Detecting Mobile Application Spoofing Attacks by Leveraging User Visual Similarity Perception

Luka Malisa
Institute of Information Security
ETH Zurich
malisal@inf.ethz.ch

Kari Kostiainen
Institute of Information Security
ETH Zurich
kari.kostiainen@inf.ethz.ch

Srdjan Capkun
Institute of Information Security
ETH Zurich
capkuns@inf.ethz.ch

***Abstract*—Mobile application spoofing is an attack where a malicious mobile application mimics the visual appearance of another one. A common example of mobile application spoofing is a phishing attack where the adversary tricks the user into revealing her password to a malicious application that resembles the legitimate one. In this paper, we propose a novel spoofing detection approach, tailored to the protection of mobile app login screens, using screenshot extraction and visual similarity comparison. We use deception rate as a novel similarity metric for measuring how likely the user is to consider a potential spoofing app as one of the protected applications. We conducted a large-scale online study where participants evaluated spoofing samples of popular mobile app login screens, and used the study results to implement and train a detection system that accurately estimates deception rate. Our collaborative detection model provides efficient detection with low overhead.**

I. Introduction

Mobile application spoofing is an attack where a malicious mobile application mimics the visual appearance of another one. The goal of the adversary is to trick the user into believing that she is interacting with a genuine application while she interacts with one controlled by the adversary. If such an attack is successful, the integrity of what the user sees as well as the confidentiality of what she inputs into the system can be violated by the adversary. This includes login credentials, personal details that users typically provide to applications, as well as the decisions that they make based on the information provided by the applications.

A common example of mobile application spoofing is a phishing attack where the adversary tricks the user into revealing her password, or similar login credentials, to a malicious application that resembles the legitimate app. Several mobile application phishing attacks have been seen in the wild [20, 32, 37]. For example, a recent mobile banking spoofing application infected 350,000 Android devices and caused significant financial losses [14]. More sophisticated attack vectors are described in recent research [4, 7, 13, 36].

The problem of spoofing has been studied extensively in the context of phishing websites [1, 2, 11, 16, 17]. Web applications run in browsers that provide visual cues, such as URL bars, SSL lock icons and security skins [10], that can help the user to authenticate the currently displayed website. Similar application identification cues are not available on modern mobile platforms, where a running application commonly controls the whole visible screen. The user can see a familiar user interface, but the interface could be drawn by a malicious spoofing application — the user is unable to authenticate the contents of the screen.

Security indicators for smartphone platforms have been proposed [12, 30], but their effectiveness relies on user alertness and they typically require either hardware modifications to the phone or a part of the screen to be made unavailable to the apps. Application-specific personalized indicators [36] require no platform changes, but increase the application setup effort. Furthermore, static code analysis has been proposed to detect API call sequences that enable spoofing attacks [4]. However, code analysis is limited to known attack vectors and many spoofing attacks do not require any specific API calls, as they only draw on the screen.

We propose a novel spoofing detection approach that is tailored to the protection of mobile app *login screens* using visual similarity. Our system periodically grabs screenshots on the user's device and extracts from them visual features, with respect to *reference values* — the login screens of legitimate apps (running on the same device) that our system protects. If a screenshot demonstrates high similarity to one of the reference values, we label the currently running app potentially malicious, and report it to the platform provider or warn the user. As our system examines screenshots, it is agnostic to the spoofing screen implementation, in contrast to approaches that examine screen similarity through code analysis.

In order to label spoofing apps accurately, our system needs to understand what kind of attacks are successful in reality, i.e., how much and what kind of visual similarity the two compared applications should have, so that the user would mistake the spoofing app as the legitimate one and fall for the attack. We capture this notion as a novel similarity metric called *deception rate*. For example, when deception rate is 20%, one fifth of the users are estimated to consider the spoofing app genuine and enter their login credentials into it. Deception rate is a conceptually different similarity metric from the ones previously proposed for similarity analysis of phishing websites. These works extract structural [3, 18, 26, 38, 39] as well as visual [8, 15, 22] similarity features and combine them into a similarity score that alone is not expressive, but enables comparison to known attack samples [18, 23]. While the previously proposed metrics essentially tell how similar

the spoofing app is to one of the known attacks, our metric determines how likely the attack is to succeed. Deception rate can be seen as a risk measure and we consider it a powerful new way to address spoofing attacks, especially in cases where a large dataset of known attacks is not available.

Our system requires a good understanding of how users remember mobile app user interfaces and how they react to perceived changes within them. Change perception has been studied extensively in general [24, 25, 31], but not in the context of mobile applications. We therefore conducted a large-scale online study on mobile app similarity perception. We used a crowd sourcing platform to carry out a series of online surveys where approximately 5,400 study participants evaluated more than 34,000 spoofing screenshot samples. These samples included modified versions of Facebook, Skype and Twitter login screens where we changed visual features such as the color or the logo.

We found that, while some users were alarmed by the login screen modifications, others attributed the changes to either a program bug or a new feature. For most of the experimented visual modifications we noticed a systematic user behavior: the more a visual property is changed, the less likely the users are to consider the app genuine.

We used the results of our user study to train our system using common supervised learning techniques. We also developed novel visual feature extraction and matching techniques. Our system shows robust screenshot processing and good deception rate accuracy (6–13% error margin), i.e., our system can precisely determine when an application is so similar to one of the protected login screens that the user is in risk of falling for spoofing. No previous visual similarity comparison scheme gives the same security property.

Additionally, we describe a novel collaborative detection model where multiple devices take part in screenshot extraction. We show that runtime detection is effective with very little system overhead (e.g., 1%). Our results can also be useful to other spoofing detection systems, as they give insight into how users perceive visual change.

Contributions. To summarize, we make the following contributions:

- We propose a novel approach for detecting mobile application spoofing attacks using *visual similarity* and introduce *deception rate* as a novel similarity metric.
- We conducted a *large-scale user study* on perception of visual modifications in mobile application login screens.
- Leveraging our study results, we implemented and trained a runtime *spoofing detection system* for Android.
- We developed novel and robust *visual feature extraction* techniques.

The rest of this paper is organized as follows. In Section II we explain the problem of mobile application spoofing. Section III introduces our approach and Section IV describes our user study. In Section V we describe the spoofing detection system. We evaluate its accuracy and performance in Section VI and analyze it in Section VII. Section VIII reviews related work, and we conclude in Section IX.

II. Problem Statement

In mobile application spoofing, the goal of the adversary is to either violate the integrity of the information displayed to the user or the confidentiality of the user input. Application phishing is an example of a spoofing attack where the goal of the adversary is to steal confidential user data. The adversary tricks the user into disclosing her login credentials to a malicious app with a login screen resembling the legitimate one. A malicious stock market application that is similar to the legitimate one, but shows fake stock market values, is an example of an attack where the adversary violates the integrity of the visual information displayed to the user. In doing so, the adversary affects the user's future stock market decisions. In what follows, we review different ways of implementing application spoofing attacks.

The simplest way to implement a spoofing attack is a repackaged or otherwise cloned application. To the user, the application appears identical to the target application, except for subtle visual cues such as a different developer name. Application repackaging has become a prevalent problem in the Android ecosystem, and the majority of Android malware is distributed using repackaging [6, 40].

