiments to evaluate our approach in the following five aspects: (1) UI page similarity comparison between the UI pages of the original apps and our generated phapps; (2) UI page generation comparison between the state-of-the-art UI generation tools and our approach; (3) Performance of our CNN classification; (4) Ability to evade detection by the state-of-the-art anti-phishing techniques; (5) A human study to identify the power and impact of our phapps.

Dataset. We randomly collect 50 Android apps (25 financial apps and 25 social apps) from the top 100 financial and social categories from the Google Play Store, as the apps in these two categories are usually security- and privacy-critical. All apps require users to login before use. These are the most famous apps (e.g., Facebook, Twitter) with over 1,000,000+ installs, mainly originating from USA, China, and European countries. We guarantee the representativeness of the selected original apps in terms of their number of installs and representative categories. Given the screenshots of login pages and icons of these apps, we generate the corresponding 50 phishing apps using our approach. The dataset of (50 original apps and 50 phapps in total) is used to conduct the following experiments. Besides the 50 financial apps and social apps used in our experiments, in order to reduce the influence of randomness, we further select 20 apps that were downloaded from different times off the Google Play Store that also contain login pages to validate the similarity of our results. From the comparison of results, the corresponding generated UI pages of these 20 apps are also sufficiently similar (they achieve over 95

TABLE 3
Phapps Used in Experiments

App Name	#ICs	Pixel Similarity	Visual Similarity	Generated Time (sec)
DBS IN	9	92.1%	4	2.2
CommBank	5	96.4%	5	1.7
DBS	8	94.8%	5	2.1
Alipay	8	96.8%	5	2.7
Gcash	5	96.2%	4.5	2.7
NetBank	6	94.3%	4	4.1
Reliant	6	93.4%	4.5	3.8
FAB	7	94.5%	4.5	2.4
First	7	94.5%	4	2.1
BankFirst	7	93.6%	5	5.0
AFCU	7	94.7%	4	2.1
GSB	7	92.9%	4	2.3
FSB	7	94.6%	4.5	1.8
ColumbiaBank	7	94.6%	4	2.9
Ulster	7	93.8%	4	2.6
Bridgewater	7	94.3%	4.5	2.0
RFCU	7	94.2%	4.5	3.6
CB	7	94.6%	4.5	3.1
Money	6	95.0%	5	2.3
Bred	3	94.9%	4.5	2.2
Oxigen	5	93.8%	5	1.8
Paga	6	96.1%	5	5.0
BankNordik	5	95.3%	4.5	3.0
Eik	5	95.4%	5	1.7
Nordoya	5	95.3%	4.5	1.8
Reddit	5	95.3%	4.5	2.1
Twitter	4	96.0%	5	2.0
VK	6	95.9%	4.5	2.4
Pinterest	4	93.8%	4.5	1.8
Askfm	9	91.8%	4	1.8
Badoo	4	95.3%	5	1.7
Bharat	8	95.8%	4.5	1.8
BNI	4	93.2%	4	3.2
Facebook	6	95.7%	5	5.0
Instagram	7	96.0%	4.5	2.2
MocoSpace	6	96.4%	4.5	2.2
MeetMe	7	95.1%	4.5	1.9
Path	4	96.5%	3.5	2.0
Weibo	7	97.0%	5	5.7
SKOUT	5	96.1%	4	2.2
Snapchat	5	98.3%	4.5	1.9
Nearby	5	96.5%	5	2.0
WeChat	7	97.1%	5	1.9
ADDA	5	93.2%	4.5	1.7
SayHi	8	94.8%	5	1.8
Vent	5	95.0%	4.5	1.8
LINE	7	95.4%	4.5	1.7
Kik	6	96.7%	4	3.9
Parlor	11	94.5%	4.5	1.8
Yapp	5	94.0%	5	1.9
Average	**6**	**96.0%**	**4.56**	**2.51**

The upper indicates 25 banking apps, the others are social apps. "#ICs" means the number of interactive components.

percent similarity on average in terms of mean absolute error (MAE) and mean squared error (MSE)) and can be used in the GUI-Squatting attack directly. More generated phishing UI pages can be found on our website [10].

5.1 UI Similarity Comparison

One of our goals is to generate phishing UI pages resembling the original. We compare the visual similarity of the generated UI pages and the original UI pages (i.e., screenshots) collected from the 50 original apps listed in Table 3. We use two widely-used image similarity metrics [53], i.e., mean absolute error and mean squared error, to measure the image similarity pixel by pixel. MAE measures the average magnitude of differences between a prediction and the actual observation. While MSE measures the average of squared differences between them. On average, our approach achieves 99 and 96 percent similarity in terms of MAE and MSE (normalized to [0, 1]), respectively. We detail the pixel-by-pixel similarity results (using MSE) of each login UI page in column "Pixel Similarity" of Table 3. "Visual Similarity" represents the similarity results via human observation which will be discussed in Section 6. "Generated time" represents the time cost on each phapp, from an image to a compiled apk.

We can see that the pixel-by-pixel similarity of all the 50 apps is over 90 percent, the average visual similarity is 4.56, and only one app is considered dissimilar with a score less than 4. The results indicate that our generated apps are similar enough to masquerade as the original ones. The average number of components on the login page is 6, only one app (Parlor) has more than 10 interactive components, indicating that attackers can easily create a phishing login page image due to the small number of components on the login pages. Our approach manages to generate each phapp within 2.51 seconds on average, with the highest time cost originating from building the apks.

Remark 1. Our approach achieves 99 and 96 percent similarity in terms of MAE and MSE, respectively, and the average visual similarity is 4.56 based on the participates' feedback from our human study. Our approach can generate a phishing app within 3 seconds.

5.2 Evaluation of the CNN Classifier

Baseline. In this experiment, apart from our method, we also take some widely-used machine learning classification models as baselines, including Logistic Regression (LR), Linear Discriminant Analysis (LDA), K-nearest Neighbors (KNN), Decision Tree (DT), Naive Bayes (NB) and Support Vector Machine (SVM). Note that since traditional machine learning algorithms need the hand-crafted features as the input, we extract two kinds of features from each image. First, for each image, we calculate its color histogram [18], i.e., a representation of the distribution of color in an image. Second, we extract Hu moments features [41] containing 6 different descriptors which capture the silhouette or outline of objects inside the image. Then we concatenate color histogram and Hu moments as the input features for all baseline models.