In a more sophisticated variant of mobile application spoofing, the malicious app masquerades as a legitimate application, such as a game. The user starts the game and the malicious application continues running in the background from where it monitors the system state, such as the list of currently running applications. When the user starts the target application, the malicious application activates itself on the foreground and shows a spoofing screen that is similar, or exactly the same, to the one of the target app. On Android, background activation is possible with commonly used permissions and system APIs [4, 13]. Background attacks are difficult to notice for the user. While API call sequences that enable background attacks can be detected using code analysis [4], automated detection is complicated by the fact that the same APIs are frequently used by benign apps.

A malicious application can also present a button to share information via another app. Instead of forwarding the user to the suggested target app, the button triggers a spoofing screen within the same, malicious application [13]. Fake forwarding requires no specific permissions or API calls which makes such attack vectors difficult to discover using code analysis. Further spoofing attack vectors are discussed in [4].

Mobile application spoofing attacks are a recent mobile malware type and a large corpus of known spoofing apps is not yet available. However, serious attacks have already taken place. The Svpeng malware infected 350,000 Android devices and caused financial loss worth of nearly one million USD [14]. The malware presents a spoofed credit card entry dialog when the user starts the Google Play application and monitors startup of targeted mobile banking applications to mount spoofing attacks on their login screens. As spoofing detection using traditional code analysis techniques has inherent limitations and many spoofing attacks are virtually impossible for the

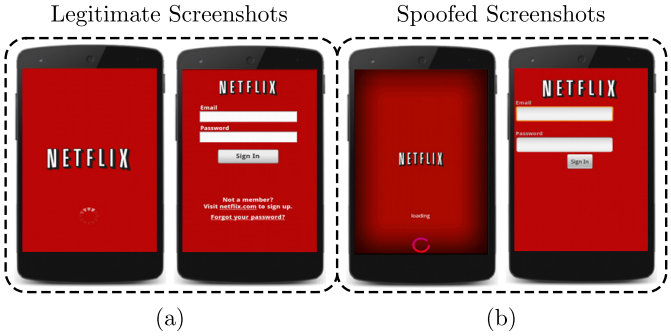

Fig. 1: Spoofing app example. (a) Shows the legitimate Netflix app and (b) the Android.Fakeneflic malware [33]. The spoofed user interface includes subtle visual modifications.

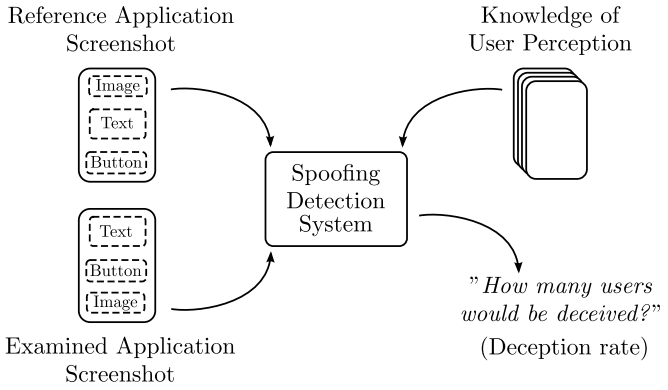

Fig. 2: Approach overview. The spoofing detection system takes as inputs screenshots of a reference app and an examined app. Based on these screenshots and knowledge on mobile application user perception, the system estimates deception rate for the examined app.

users to notice, the exact extent of the problem remains largely unknown. Due to the already seen serious attacks, we believe it is useful to seek novel ways to address the problem of mobile application spoofing.

The problem of mobile application spoofing has many similarities to the one of web phishing. The majority of the existing web phishing detection schemes [3, 18, 26, 38, 39] train a detection system using a large dataset of known phishing websites. As a similar dataset is not available for mobile apps, these approaches are not directly applicable to mobile app spoofing detection. We also argue that the specific nature of mobile applications benefits from a customized approach. In the next section, we introduce a novel detection approach that is tailored to mobile app login screens and draws from user perception. The focus of this work is on mobile app spoofing and web phishing is explicitly out of scope.

III. Our Approach

In this section, we first describe the rationale behind our approach and introduce deception rate as a similarity metric. We then describe how this approach is instantiated into a case study on login screen spoofing detection. Finally, we describe our attacker model.

A. Visual Similarity and Deception Rate

The problem of application spoofing can be approached in multiple ways. First, code analysis has been proposed to detect API call sequences that enable spoofing attacks [4]. However, code analysis is limited to known attack vectors and cannot address spoofing attacks that do not require specific API calls (e.g., fake forwarding). A second approach is to analyze the application code or website DOM trees to identify applications with *structural* user interface similarity [3, 18, 26, 38, 39]. A limitation of this approach is that the adversary can complicate code analysis, e.g., by constructing the user interface pixel by pixel. Third, the mobile platform can be enhanced with security indicators [12, 30]. However, indicator verification imposes a cognitive load on the user and their deployment typically requires either part of the screen to be made unavailable to the applications or hardware modifications to the device. Application-specific personalized indicators [36] can be deployed without platform changes, but their configuration increases user effort during application setup.

In this paper, we focus on a different approach and study the detection of spoofing attacks based on their *visual similarity*. Previously, visual similarity analysis has been proposed for detection of phishing websites [15, 35, 38]. Designing an effective spoofing detection system based on visual similarity analysis is not an easy task, and we illustrate the challenges by providing two straw-man solutions.

The first straw-man solution is to look for mobile apps that have exactly the same visual appearance. To avoid such detection, the adversary can create a slightly modified version of the spoofing screen. For example, small changes in the login screen element positions are hard to notice and are unlikely to retain the user from entering her login credentials. Consequently, this approach would fail to catch many spoofing attacks. Such visually modified attacks are a realistic threat and examples have been observed in the wild. The Android.Fakeneflic malware [33], discovered on Google's Android market, impersonated the legitimate Netflix application by creating a user interface, including the login screen, that visually differed from the legitimate one (Figure 1). Such attacks would not be detected by a simple comparison scheme that looks for an exact visual match.

The second straw-man solution is to flag all applications that have some visual similarity to a reference application, with regards to a well-known similarity metric (e.g., pixel difference). However, the chosen metric may not capture the visual properties that users consider relevant. As our user-study shows, many visually different screens are perceived by users as similar. Therefore, finding an accurate similarity threshold can be challenging, especially without a large corpus of known

attacks and their success rates. For accurate mobile application spoofing detection, more sophisticated techniques are needed.

In this paper we take a different approach and design a spoofing detection system that estimates how many users would fall for a spoofing attack. We use *deception rate* as a novel similarity metric that represents the estimated attack success rate. Given two screenshots, one of the examined app and one of the protected reference app, our system (Figure 2) estimates the percentage of users that would mistakenly identify the examined app as the reference app (deception rate). This estimation is done by leveraging results from a study on how users perceive visual similarity on mobile app user interfaces. The deception rate can be seen as a risk measure that allows our system to determine if the examined application should be flagged as a potential spoofing app. An example policy is to flag any application where the deception rate exceeds a threshold.

Deception rate is a conceptually different similarity metric from the ones previously proposed for similarity analysis of phishing websites. These works extract structural [3, 18, 26, 38, 39] as well as visual [8, 15, 22] similarity features and combine them into a similarity score that alone is not expressive, but enables comparison to known attack samples [18, 23]. The extracted features can also be fed into a system that is trained using known malicious sites [15, 35, 38]. Such similarity metrics are interpreted with respect to known attacks, and may not be effective in detecting spoofing attacks with an appearance different from the ones previously seen.

Deception rate has different semantics, as it captures the *perceived similarity* of spoofing screens. For example, a mobile app login screen where elements have been reordered may have different visual features but, as our user study shows, is perceived similarly by many users. Deception rate estimates how many people would mistakenly identify the spoofing app as the genuine one (risk measure), and contrary to the previous similarity metrics, this metric is applicable also in scenarios where a large dataset of known spoofing samples are not available. Realization of such a system requires a good understanding of what type of mobile application interfaces users perceive as similar and what type of visual modifications users are likely to notice. This motivates our user study, the results of which we describe in Section IV.

B. Case Study: Login Screen Spoofing

In this work, we focus on spoofing attacks against mobile application login screens, as they are the most security-sensitive ones in many applications. We examined the login screens of 230 different apps and found that they all follow a similar structure. The login screen is a composition of three main elements: (1) the logo, (2) the username and password input fields, and (3) the login button. Furthermore, the login screen can have additional, visually less salient elements, such as a link to request a forgotten password or register a new account. In some mobile applications, the login screen is the first (initial) screen the user sees. Other apps distribute these elements across two screens: the first screen contains the logo,

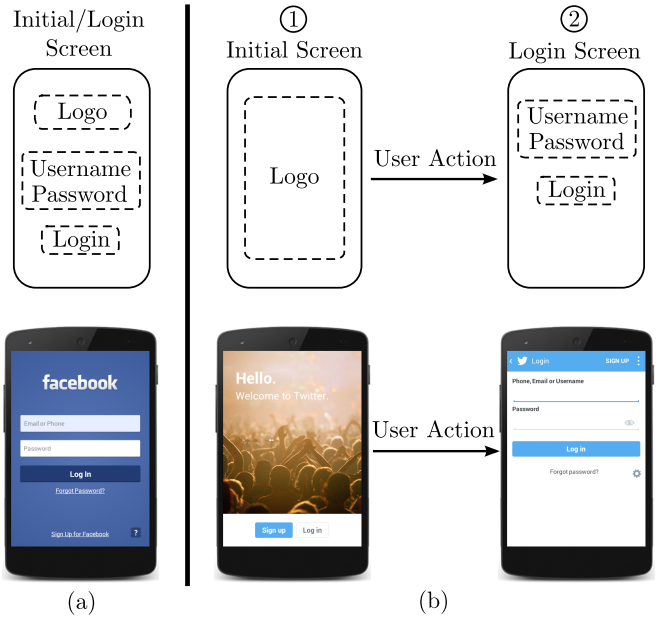

Fig. 3: Model for mobile application login screens. The login screen has three main elements: logo, username and password input fields, and login button. The login functionality is either (a) standalone or (b) distributed.

or a similar visual identifier, as well as a button that leads to the login screen, where the rest of the main elements reside.

The common structure of mobile app login screens enables us to model them, and their simple and clean designs provide a good opportunity to experiment on user perception. Mobile app login screens have fewer modification dimensions to explore, as compared to more complex user interfaces, such as websites. Throughout this work we use the login screen model illustrated in Figure 3 that captures both standalone and distributed logins screens. Out of the 230 apps we examined, 136 had a standalone login screen, while 94 had a distributed one. All apps conformed to our model. We experiment on user perception with respect to this model, as the adversary has an incentive to create spoofing screens that resemble the legitimate login screen. Our study confirms this assumption.

C. Attacker Model

We assume a strong attacker capable of creating arbitrary spoofed login screens, including login screens that deviate from our model (see Section VII-C). We distinguish between two spoofing attack scenarios regarding user expectations and goals. In all the spoofing attacks listed in Section II, the user's intent is to access the targeted application. This implies that the user expects to see a familiar user interface and has an incentive to log in. The adversary could also present a spoofing screen unexpectedly, when the user is not accessing the target application. In such cases, the user has no intent, nor similar incentive, to log in. We focus on the first category, as we consider such attacks more severe and more likely to succeed.

We assume an attacker that controls a malicious spoofing app running on the user smartphone. Besides the spoofing

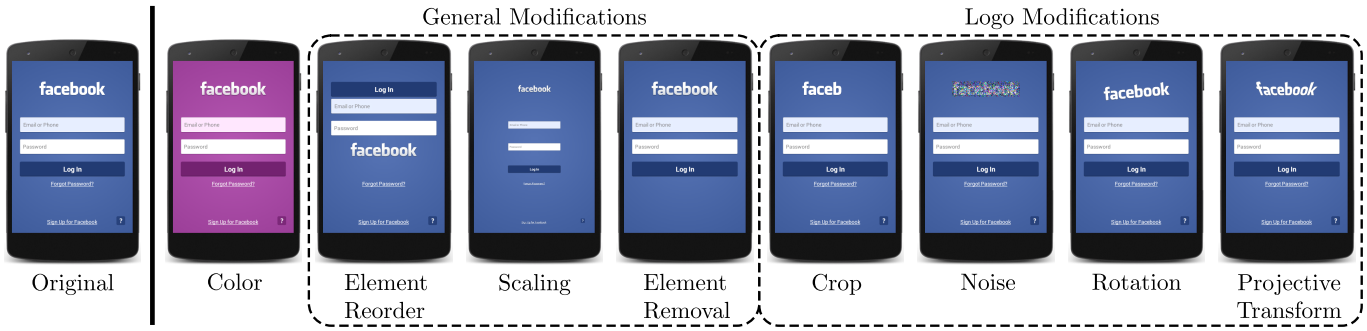

Fig. 4: Examples of Facebook login screen spoofing samples. The original login screen is shown on the left. We show an example of each type of visual modification we performed: color, general modifications, and logo modifications.

screen, the attacker-controlled app appears to the user as entirely benign (e.g., a game). The attacker can construct the spoofing screen statically (e.g., using Android XML manifest files) or dynamically (e.g., by creating widgets at runtime). In both cases, the operating system is aware of the created element tree, a structure similar to DOM trees in websites. The attacker has also the choice of drawing the screen pixel by pixel, in which case the operating system sees only one element, a displayed picture. The attacker can also exploit the properties of human image perception. For example, the attacker can display half of the spoofed screen in one frame, and the other half in the subsequent frame. The human eye would average the input signal and perceive the complete spoofing screen.

IV. Change Perception User Study

Visual perception has been studied extensively in general, and prior studies have demonstrated that users are surprisingly poor at noticing changes in images that are shown in a consecutive order (change blindness) [25, 31]. While such studies give us an intuition on how users might notice, or fail to notice, different login screen modifications, the results are too generic to be directly applied to the spoofing detection system outlined above. To the best of our knowledge, no previous studies on user perception of visual changes in mobile application user interfaces exist.

We conducted a large-scale online study on the similarity perception of mobile app login screens, and the purpose of this study was three-fold. We wanted to (1) understand the effect of different types of visual login screen modifications, (2) gather training data for the spoofing detection system, and (3) gain insights that could aid us in the design of our system. The study was performed as online surveys on the crowd sourcing platform CrowdFlower. The platform allows creation of online jobs that human participants perform in return of a small payment. In each survey, the participants evaluated a single screenshot of a mobile app login screen by answering questions (see Section IV-C).