Setup. Among 4,420 login-related images (Section 4.1), we sample an even number of sub-images from each of the 6 types of UI component: CheckBox, ImageButton, EditText, Button, TextView, Others (see Section 4.1 where it is specified). We then formulate the component classification into a multi-class classification problem. To mitigate the impact of unbalanced data [68], we take 7,900 sub-images for each component i.e., only sampling 7,900 images if one component has more than 7,900 images. Therefore, there are totally 47,400 (7900×6) images for 6 different component types.

TABLE 4
Performance Comparison Among Different Methods

Methods	CNN	LR	LDA	KNN	DT	NB	SVM
Accuracy	83.3%	48.3%	47.7%	68.9%	70.6%	26.6%	36.8%

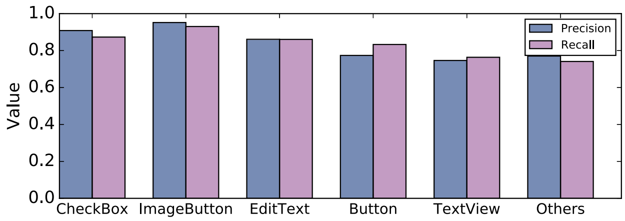


Fig. 7. Performance of our model in 6 different components.

We partition this into an 80 percent split for training, the remaining 20 percent for testing.

Results. Table 4 shows the accuracy of prediction for all seven classification methods. We can see that our model outperforms all baselines with 83.3 percent accuracy, which is 18 percent higher than that of the next best model (Decision Tree 70.6 percent). The results are reasonable, as often in computer vision applications, deep learning outperforms classical machine learning techniques due to reasons such as the abstraction of latent features with suitable algorithms (e.g., CNN). We further analyze the accuracy of our classification between the different component types in Fig. 7. Checkbox and ImageButton both have very high precision, larger than 0.9, with EditText also with a reasonably high precision of 0.86. However, it seems that our model makes more mistakes in classifying Button, TextView and Others with precision below 0.8. We further check which components were misclassified, and find that the most frequent misclassification is that TextView were often misclassified as Button. That is because some TextViews are very similar to Buttons, In particular, TextViews with short text on a certain background color (like blue) which is also commonly used in Button. It is difficult to discriminate them even for human by looking at the single component without considering the context of the component. For the 50 generated phapps in our experiments, only 5 cases failed due to the wrong classification of EditText as TextView, so we manually relabel these components.

Remark 2. Our classification model outperforms all machine learning baselines, with the accuracy (83.3 percent) of our model 18 percent higher than that of the best model among 6 baselines.

5.3 Comparison With State-of-the-Art Techniques

In this section, we choose two state-of-the-art end-to-end GUI code generation tools, PIX2CODE [14] and UI2CODE [19], to compare the similarity of the generated UI pages and the original pages with the similarity of our generated UI pages. We use UI2CODE and PIX2CODE to generate 50 corresponding UI pages. Since PIX2CODE may fail to generate UI pages due to failures in translation from UI pages to the intermediate language (i.e., DSL), and UI2CODE may fail to generate UI pages due to failures in generation from UI pages to an executable apk (i.e., build failure), they can only generate 20 and 35 of the UI pages, respectively. We measure the similarity using MAE and MSE based on the successfully generated UI pages.

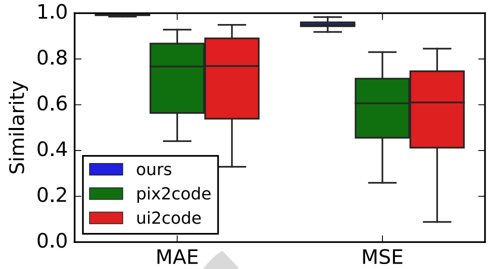
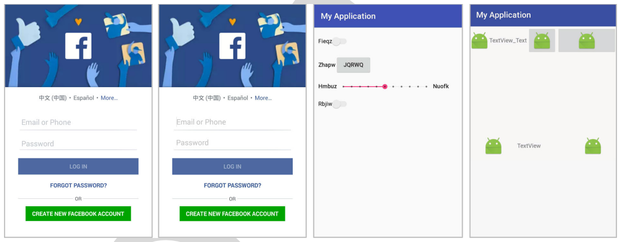


Fig. 8. Pixel similarity comparisons with UI2CODE & PIX2CODE.



(a) Original (b) Ours (c) Pix2code (d) UI2code

Fig. 9. Generated UI comparisons with UI2CODE and PIX2CODE.

Fig. 8 shows the distribution of pixel-by-pixel similarity on the successfully generated UI pages. Our approach outperforms PIX2CODE and UI2CODE in terms of similarity of the generated UIs, achieving over 96 percent pixel-to-pixel similarity. One primary reason is that the two approaches aim to reduce the burden on the GUI code development, but they are not competent in generating an almost identical UI page due to lack of realistic GUI-hierarchies of components and containers of UI pages. Moreover, their approaches cannot extract component attributes, such as coordinate positions, colors and types. Similarity using MAE of UI2CODE and PIX2CODE is mainly between 60-80 percent. As for the metric of MSE, they are mainly between 40-70 percent. To understand the significance of the similarity differences between ours and the pages generated from UI2CODE and PIX2CODE, we apply one-way ANOVA (analysis of variance) [6] for multi-group comparison. We use the standard metric: $\alpha = 0.05$. It shows that the results are significant with a p-value < 0.01.

Fig. 9 displays an example of the generated UIs using PIX2CODE, UI2CODE, and our approach based on the same original UI page. As observed in Figs. 9c and 9d, there is a substantial difference between the original and generated UIs by PIX2CODE and UI2CODE with a human visual comparison. Note that, as for PIX2CODE, some of the generated UI similarity measured by MAE and MSE is still high since some original UI pages contained a white background with login components, as shown in Fig. 9a. Thus when measuring pixel-to-pixel similarity, a large number of pixels are regarded as the same or with high similarity, producing a large similarity value that may overstate how visually similar they appear to a human performing visual comparisons. As for UI2CODE, as shown in Fig. 9d, the results are better than PIX2CODE;

TABLE 5
Detection Results of Multiple Anti-Phishing Techniques for Different Mobile Phishing Attacks

Attack Types \ Detection Techniques	Anti-phishing Techniques: Layout Similarity DROIDEAGLE [66]	Visual Similarity -based [50]	Personalized Indicator [51], [52]	Window Integrity [59] WINDOWGUARD
Similarity Attack	◐	◐	◐	○
Window Overlay	○	○	○	●
Task Hijacking	○	○	○	●
GUI-Squatting Attack	○	○	○	○

●: *Fully detect* ◐: *Partially detect* ○: *Unable to detect*

however, the generated UI pages by UI2CODE still have a big visual difference compared to the original UI page.