We first performed an initial study, where we experimented with visual modifications on the Android Facebook application. We chose Facebook, as it is a widely used and well-remembered application. After that, we carried out follow-up studies where we tested similar visual modifications on Skype and Twitter apps, as well as combinations of visual changes. Below, we describe the Facebook study in detail and summarize the results of the follow-up studies. We did not collect any private information about our study participants. The ethical board of our institution reviewed and approved our user study.

A. Sample Generation

A *sample* is a screenshot image presented to a study participant for evaluation. We created eight datasets of Facebook login screens samples, and in each dataset we modified a single visual property. The purpose of these datasets was to evaluate how users perceive different types of visual changes as well as to provide training data for the spoofing detection system (Section V). Here, we describe each performed modification:

- *Color modification.* We modified the hue of the application login screen. The hue change affects the color of all elements on the login screen and the dataset contained samples representing uniform hue changes over the entire hue range.
- *General modifications.* We performed three general modifications on the login screen elements. (1) We reordered the elements, and Figure 4 (Element Reorder) shows an example where the logo and the login button exchanged places. (2) We scaled down the size of the elements. We did not increase the size of the elements, as the username and the password fields are typically full width of the screen. (3) We removed any extra elements from the login screen. Figure 4 (Element Removal) depicts a sample where the links to request a forgotten password and register a new account have been removed.
- *Logo modifications.* We performed four modifications on the logo. (1) We cropped the logo to different sizes, taking the rightmost part of the logo out. (2) We added noise of different intensity, (3) rotated the logo both clockwise and counterclockwise, and (4) performed projective transformations on the logo.

We created such synthetic spoofing samples as no extensive mobile spoofing app dataset is available. While the chosen

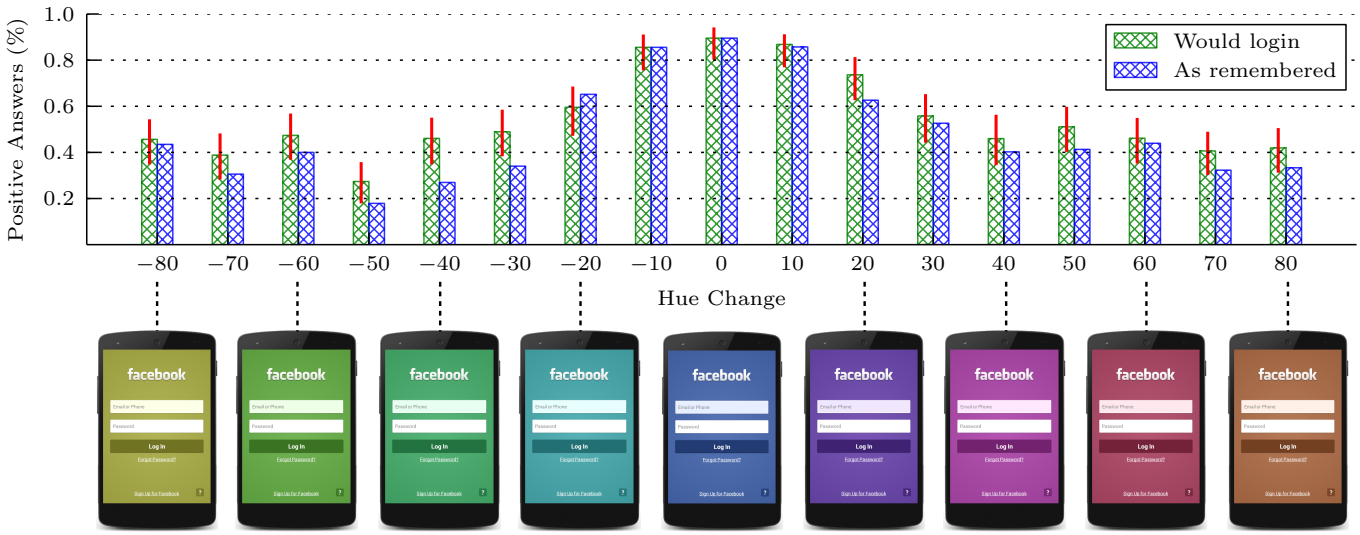

Fig. 5: Color modification results. We illustrate the percentages of users that perceived a Facebook login screen sample with modified color as genuine (*as-remembered rate*) and would login to the application if such a screen is shown (*login rate*). Color has a significant effect on both rates.

modifications cover known spoofing attacks (e.g., the Android.Fakeneflic malware shown in Figure 1), the goal of our work is not to optimize the system for the detection of known attacks, but rather to implement a system that is able to detect also previously unseen spoofing screens. The sample set could be extended in many ways (e.g., replace logo with text). We note that the list of possible visual modifications is practically endless and a single user study cannot cover all of them.

B. Participant Recruitment

We recruited test participants by publishing survey jobs on the crowd sourcing platform. An example survey had a title *"Android Application Familiarity"* and the description of the survey was *"How familiar are you with the Facebook Android application?"*. We specified in the survey description that the participant should be an active user of the tested application and defined a reward of 10 cents (USD) for each completed survey. We recruited 100 study participants for each sample, accepted participants globally, and required the participants to be at least 18 years old. The study participants were allowed to evaluate multiple samples from different datasets, but only one sample from each dataset. For example, a study participant could complete two surveys: one where we evaluated color modification samples and another regarding logo crop, but the same participant could not complete multiple surveys on color modification. In total 2,910 unique participants evaluated 5,900 Facebook samples. Table I provides the study statistics.

C. Study Tasks

Each survey included 12 to 16 questions. We asked preliminary questions on participant demographics, tested application usage frequency, and a control question with a known correct answer. We showed the study participant a sample login screen screenshot and asked the participant to evaluate it using the following questions: *"Is this screen (smart phone screenshot) the Facebook login screen as you remember it?"* and *"If you would see this screen, would you login with your real Facebook password?"*. We provided *Yes* and *No* reply alternatives on both questions. Using the percentage of *Yes* answers, we compute *as-remembered rate* and *login rate* for each evaluated sample. We also asked the participants to comment on their reason to log in or retain from logging in. A listing of all survey questions is available online: http://goo.gl/1ZR6Ka

Unique study participants	2,910
Participants that completed multiple surveys	1,691
Screenshot samples	59
Total evaluations	5,900
Accepted evaluations after filtering	5,376
Average number of accepted evaluations per sample	91

TABLE I: Statistics of the Facebook user study.

Age	
18-29	55.12%
30-39	29%
40-49	11.82%
50-59	3.33%
60 or above	0.72%

Gender	
Male	72.54%
Female	27.45%
Education	
Primary school	2.06%
High school	34.57%
Bachelor	63.36%

TABLE II: Demographics of the Facebook user study.

D. Results

We discarded survey responses where the study participants did not indicate active usage of the Facebook app or gave an incorrect reply to the control question. After filtering, we had 5,376 completed surveys and, on the average, 91 user evaluations per screenshot sample (see Table I). Table II shows the demographics of our study participants.