Remark 3. Our new approach significantly outperforms PIX2CODE and UI2CODE in terms of pixel-by-pixel similarity of the generated UI pages. The comparison results are significant with p-value < 0.01.

5.4 Bypassing Anti-Phishing Techniques

As shown in Table 5, we choose the most representative mobile anti-phishing and malware detection techniques with different detection strategies to demonstrate that our generated phapps can bypass the state-of-the-art detection approaches [44], [50], [51], [52], [59], [66], [72]. Since these tools are not open source projects, we re-implemented the core functions to conduct our experiments.

Anti-Phishing Techniques. DROIDEAGLE [66] relies on the layout tree to generate layout hash values, and then compares the layout hash values with their repository. Before generating layout hash values, the tool prunes all leaves in the layout tree before hashing, and generates a hash value only for the layout skeleton. Fig. 10a shows the original layout tree of Twitter. Attackers may carry out a similarity attack by deleting the leaf node "CheckBox", resulting in Fig. 10b. However, the hierarchies of the two trees are the same (i.e., LinearLayout, ScrollView, LinearLayout, and LinearLayout), leading to the same layout hash values, thus Fig. 10b can be detected by DROIDEAGLE. Fig. 10c shows the layout tree from our generated phapp, which only has a root node and several leaf nodes. The hierarchy of our layout tree is $*$Layout (e.g., LinearLayout and RelativeLayout), which has a big difference with the original hierarchy.

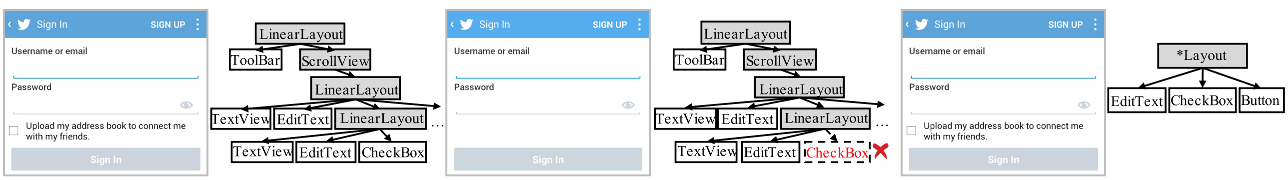
To demonstrate that our generated phapps can successfully bypass the detection of DROIDEAGLE, we first use APKTOOL to translate binary XML files to plain files, and re-implement the procedure of extracting branch nodes (i.e., internal nodes) together with their attributes (e.g., width, height, text). We then compare the extracted node sequence of the original apps with that of the phapps, without further computing their corresponding hash values. Obviously, the hierarchies of the two trees are different, so DROIDEAGLE does not work for phapps.

Malisa et al. [50] use visual similarity comparison on the installed apps on the mobile device by taking screenshots, to detect spoofing apps which have visual differences (i.e., repositioning elements). They do not focus on the detection of perfect copies like ours, and the similarity comparison is not scalable to analyze a large number of apps due to heavy runtime overhead on users' devices. Furthermore, the phapps prohibited screenshots to be taken by third-party apps, such as the pre-installed apps on the users' devices; thus, this approach does not work for our phapps.

Personalized security indicators rely on users to detect phishing attack. When the user starts an app for the first time, he is asked to choose a security indicator for the app, he can also skip it if he does not want to set it up. After that, whenever the app starts, it authenticates itself by showing the security indicator. Users can distinguish benign apps from phishing apps. However, previous work identified that users tend to ignore personalized security indicators [63]. Moreover, many research communities have proved that it is an ineffective phishing detection technique [16]. However, among the 50 selected financial and social apps in our experiments, we did not find any of these apps using personalized security indicators. Marforio et al. [51], [52] revisited personalized security indicators to detect mobile phishing attacks. However, if we conduct a personalized phishing attack, our generated UI can capture the security indicators and will show the correct indicators to users to bypass the detection.

WINDOWGUARD [59] uses the integrity of Android Window Integrity (AWI) to detect phishing attacks efficiently. However, phapps do not use window overlaying or task hijacking when running on mobile devices. Therefore, AWI has no effect on phapps, and WINDOWGUARD also does not work for phapps.

Malware Detection Techniques. Signature, behavior, and dynamic-based detection always rely on the declaration of resource permissions, API calls, system calls, and pre-defined



(a) Original Layout (b) Similar Layout (without CheckBox) (c) Our Layout

Fig. 10. Layout comparisons.

rules to detect Android malware with big data [47], [67], [75]. Our generated phapps only use INTERNET permission, the most commonly-used permission. Meanwhile, Socket, and HTTP/HTTPs communications are very normal ways to communicate between the client and the server. Thus phapps can bypass such techniques. For learnined techniques, we trained a machine learning based classifier on a malicious dataset from DREBIN [13] using Support Vector Machine. For the features, we replicate their defined feature sets (e.g., requested permissions, hardware components, suspicious API calls). We use the trained SVM classifier to classify our 50 phapps. The result we obtained demonstrates that the classifier does not work for phapps. We suspect there are not enough malicious features that can be extracted from phapps.

VIRUSTOTAL contains 61 anti-virus engines, e.g., MCAFEE and KASPERSKY. When we upload our generated phapps, none of the anti-virus engines flag our phapps as malicious. Therefore, our generated phapps are also able to bypass the state-of-the-art Android malware detection techniques.

Traffic Analysis. Traffic analysis [28], [76] can also be used to analyze abnormal behaviors when there is communication between the client (phapp) and server. If phapp only contains the code of credential collection, it would only produce one directional traffic from the client to the server, which would be easily detected by traffic analysis because there is no response and traffic being sent back to the client side (phapp). To bypass traffic analysis, we implement SSL/TLS authentication and server identity verification via HTTPS for each phapp, making the communication behavior of phapps closer to normal apps. In fact, according to the recent work [23], [24], a number of normal apps do not correctly implement the server verification part, while our generated phapps implement correct communication between the client and the server. Therefore, even if the traffic analysis is employed to detect the abnormal behaviors of our phapps, phapps are able to bypass the detection.

Activity Transition Analysis. Activity transition represents the interaction between different activities. If phapp only contains one activity, this approach of detection will be able to identify it by leveraging activity transition graphs (ATG). For example, defenders can use the existing inter-component communication analysis tools (e.g., IC3 [54] and StoryDroid [22]) to check the activity relations. To evade the detection of them, in the deception code generation phase, we implement several common and widely-used activities into phapp, and also build up relations between the login activity and other activities. In the evaluation, we use IC3 to extract the activity transition graphs and compare them with the transition results of normal apps. For example, phapps have the normal relations (e.g., LoginActivity→RegistrationActivity, LoginActivity→MainActivity, MainActivity→SettingActivity). We find that the relation of phapps is similar to normal apps, resulting in successfully evading detection by activity transition analysis.

Remark 4. Our generated phishing apps (phapps) can bypass the state-of-the-art anti-phishing techniques, Android malware detection techniques, industrial virus engines, traffic analysis, and activity transition analysis successfully.

6 HUMAN STUDIES

In Section 5.4, we have demonstrated that our generated phapps can bypass the state-of-the-art detection tools. Another important point of the phishing attack is that the attacker is able to obtain users' information without altering the user. In this section, we demonstrate that these phapps can attack users and obtain their credentials in real scenarios. Since the generated phapps require interaction with users to obtain their input data (i.e., username, password), we design and conduct a human study to evaluate the practicality of the generated phishing apps. Our goals are to check:

- if we can obtain user credentials from the generated phapps without users' awareness.
- if users can differentiate the generated phapps from their original apps based on their login pages.

6.1 Settings of Human Studies

Dataset of Human Study. We use our generated 50 phapps for our human study. The 100 apps (50 original apps and 50 generated apps) are randomly installed on 20 mobile devices (e.g., Nexus 5 and Nexus 5X with Android 4.4) with 8 apps on each device, among which 4 apps are phapps (with 2 financial apps and 2 social apps) and the other 4 are the original apps (still with 2 financial apps and 2 social apps).

Participant Recruitment. We recruit 20 people from our university to participate in the experiment via emails and word-of-mouth. The recruited participants have a variety of occupations, ranging from doctoral students, post-doctoral researchers to administrative staff, including app developers, computer vision researchers, etc. They come from different countries, such as the US, China, Singapore, and European countries (i.e., Spain and Ireland). The male-to-female ratio of participants is 7:3. All of the participants have used Android OS before, and 84.6 percent of them have used Android for more than one year. The participants were compensated with a $10 shopping coupon for their participation in the study.

Experiment Procedures. The experiment begins with a brief introduction. We explain to the users and walk them through all of the features that we want them to use. To better mimic the real world scenario, instead of telling users the fact that there are phapps inside and creating unnecessary attention, we only provide a list of tasks for users to accomplish while they are exploring the provided apps, followed by a questionnaire. Each participant is asked to work on the 8 apps randomly and explore them on the assigned Android device. We also asked them to register each corresponding normal apps before our human study and get familiar with the basic functionalities. During the experiment, all apps are used without any interventions or discussions among the participants.

There are five main tasks that participants were asked to complete. Participants need to (1) log in the apps using their credentials; (2) explore functionalities and they can terminate the exploration at any point of the process; (3) give a similarity score between the login pages from phishing apps and the corresponding original ones; (4) distinguish if the current page is from a phapp; (5) give a confidence score about the app related to the deception response given by the phapp.

TABLE 6
The Questions for Participants to Answer

Task	Questions	Likert Scale Score 1-5
T1 T2	Q1: How is the UI design of each app? Q2: Did you notice anything out of ordinary? If yes, specify the app and the problem (e.g., UI layout problem, functionality problem).	Completely unacceptable -Very good
T3	Q3: What's the visual similarity of the two login pages? If has, please write the differences.	Very different -Very similar
T4	Q4: Do you think it is a phishing app according to its login page? Show your confidence.	Very unsure -Very sure
T5	Q5: When you see the responses after clicking "login" button, show your confidence that it is a phishing app?	Very unsure -Very sure

After the experiments, participants are asked to complete a questionnaire in Table 6:

T1&T2. We first ask each participant the overall opinions about each app including the UI design (Q1). Second, they are asked if they notice any weirdness and related details to see if they spot the phapps (Q2).

T3. We then provide login pages from phishing apps and the corresponding original ones of 8 apps to each user to let them score the similarity and point out differences (if any) between the two kinds of pages (Q3). As there are 20 participants, each app in our dataset has been checked by two users to avoid bias.

T4. After they finish answering Q3, we randomly sample 8 different apps (half original, half phishing). We explicitly tell them that there are phapps inside and ask them to check which ones are phapps by only looking at the login pages, and rate their confidence of their choices (Q4) [50].

T5. We then randomly provide 10 response pages from 10 phishing apps after clicking the "login" button, and each of them displays a response of that in Table 2. We ask them for the confidence score about the app regarded as a phishing app (Q5).

Note that all questions have to be answered in the order, listed in Table 6, to stimulate the real environment, where information about the phapps would be unknown to a phishing victim. Different questions are placed to different pages in the survey, so the participants do not know the next questions until they finish answering the current questions. We do not tell participants that there are phapps before Q3, and want to see if they can spot the phapps or any abnormalities by themselves.

6.2 Results of Human Studies

It takes about 35 minutes for each participant to finish the human study, including 16 minutes (2 minutes each) for using the apps, 10 minutes for filling the questions, and 10 minutes to check the image similarity. For all 80 phapps in the experiment, we successfully receive users' usernames and passwords on our hacking server (Nexus 5X, Android 7.1.1). We show the human study results as follows.

TABLE 7
The Results of Phishing App Identification

Metrics	Number	Confidence Scores
TP	26	3.73
FP	24	3.58
TN	56	3.96
FN	54	3.93

Answer to Q1. Most participants hold neutral views on design of UI pages, and there is no significant difference of satisfaction scores of UI design between the original apps (3.85) and phapps (3.47). We interviewed the participants who are not satisfied with the UI design of the whole app, and asked them the reasons for that. Their answers are mainly about two respects: (1) The UI design is too compact, e.g., setting options or other login options (e.g., login with facebook) appear in the page. (2) The UI design is too simple, e.g., only two inputs (username, password) and a "login" button are shown in the login page. But according to our observation, their satisfaction is influenced by whether the app ran well. Those phapps with response messages showing problems about the apps seemed to receive lower satisfaction scores.