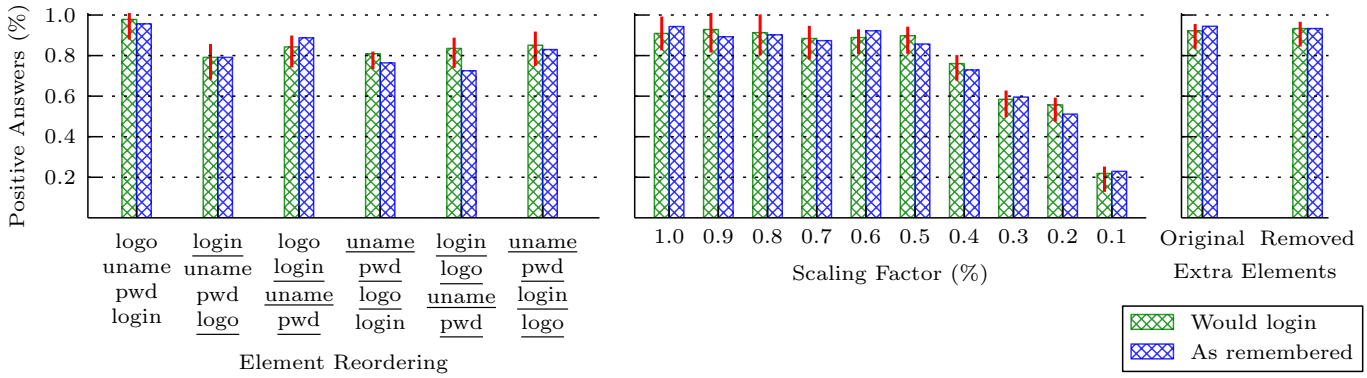

Fig. 6: General modifications results. Percentages of users that perceived a Facebook login screen sample with general modifications as genuine and would login. Element reordering modification had a small but statistically significant effect, scaling caused a significant effect, and extra element removal showed no effect.

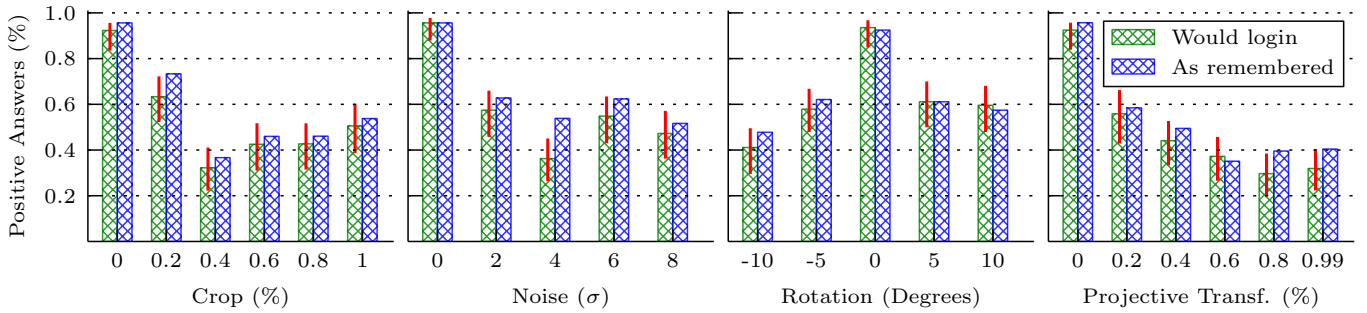

Fig. 7: Logo modifications results. Percentages of users that perceived a Facebook login screen sample with logo modifications as genuine and would login to the application. All logo modifications caused a significant effect.

Color modification. The color modification results are illustrated in Figure 5. We plot the observed login rate in green and the as-remembered rate in blue for each evaluated sample. The red bars indicate bootstrapped 95% confidence intervals. We performed a chi-square test of independence with a significance level of $p = 0.05$ to examine the relation between the login responses and the sample color. The relation between these variables was significant ($\chi^2(16, N = 1551) = 194.44), p < 0.001$) and the study participants were less likely to log in to screens with high hue change. When the hue change is maximal, approximately 40% of the participants indicated that they would still log in. For several samples we noticed slightly higher login rate compared to as-remembered rate. This may imply that some users were willing to log in to an application, although it looked different from their recollection. We investigated reasons for this behavior from the survey questions and several participants replied that they noticed the color change, but considered the application genuine nonetheless. One participant commented: *"Probably Facebook decided to change their color."* However, our study was not designed to prove or reject such hypothesis.

General modifications. The general element modification results are shown in Figure 6. Both element reordering ($\chi^2(5, N = 546) = 15.84, p = 0.007$) and scaling ($\chi^2(9, N = 916) = 245.56, p < 0.001$) had an effect on the observed login rates. Samples with scaling 50% or less showed login rates close to the original, but participants were less likely to login to screens with high scaling. This could be due to users' habituation of seeing scaled user interfaces across different mobile device form factors (e.g., smartphone user interfaces scaled for tablets). One participant commented his reason to login: *"looks the same, just a little small."* When the elements were scaled more than 50%, the login rates decreased fast. At this point the elements became unreadably small. Removal of extra elements (forgotten password or new account link) had no effect on the login rate ($\chi^2(1, N = 180) = 0.0, p = 1.0$).

Logo modifications. The logo modification results are shown in Figure 7. The relation between the login rate and the amount of crop was significant ($\chi^2(5, N = 540) = 83.75, p < 0.001$). Interestingly, we noticed that the lowest login rate was observed for the 40% crop sample. This implies that the users may find the login screen more trustworthy when the logo is fully missing compared to seeing a partial logo, but our study was not designed to prove such hypothesis.

The amount of noise in the logo had an effect on login rates ($\chi^2(4, N = 460) = 75.30, p < 0.001$), as users were less likely to log in to screens with noise. Approximately half of the study participants answered that they would login even if the logo was unreadable due to noise. This result may imply habituation to software errors and one of the

participants commented the noisy logo: *"I would think it is a problem from my phone resolution, not Facebook."* Participants were less likely to log in to screens with a rotated logo ($\chi^2(4, N = 462) = 57.25, p < 0.001$) or a projected logo ($\chi^2(5, N = 542) = 102.45, p < 0.001$).

Conclusions. The experimented eight visual modifications were perceived differently. While some modifications caused a predominantly systematic pattern (e.g., color), in others we did not notice a clear relation between the amount of the modification and the observed login rate (e.g., crop). One modification (extra element removal) caused no effect. We conclude that the spoofing detection system should be trained with samples that capture various types of visual modifications. Approaches where all types of visual changes are treated the same are unlikely to be effective.

E. Follow-up Studies and Study Method

We performed similar studies for Skype and Twitter apps, but due to space limitations we do not report the details. Skype has a standalone login screen and, as a general observation, we note that Skype results were comparable to those of Facebook. Twitter app has a distributed login screen and we noticed different patterns than in the previous two studies. Additionally, we evaluated combinations of two and three visual modifications on these apps. In total we collected 34,240 user evaluations from 5,438 unique study participants, and we used the collected data to train our detection system.

In our study, we measured login rates by asking study participants questions. We chose this approach to allow large-scale data collection for thousands of login screen evaluations, from globally-distributed participants. A common approach in spoofing (e.g., phishing) studies is to observe participants during a login operation. Scaling this method for such a large number of evaluations is challenging, as it requires either installation of malware-like apps on a large number of phones (ethical considerations) or a large app provider changing the user interface of their application for the study (possible negative user experience). Participants in our study were allowed to evaluate multiple samples from different datasets which may have influenced the results of our study.