Answer to Q2. Among all 160 apps, participants found that 34 of them exhibit some kind of weirdness, 27 of them belong to phapps in our experiments, indicating that users cannot notice any weirdness for a majority of phapps (53/80 = 66.25 percent). We further asked participants what kinds of weirdness they found, and the results show that they regard most of the weirdness (24/27 = 88.9 percent) as functional problems with complaints about "Crash", "Server failed", "Network unavailable", etc. They regard other weirdness (3/27 = 11.1 percent) as UI problems, e.g., lack of features of remembering username with auto-filling in the EditText bar. But none of them raised concerns that this was a phishing app.

Answer to Q3. The results can be seen in the column "Visual Similarity" of Table 3 and the average score is 4.56. As users can only select a score between 1 to 5, it means that most users select 5, i.e., two screenshots are almost perfectly the same. The visual similarity results correlate with our observations given by pixel similarity through computing MSE. Both results verify the quality of the generated login pages.

Answer to Q4. Different from other questions, we now inform participants that there are phapps in this experiment but without telling which apps are phapps. Participants then determine if the app is a phapp or an original app by looking at their login pages, and mark their confidence. The results can be seen in Table 7, where TP represents the number of phapps which are correctly determined, and FP represents the number of original apps which are wrongly determined as phishing. TN represents the number of original apps which are correctly determined, and FP represents the number of phapps which are wrongly determined as benign. Although the number of phapps and the original apps are the same (80 in each) in our experiments, participants regard 50 of them as phapps and the other 110 of them as original apps. In addition, it seems that users have higher confidence in their selection of original apps (average of TN and FN: 3.945) than that of phapps (average of TP and FP: 3.655).

Among 50 login pages which were described as phapps by the participants, 26 (52 percent) of them are right, while 24 (48 percent) of them are wrong. Both TP and FP have similar confidence scores. The probability of correct prediction is almost the same to random guess (50 percent for a binary guess). Similar observations also apply to TN and

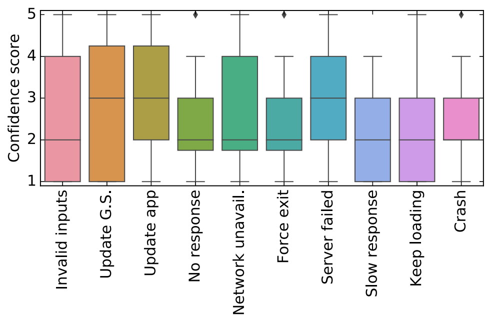


Fig. 11. The confidence of treating apps as phishing apps according to different responses.

FN. These results further demonstrate the effectiveness of our method for phapp generation, as users cannot accurately spot phapps with special attention given after being told that phapps exist.

Answer to Q5. There are 10 kinds of different responses as listed in Table 2. We try to explore which of them are more likely to invoke alarm from an user. The results are shown in Fig. 11. By looking at the mean confidence of different responses, we can see that users are more concerned about "Update Google service", "Update app", and "Server failed". As they are all about downloads, it seems that users are more sensitive to Internet interaction and think that it may bring security risks to their apps. Considering both the mean and lowest confidence values, we find that "Invalid inputs", "Slow response", and "Keep loading" cause fewer concerns. Therefore, when applying our approach in practice, it is better to adopt these responses inside the generated apps. According to the results, these collected response types from real apps achieve different reliability when used in phishing apps. The reason for such random assignment of responses is to defend against the pattern-based detection approaches. Moreover, before the response is shown, the user credentials have already been successfully stolen.

Remark 5. We summarized the key findings based on participants' feedback from the human study. Our phapps successfully masquerade as original apps without raising users' special attention in information leakage. Even in cases when users did raise concerns, we were able to mislead them to believe it was a functional problems as opposed to a security or privacy threat. The login pages of phapps are so similar to the ones of original apps that participants cannot distinguish between them. Responses like "Keep loading" and "Slow response" are more effective in placating users' security concerns than other responses like "Update Google service" and "Update app."

7 Discussion

Limitations of our Approach. (1) Our approach does not fully handle the font family/color of the text extracted from the EditText component, causing a small visual difference if the app uses a special font family. Fortunately, according to the results of the human study, users are insensitive of such differences. (2) Since we generate components with normal attributes, such as a plain background of EditTexts, if the original app uses a colorful image (e.g., photos) as the background of EditTexts, we cannot generate a perfect copy of its UI page. (3) As for targeted UI pages with smaller resolutions, we need to scale the component to an equivalent size to deploy the same phapp on devices with larger resolutions.

Deception Code Generation. As for deception code, we generate responses for each interactive component such as "Button" and "TextView" with component listeners. According to the comprehensive experiments, we notice that *page confusion* plays a more important role than *logic deception* in GUI-squatting attacks. Specifically, in the human study, there is only one person (1/20) who clicked other interactive components first before directly starting the login process. Nevertheless, receiving such responses after clicking other interactive components, they still regarded it as a functional issue (*logic deception*), and then proceeded to the login process. In other words, phapps are able to extract the users' credentials because of the high page similarity (*page confusion*) and the realistic responses encoded the deception code (*logic deception*).

Moreover, compared with repackaging and cloning techniques for phishing attacks, our approach generates mobile phishing apps without any domain-knowledge, and there is no other inputs required except the login page(s) of an original app. Such a light-weight input enables us to generate a phishing app with less complexity but with more reliability of the login pages; thus the deception logic aims to generate the corresponding responses for the interactive components in order to convince users when logging in. There are four main problems to use the original app in addition to login-related pages as inputs when generating phishing apps. (1) First, the original apps are often closed-source, the source code and resource files are unavailable. Even if we are able to obtain it by reverse engineering the original apk file, the process is still affected by the packing and obfuscation techniques as we mentioned in Section 1. (2) Even if the source code of the app is available, the functionalities associated with the components can also be deleted by the technique in [42]. It is difficult to extract the functionalities associated with the components from the source code since many dependencies of the logic code, including third-party libraries and resource files, need to be considered. (3) More sophisticated logic code means more UI pages involvement and maintenance. (4) It is a time-consuming task to reverse engineer and extract functionalities associated with the components.