V. Spoofing Detection System

Through our user study we gained insight into what kind of visual modifications users notice, and more importantly, fail to notice. In this section we design a spoofing detection system that leverages this knowledge. We instantiate the system for Android devices, while many parts of the system are applicable to other mobile platforms as well.

A. System Overview

Our system is designed to protect *reference applications*, i.e., legitimate apps with login functionality. The goal of our system is to, given a screenshot, estimate how many users would mistake it for one of the known reference applications. The detection system (Figure 8) consists of two parts: a training and pre-processing component that runs on the marketplace and a runtime detection system on users' phones. On the marketplace, each reference app's login screen is detected, pre-processed, and a deception rate estimator is trained using the user perception data from our user study. The analyzed login screens serve as the *reference values* for the on-device detection.

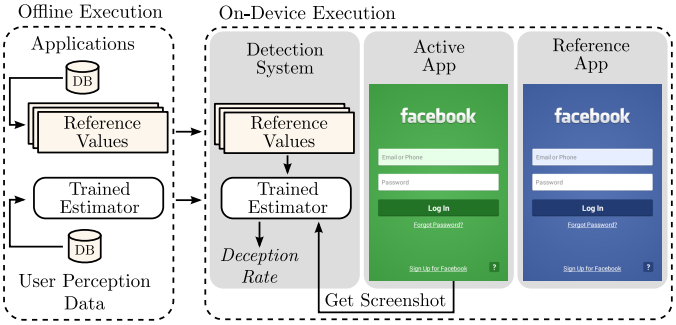

Fig. 8: Detection system overview. The system pre-processes legitimate apps offline (e.g., at the marketplace) to obtain reference values, and trains an estimator. On the user's device, the system periodically extracts screenshots and estimates their deception rate.

On the device, the system periodically extracts a screenshot of the currently active application. We analyze screenshot extraction rates needed for efficient, as well as effective detection in Section VII. Each extracted screenshot is analyzed using the estimator with respect to the reference values of the protected apps. Both the trained estimator and the reference values are downloaded from the marketplace (e.g., upon installing an app). The system outputs a deception rate for each analyzed screenshot, with respect to each protected app. The deception rates can be used to inform the marketplace or warn the user.

Which apps should be protected (i.e., labeled as reference apps), can be determined in multiple ways: the user can choose the apps that require protection, the system can automatically select the most common spoofing targets (e.g., Facebook, Skype, Twitter), or all installed apps with login functionality can be protected. In this paper, we focus on the approach where protected apps are chosen by the user. A complete view of the system is illustrated in Figure 9, and we proceed by describing each part of the system in detail.

B. Reference Application Analysis

The system protects reference apps from spoofing. To analyze an extracted screenshot with respect to a reference value, we first obtain the reference application login screen and identify its main elements (reference elements) according to our login screen model (Figure 3). We assume reference application developers that have no incentive to obfuscate their login screen implementations. On the contrary, developers can be encouraged to mark the part of the user interface (activity) that contains the login screen that should be protected. This analysis is a one-time operation performed, e.g., at the mar-

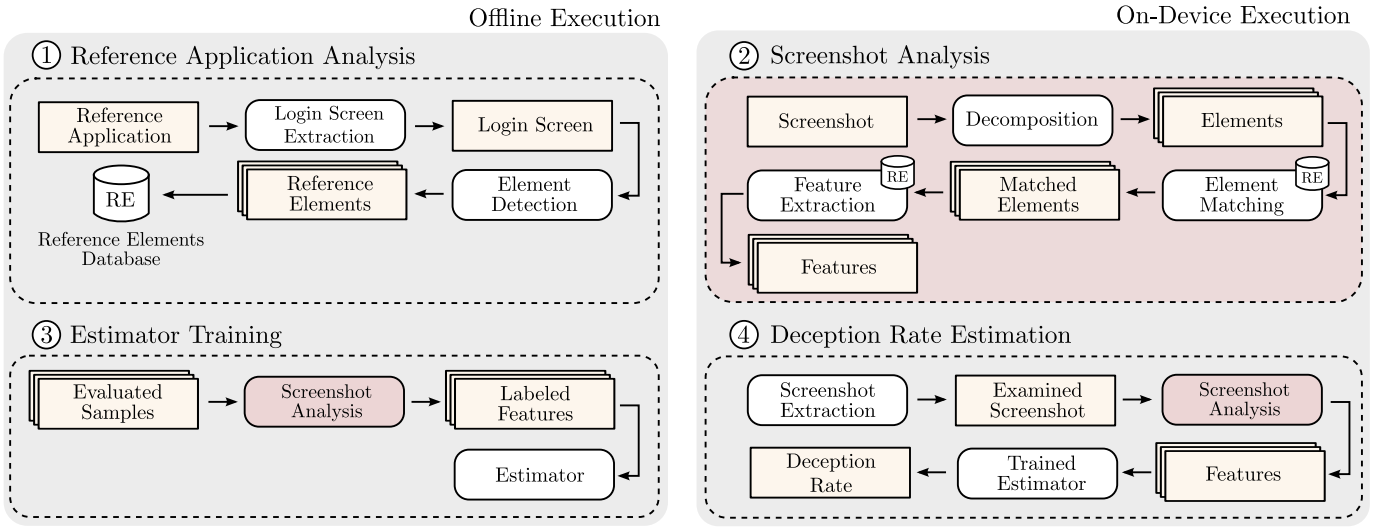

Fig. 9: Detection system details. The system consist of four main components: reference app analysis, screenshot analysis, estimator training and deception rate estimation.

ketplace on every app update, and its results distributed to the mobile devices.

On Android applications, windows and their contained elements are represented as *activities*. To find the activity that represents the login screen, we developed a tool that hooks activity and user interface element creation events. When the reference application is started, the tool hooks the creation of the first activity and searches the element tree of the activity for a password input field. We identify the first encountered text box with the `TYPE_TEXT_VARIATION_PASSWORD` flag set as the password field. If the tool finds a password input field on the first screen, it considers it a standalone login screen. The tool extracts the rest of the login screen elements by further examining the element tree. If the name of the element object class contains the word *"button"*, or if the element inherits from the `Button` Android class, the tool considers it as the login button. We identify the username field as the element with `TextView` type and consider the largest image (`ImageView`, `ImageButton`) as the logo.

If the first screen does not contain a password field, the tool considers it the initial screen in our model, extracts the logo as above, and examines all activities that can be created from the first screen. The tool identifies buttons, triggers each of them, and hooks any new created activity. For each new activity, the tool searches for a password field, and if not found, moves on to the next activity. Once a password field is found, the tool considers the examined activity as the login screen and identifies the username and login button elements as above. The tool gathers the identified elements into a tree structure and, for each element, stores its type, location, size, and content (screen capture over the element area).

C. Screenshot Analysis

The goal of the screenshot analysis is to, given the screenshot of the examined application as well as the reference elements, produce suitable features for deception rate estimation and estimator training. The screenshot analysis includes three operations: decomposition, element matching, and feature extraction, as shown in Figure 9.

Decomposition. Screenshot decomposition is illustrated in Figure 10. First, we perform common edge detection and then dilate all detected edges to fill small areas such as text. We perform a closure operation on the dilated elements to merge closely situated elements, such as individual letters in a block of text, and use a morphological gradient to determine the borders of salient elements as well as to ensure that elements that share a border get detected as two separate elements. We run a connected components algorithm to identify the regions. We filter regions smaller than a threshold and we place a bounding box around each detected area and we convert the elements into an element hierarchy tree. An element is considered a child of a parent when its bounding box is fully contained within the one of the parent. For each element, we store its location, size, and a screenshot of its area.

Element matching. The next step is to match the detected elements to the reference elements, as illustrated in Figure 11. To identify which element is the closest match to the reference logo, we use a known image feature extractor. While SIFT extractors [19] have been successfully applied for detection of logos in natural images [28], we found SIFT to be ill-suited for mobile application logos, especially in cases where only partial (cropped) logos were present. The shapes of mobile app logos are typically smooth, compared to the ones seen in natural images, and have small dimensions. Consequently, SIFT was unable to identify enough keypoints for accurate detection. We found that the ORB feature extractor [27] performed better in our context.

Matching the reference application logo to an element in the examined screenshot works as follows. We compute ORB keypoints over the reference logo as well as the whole

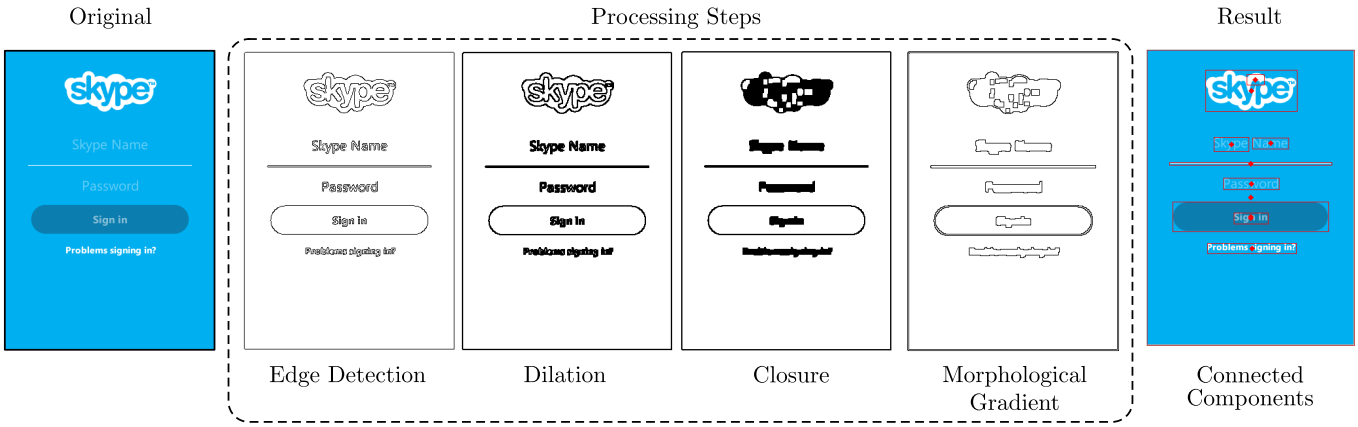

Fig. 10: Decomposition process. The processing steps in the middle includes common image analysis techniques. The final step is a connected components algorithm and filtering of smaller regions. For visual clarity, we inverted the colors in the processing steps.

examined screenshot and we match the two sets. The element that matches with the most keypoints, and exceeds a minimum point density threshold, is declared as the logo. We observe that ORB matching gives good results on all of our spoofing samples, except the ones with significant noise. Finding an image feature extractor that is resistant to noise in this setting is part of our future work.

For the remaining elements, keypoint extraction is generally not effective, as the login screen elements typically have few keypoints due to their simplicity. For every element of the examined screenshot, we perform template matching to every reference element (username field, password field, login button), on different scaling levels. The closest match determines the type of the element. After these steps, we have a mapping between the examined application elements and the reference elements (Figure 11).

Feature extraction. Once the elements are matched, we extract visual features from them. We extract two common features (color and element scaling) and more detailed logo features, as users showed sensitivity to logo changes. The extracted features are relative, rather than absolute, as their values are computed with respect to the reference elements or entire reference screen. We explain our features below:

1) *Hue.* The difference between the average hue value of the examined screenshot and the reference screen.
2) *Element Scaling.* The ratio of minimum-area bounding boxes between all reference and examined elements, except the logo.
3) *Logo Rotation.* The difference between the angles of minimum-area bounding boxes of the examined and reference logo.
4) *Logo Scaling.* We perform template matching between the examined and reference logos at different scales and express the feature as the scale that produces the best match. We undo possible logo rotation before template matching.
5) *Logo Crop.* We calculate the amount of logo crop as the ratio of logo bounding box areas. We compensate for the possible area reduction of scaling by reversing the resize operation.
6) *Logo Degradation.* As precise extraction of logo noise and projection is difficult, we approximate similar visual changes with a more generic feature that we call logo degradation. Template matching algorithms return the position and the minimum value of the employed similarity metric and we use the minimum value as the logo degradation feature. We undo possible scaling and rotation before template matching.

In cases where no logo was identified in the matching phase, all logo features are set to null (except logo crop which is set to 100%). Our analysis is designed to extract features from screenshots that follow our login screen model, and many of our features (color change, scaling) can be also observed in known spoofing apps (Android.Fakeneflic).

D. Estimator Training and Deception Rate Estimation

The detection system is trained using the available user perception data and, for our implementation, we train the system using the results of the user study. We extract features from every sample of the study and augment the resulting feature vectors with the observed login rate. In feature extraction, as the reference value we use the unmodified login screen of the app that the sample represents. As deception rate (i.e., the percentage of users that would confuse the examined screenshot with the reference application) is a continuous variable, we estimate it using a regression model. Training can be performed offline for each reference application separately.

Deception rate estimation is performed on the device at detection system runtime. As illustrated in Figure 9, the extracted screenshot is first analyzed. The decomposition phase of the analysis is performed once, and the rest of the analysis steps are repeated for each reference app. The extracted features are used to run the trained estimator. The result of the estimation

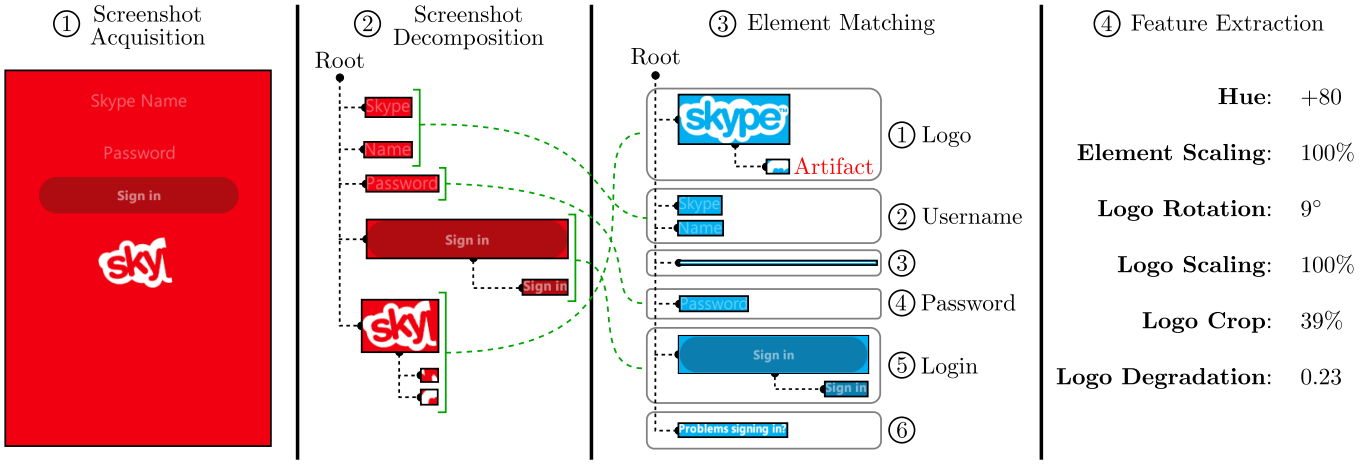

Fig. 11: Summary of the screenshot analysis. (1) The starting point is a mobile application login screenshot. (2) We decompose the screenshot to a tree hierarchy. (3) We match the detected elements to reference elements. (4) We extract features from the detected elements with respect to the reference elements.

operation is a set of deception rates, one for each protected app. If any of the deception rates exceeds a threshold value, one or more possible spoofing apps have been found and the identities of them can be communicated to the application marketplace or the user can be warned.