Mitigation of GUI-Squatting Attack. We introduce the following methods to mitigate our generated phapps. (1) *Static analysis of back-end code.* Due to lack of complete logic code like original apps, phapps may be distinguished from original apps through an in-depth static analysis. Specifically, in this work, apart from the login activity, we integrate some widely-used activities and also build up the relations between them. However, the whole logic is still missing. If defenders can generate the whole picture of phapps at a high level, the general feature or the detectors based on the imbalanced structure of two code branches [55] will help to identify phapps. (2) Taint analysis. Although the technique is able to track user credentials from source to sinks (i.e., server URL), they need to determine whether the remote URL is malicious. For example, one app may contain several URLs linking to other websites apart from the official website related to this app, and it is difficult to determine whether the unofficial URLs are malicious or not. It is also difficult to maintain a comprehensive black-list for comparison or have applications nominate white-listed destinations for authentication. (3) *Relying on the Android app market assessment*. Both the official and third-party Android markets should first analyze similar apps with same or similar UI pages and app names, and further identify whether it is a phishing one. But it is an ineffective way since it relies on a large-scale reference dataset.

8 Related Work

Web Phishing. Gupta et al. [38] summarized that web phishing attacks have two traditional strategies: spoofed emails and fake websites. Spoofed emails induce users to click links in the email and redirect to a malicious website from untrusted servers to extract victims' information. Numerous approaches have been proposed to filter out phishing emails. Fette et al. [34] utilized machine learning to classify the spoofed emails with a high accuracy. CANTINA [77] proposed a content-based approach to detect phishing websites, based on the TF-IDF information retrieval algorithm. Pan et al. [56] examined anomalies in web pages (e.g., the discrepancy between a website's identity) to detect phishing web pages. Fu et al. [36] and Liu et al. [48] used visual similarity comparison to distinguish phishing web pages. DOMAntiPhish [62] leveraged layout similarity information to distinguish malicious and benign web pages. Ma et al. [49] trained a predictive classifier based on the web URLs to identify phishing URLs. However, since attributes in mobile apps are different from those in web pages, these detection techniques are not applicable to mobile systems. In this paper, we focus on phishing attacks under mobile environments.

Mobile Phishing. App-based phishing attack is a major problem on mobile devices [31], [33], [37], [70], and phishing apps are one of the most popular types in malicious apps [25], [26], [27], [30], [32], [69]. Repackaged apps are the most useful technique to perform similarity attacks (spoofing attacks) for mobile phishing [15]. ResDroid [64] leverage new features extracted from core resources and source code to detect repackaged apps; however, phapps do not rely on repackaging techniques. Sun et al. [66] introduced that attackers can analyze the GUI code of the original apps, modify the corresponding layout code, and then add logical code to manipulate the original logic. However, developers can obfuscate or pack their apps to avoid repackaging malware attacks (e.g., repackaging phishing attacks). Meanwhile, this process heavily relies on the attacker's knowledge about the original app code. Bianchi et al. [15] extracted API call sequences via static code analysis to detect phishing apps, however, static analysis is limited to known attack vectors, and many similarity attacks don't require specific API calls. DroidEagle [66] used the similarity of layout tree between official apps and third-party apps to detect mobile phishing apps. Marforio et al. [51], [52] leveraged personalized security indicators as a mechanism to avoid mobile phishing attacks.

MobiFish (AppFish) [73], [74] used OCR techniques to extract texts from the screenshot of a login interface. It identifies the identity from the extracted texts, and compares it with the actual identity from a remote server of mobile apps. If two identities are different, there is a warning presented to users. However, it has two shortcomings: (1) Many login pages do not contain app identities; (2) A white-list of legitimate domains are required, in addition to a database of suspicious applications that needs to first be constructed and continuously updated.

In this paper, we propose GUI-Squatting attacks; however, code obfuscations and packs will not affect the capability of our approach, and knowledge of the original app code is not essential. Moreover, our approach can bypass the state-of-the-art repackaging or clone detection techniques [20]. In addition to similarity attacks, window overlay and task hijacking are common mechanisms to execute mobile phishing attacks [21], [60], [61]. Although we do not focus on these two methods, our approach can also help generate the similar UI pages that can be leveraged by these two attacks. However, these two methods can be detected and mitigated by many cutting-edge detection techniques [15], [59], [60]. A recent defense solution has been proposed in [15] based on GUI-related APIs/permissions. WindowGuard proposed a security model, Android Window Integrity [59], to protect the system against all GUI attacks, including window overlay and task hijacking. But our generated phapps are able to bypass all of these detection techniques successfully.

9 Conclusion

In this paper, we propose a novel approach to automatically generate platform-independent phishing apps, to enable a powerful and large-scale phishing attack (GUI-Squatting attack) on different categories of apps within 3 seconds. Our human study demonstrates the effectiveness of our generated phishing apps which successfully steal users' information imperceptibly. Additionally, the generated apps can successfully bypass the state-of-the-art detection techniques. Finally, by discussing methods to mitigate our generated apps, we thereby assist security defenders to further explore and understand the characteristics of new mobile phishing apps.

Acknowledgments

We appreciate the constructive comments from reviewers. This work is partially supported by National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001) and the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001). Lihua Xu is supported in part by NSFC Grant 61502170, the Science and Technology Commission of Shanghai Municipality Grants 18511103802. The authors would like to thank Nvidia for their GPU support.

References

[1] Alleged 'Nazi' Android FBI ransomware mastermind arrested in Russia, 2015. [Online]. Available: https://www.forbes.com/
[2] Android packers, 2018. [Online]. Available: https://www.blackhat.com/docs/asia-15/materials/asia-15-Park-We-Can-Still-Crack-You-General-Unpacking-Method-For-Android-Packer-No-Root.pdf
[3] Canny edge detection, 2018. [Online]. Available: https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html

[4] Dilatation edge, 2018. [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

[5] GDPR, 2018. [Online]. Available: https://www.tripwire.com/solutions/compliance-solutions/gdpr/

[6] One-way analysis of variance, 2018. [Online]. Available: https://en.wikipedia.org/wiki/One-way_analysis_of_variance

[7] OpenCV, 2018. [Online]. Available: https://opencv.org/

[8] Optical character recognition, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Optical_character_recognition

[9] Phishers leveraging GDPR-Themed scam emails to steal users' information, 2018. [Online]. Available: https://securityboulevard.com

[10] Squatting attack, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Squatting_attack

[11] Tesseract, 2018. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[12] UIAutomator, 2018. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[13] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2014~~, pp. 23–26~~.

[14] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," 2017, *arXiv: 1705.07962*.

[15] 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 *Proc. IEEE Symp. Security Privacy*, 2015, pp. 931–948.

[16] C. Bravo-Lillo, L. F. Cranor, J. Downs, and S. Komanduri, "Bridging the gap in computer security warnings: A mental model approach," *IEEE Security Privacy*, vol. 9, no. 2, pp. 18–26, Mar./Apr. 2011.

[17] CAPEC, "Mobile phishing," 2017. [Online]. Available: https://capec.mitre.org/data/definitions/164.html

[18] O. Chapelle, P. Haffner, and V. N. Vapnik, "Support vector machines for histogram-based image classification," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 1055–1064, Sept. 1999.

[19] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 665–676.

[20] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[21] 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 *Proc. 23rd USENIX Conf. USENIX Security Symp.*, 2014, pp. 1037–1052.

[22] S. Chen et al., "StoryDroid: Automated generation of storyboard for Android apps," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 596–607.

[23] S. Chen et al., "AUSERA: Large-scale automated security risk assessment of global mobile banking apps," 2018, *arXiv: 1805.05236*.

[24] S. Chen et al., "Are mobile banking apps secure? What can be improved?" in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2018, pp. 797–802.

[25] S. Chen et al.,"Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Comput. Security*, vol. 73, pp. 326–344, 2018.

[26] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A streaminglized machine learning-based system for detecting android malware," in *Proc. 11th ACM Asia Conf. Comput. Commun. Security*, 2016, pp. 377–388.

[27] S. Chen, M. Xue, and L. Xu, "Towards adversarial detection of mobile malware: Poster," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, 2016, pp. 415–416.

[28] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on Android apps via traffic analysis," in *Proc. 5th ACM Conf. Data Appl. Security Privacy*, 2015, pp. 297–304.

[29] S. Dong et al., "Understanding Android obfuscation techniques: A large-scale investigation in the wild," 2018, *arXiv: 1801.01633*.

[30] L. Fan, M. Xue, S. Chen, L. Xu, and H. Zhu, "Poster: Accuracy vs. time cost: Detecting Android malware through pareto ensemble pruning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 1748–1750.

[31] A. P. Felt and D. Wagner, *Phishing on Mobile Devices. na*, 2011.

[32] R. Feng et al., "MobiDroid: A performance-sensitive malware detection system on mobile platform," in *Proc. 24th Int. Conf. Eng. Complex Comput. Syst.*, 2019, pp. 61–70.

[33] E. Fernandes et al., "Android UI deception revisited: Attacks and defenses," in *Proc. Int. Conf. Financial Cryptography Data Security*, 2016, pp. 41–59.

[34] I. Fette, N. Sadeh, and A. Tomasic, "Learning to detect phishing Emails," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 649–656.

[35] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and Dagger: From two permissions to complete control of the UI feedback loop," in *Proc. IEEE Symp. Security Privacy*, 2017, pp. 1041–1057.

[36] A. Y. Fu, L. Wenyin, and X. Deng, "Detecting phishing web pages with visual similarity assessment based on earth mover's distance (EMD)," *IEEE Trans. Depend. Secure Comput.*, vol. 3, no. 4, pp. 301–311, Oct.–Dec. 2006.

[37] D. Goel and A. K. Jain, "Mobile phishing attacks and defence mechanisms: State of art and open research challenges," *Comput. Security*, vol. 73, pp. 519–544, 2018.

[38] B. Gupta, N. A. Arachchilage, and K. E. Psannis, "Defending against phishing attacks: Taxonomy of methods, current issues and future directions," *Telecommun. Syst.*, vol. 67, no. 2, pp. 247–267, 2018.

[39] G. Ho, A. S. M. Javed, V. Paxson, and D. Wagner, "Detecting credential spearphishing attacks in enterprise settings," in *Proc. 26rd USENIX Security Symp.*, 2017, pp. 469–485.

[40] J. Hong, "The state of phishing attacks," *Commun. ACM*, vol. 55, no. 1, pp. 74–81, 2012.

[41] M.-K. Hu, "Visual pattern recognition by moment invariants," *IRE Trans. Inf. Theory*, vol. IT-8, no. 2, pp. 179–187, Feb. 1962.

[42] J. Huang, Y. Aafer, D. Perry, X. Zhang, and C. Tian, "Ui driven Android application reduction," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 286–296.

[43] INCAPSULA, "Phishing attacks," 2018. [Online]. Available: https://www.incapsula.com/web-application-security/

[44] J. Jiang, S. Wen, S. Yu, Y. Xiang, and W. Zhou, "Identifying propagation sources in networks: State-of-the-art and comparative studies," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1, pp. 465–481, 1Q 2017.

[45] K. Krombholz, H. Hobel, M. Huber, and E. Weippl, "Advanced social engineering attacks," *J. Inf. Security Appl.*, vol. 22, pp. 113–122, 2015.

[46] C.-C. Lin, H. Li, X.-Y. Zhou, and X. Wang, "Screenmilker: How to milk your Android screen for secrets," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2014.

[47] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 2, pp. 1397–1417, 2Q 2018.

[48] W. Liu, X. Deng, G. Huang, and A. Y. Fu, "An antiphishing strategy based on visual similarity assessment," *IEEE Internet Comput.*, vol. 10, no. 2, pp. 58–65, Mar./Apr. 2006.

[49] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious URLs," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 1245–1254.

[50] L. Malisa, K. Kostiainen, and S. Capkun, "Detecting mobile application spoofing attacks by leveraging user visual similarity perception," in *Proc. 7th ACM Conf. Data Appl. Security Privacy*, 2017, pp. 289–300.

[51] C. Marforio, R. Jayaram Masti, C. Soriente, K. Kostiainen, and S. Čapkun, "Evaluation of personalized security indicators as an anti-phishing mechanism for smartphone applications," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2016, pp. 540–551.

[52] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Capkun, "Hardened setup of personalized security indicators to counter phishing attacks in mobile banking," in *Proc. 6th Workshop Security Privacy Smartphones Mobile Devices*, 2016, pp. 83–92.

[53] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with REMAUI," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 248–259.

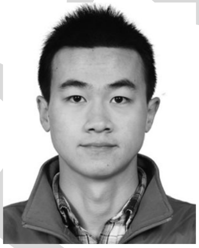
[54] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android intercomponent communication analysis," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 77–88.

[55] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2017.

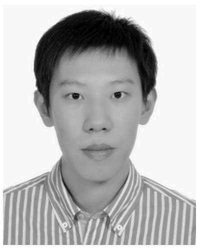
[56] Y. Pan and X. Ding, "Anomaly based web phishing page detection," in *Proc. 22nd Annu. Comput. Security Appl. Conf.*, 2006, pp. 381–392.
[57] PHISHING.org, "History of phishing," 2017. [Online]. Available: http://www.phishing.org/history-of-phishing
[58] PHISHING.org, "Phishing techniques," 2017. [Online]. Available: http://www.phishing.org/phishing-techniques
[59] C. Ren, P. Liu, and S. Zhu, "WindowGuard: Systematic protection of GUI security in Android," in *Proc. Annu. Symp. Netw. Distrib. Syst. Security*, 2017.
[60] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in Android," in *Proc. 24th USENIX Conf. Security Symp.*, 2015, pp. 945–959.
[61] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proc. 22nd USENIX Conf. Security Symp.*, 2013, pp. 97–112.
[62] A. P. Rosiello, E. Kirda, F. Ferrandi, and C. Kruegel, "A layout-similarity-based approach for detecting phishing pages," in *Proc. 3rd Int. Conf. Security Privacy Commun. Netw. Workshops*, 2007, pp. 454–463.
[63] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *Proc. IEEE Symp. Security Privacy*, 2007, pp. 51–65.
[64] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Security Appl. Conf.*, 2014, pp. 56–65.
[65] T. Su et al., "Guided, stochastic model-based GUI testing of Android apps," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 245–256.
[66] M. Sun, M. Li, and J. Lui, "DroidEagle: Seamless detection of visually similar Android apps," in *Proc. 8th ACM Conf. Security Privacy Wireless Mobile Netw.*, 2015, Art. no. 9.
[67] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1744–1772, Apr./Jun. 2019.
[68] Y. Sun, A. K. Wong, and M. S. Kamel, "Classification of imbalanced data: A review," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 23, pp. 687–719, 2009.
[69] C. Tang et al., "A large-scale empirical study on industrial fake apps," in *Proc. 41st Int. Conf. Softw. Eng.: Softw. Eng. Practice*, 2019, pp. 183–192.
[70] K. Thomas et al., "Data breaches, phishing, or malware?: Understanding the risks of stolen credentials," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 1421–1434.
[71] wandera, "Mobile data report: Focus on phishing," 2017. [Online]. Available: http://go.wandera.com/rs/988-EGM-040/images/Phishing%20%282%29.pdf
[72] S. Wen, M. S. Haghighi, C. Chen, Y. Xiang, W. Zhou, and W. Jia, "A sword with two edges: Propagation studies on both positive and negative information in online social networks," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 640–653, Mar. 2015.
[73] L. Wu, X. Du, and J. Wu, "MobiFish: A lightweight anti-phishing scheme for mobile phones," in *Proc. 23rd Int. Conf. Comput. Commun. Netw.*, 2014, pp. 1–8.
[74] L. Wu, X. Du, and J. Wu, "Effective defense schemes for phishing attacks on mobile computing platforms," *IEEE Trans. Veh. Technol.*, vol. 65, no. 8, pp. 6678–6691, Aug. 2016.
[75] T. Wu, S. Wen, Y. Xiang, and W. Zhou, "Twitter spam detection: Survey of new approaches and comparative study," *Comput. Security*, vol. 76, pp. 265–284, 2018.
[76] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 104–117, Jan. 2013.
[77] Y. Zhang, J. I. Hong, and L. F. Cranor, "Cantina: A content-based approach to detecting phishing web sites," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 639–648.
[78] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proc. 2nd ACM Conf. Data Appl. Security Privacy*, 2012, pp. 317–326.

Sen Chen received the PhD degree in computer science from the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in June 2019. Currently, he is a postdoctoral research fellow in School of Computer Science and Engineering, Nanyang Technological University, Singapore. Previously, he was a visiting scholar of Cyber Security Lab, SCSE, NTU from October 2016 to June 2019. His research interests include mobile security and testing, AI security and testing, and big data. He has published broadly in top-tier security and software engineering venues, including CCS, ASIACCS, ICSE, FSE, ASE, and TSE. For more information, visit https://sen-chen.github.io/

Lingling Fan received the BEng and PhD degrees in computer science from East China Normal University, Shanghai, China, in June 2014 and June 2019, respectively, and now she is a research fellow in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Her research interests include program analysis and testing, android application analysis and testing, and model checking. She got an ACM SIGSOFT Distinguished Paper Award at ICSE 2018. For more information, visit https://lingling-fan.github.io/



Chunyang Chen received the bachelor's degree from the Beijing University of Posts and Telecommunications, China, June 2014, and the PhD degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He is currently a lecturer (assistant professor) in the Faculty of Information Technology, Monash University, Australia. His research interests include mining software repositories, text mining, deep learning, and human computer interaction.



Minhui Xue is currently a lecturer (assistant professor) with the School of Computer Science, University of Adelaide. He is also an honorary lecturer with Macquarie University. His current research interests include machine learning security and privacy, system and software security, and Internet measurement. He published widely in top security and software engineering conferences, including IEEE S&P, NDSS, ACM IMC, PETS, IEEE/ACM FSE, IEEE/ACM ASE, and ACM ISSTA. He is the recipient of the ACM SIGSOFT Distinguished Paper Award and IEEE Best Paper Award, and his work has been featured in the mainstream press, including The New York Times and Science Daily.

Yang Liu received the bachelor of computing (Honours) degree in the National University of Singapore (NUS), in 2005, and started his post doctoral work in NUS, MIT, and SUTD, in 2010. In 2011, he was awarded the temasek research fellowship at NUS to be the Principal Investigator in the area of Cyber Security. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang assistant professor. He is currently an associate professor and the director of the cybersecurity lab in NTU. He specializes in software verification, security, and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. His work led to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT). By now, he has more than 200 publications and 6 best paper awards in top tier conferences and journals. With more than 20 million Singapore dollar funding support, he is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.

Lihua Xu received the PhD degree in computer science from the University of California, Irvine, June 2009. She is currently an associate professor at New York University Shanghai and East China Normal University, China. Before joining New York University Shanghai as part of its overseas talent introduction program, in 2012, she held tenure-track assistant professor position at Rochester Institute of Technology. Her current research focuses on software engineering, automated software testing, and mobile security.