E. Implementation

We implemented the reference application analysis tool as a modified Android emulator environment. Similar analysis can be implemented by instrumenting the reference application, but we modified the runtime environment to make the analysis more robust, and to support analysis of native applications. We implemented the remaining offline tools as various Python scripts using the OpenCV [5] library for image processing and scikit-learn for estimator training.

The on-device detection system can be implemented in multiple ways, including a modification to the Android runtime or as a standalone application. For ease of deployment, we implemented the on-device components as a regular Android (Java) application using the OpenCV library.

VI. Evaluation

In this section, we evaluate the accuracy, robustness, and performance of our system.

A. Reference Application Analysis Accuracy

We evaluated the accuracy of our reference app analysis tool (Section V-B) on 1,270 apps, downloaded randomly from Google Play and other marketplaces. The tool reported 572 potential login screens. We manually verified all of them and found 230 login, 153 user registration, and 77 password change screens. The remaining 120 screens contained no login related functionality, and those we classify as false positives.

We manually verified 50 random apps from the set of 698 apps our tool reported as not having a potential login screen. We found 3 false negatives due to an implementation bug that was since fixed. We conclude that the tool can effectively find all login screens that require protection. The tool provides an over approximation, but a small number or false positives does not hamper security, as they only introduce additional reference values for similarity comparison. Moreover, developers have an incentive to help the reference login screen detection and they can explicitly mark which activity constitutes the login screen for even more accurate detection.

B. Decomposition Accuracy

To evaluate the accuracy of our screenshot decomposition algorithm, we decomposed the screenshots of the 230 login screens. We manually verified the results and found that we detected all login screen elements correctly on 175 screens. We found 29 screens that correctly decomposed all but one element, and 9 screens with correct decompositions for all but two elements. Our algorithm failed to decompose 18 screens.

Certain types of login screens are challenging for our approach. For example, the login screen of the Tumblr application contained a blurred natural image in the background, and our algorithm detected many erroneous elements (see Figure 12). Our current implementation is optimized for clean login screens, as those are the pre-dominant login screen types. The majority (92%) of analyzed screenshots were visually simple and decomposed. We discuss noisy spoofing screens as a possible detection avoidance technique in Section VII-C.

C. Deception Rate Estimation Accuracy

To evaluate the deception rate estimation accuracy, we trained our detection system using the results of our user study. Our total training data consists of 316 user-evaluated samples of visual modifications and each sample was evaluated either by 100 (single modification) or 50 (two and three modifications) unique study participants. From the training data, we omitted samples that express visual modifications that our current implementation is unable to extract (e.g., noise).

We experimented with several regression models of different complexities and trained two linear models (Lasso and linear

Fig. 12: Decomposition examples. The login screen decomposition algorithm works well in practice. We outline in red the borders of detected elements, while the red diamond represent element centroids. Some login screens (tumblr, last screenshot) are visually complex and are inherently hard for our approach to analyze.

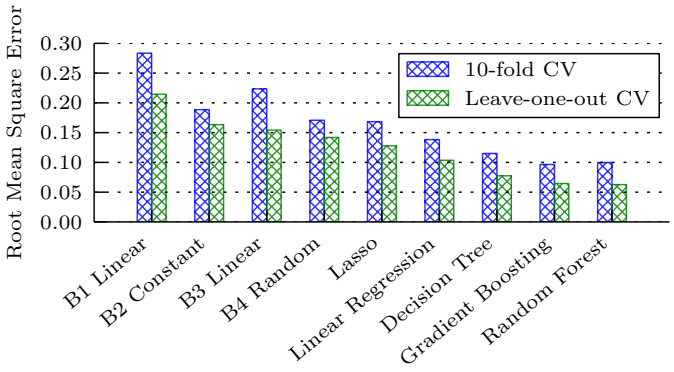

Fig. 13: Deception rate accuracy. Evaluation of five regression and four baseline models (B1–B4) trained on the combined datasets of Facebook and Skype. The random forest regressor performs the best.

regression), a decision tree, as well as two ensemble learning methods (gradient boosting and random forests). To compare our detection accuracy to straightforward approaches, we define four baseline models out of which the latter two utilize prior knowledge obtained from our user study:

- *B1 Linear.* The deception rate drops linearly with the amount of visual modification from 1 to 0.
- *B2 Constant.* The deception rate is always 0.75.
- *B3 Linear.* The deception rate drops linearly with the amount of visual modification from 1 to 0.2. Login rates stayed predominantly above 20% in our study.
- *B4 Random.* The deception rate is a random number in the range 0.3–0.5; the most observed range in our study.

To estimate the deception rate, we extract features from the analyzed screenshot with respect to a reference app and we feed the feature vector to the trained regressor. The estimator outputs a deception rate that can be straightforwardly converted into a spoofing detection decision. We performed two types of model validation: leave-one-out and 10-fold cross-validation. We report the results in Figure 13 and we observe that the more complex models perform significantly better than our baseline models. The best model was random forest, with a root mean square (RMS) error of 6% and 9% for the leave-one-out and 10-fold cross validations respectively (95% of the estimated deception rates are expected to be within two RMS errors from their true values). The low RMS values show that a system trained on user perception data can accurately estimate deception rates for mobile application spoofing attacks.

The detection system should estimate deception rate accurately even for apps it did not encounter before. To evaluate the estimation accuracy of attacks that target apps that were not present in the training data, we trained a random forest regressor using Facebook samples, and evaluated it on Skype samples, and vice-versa. We observed an RMS error of 13% in both cases. When samples from the spoofing target app are not part of the training dataset, the estimation accuracy decreases slightly. We conclude that our system is able to accurately estimate deception rate in the tested scenarios, even if the target application is not part of the training data. Our training set has limited size and with more extensive training data we expect even better accuracy.

D. Performance Evaluation

We evaluated the performance of the on-device screenshot analysis and deception rate estimation operations. For the offline (marketplace) components we only evaluated accuracy, as those are fast and not time-critical operations. We measured the performance of our implementation on three devices: an older smartphone model (Samsung Galaxy S2) and two more recent devices (Nexus 5 and Nexus 6). Table III shows execution times averaged over 100 runs. Analysis with respect to a single reference application takes from 183 ms (Nexus 5) to 407 ms (Galaxy S2). The process scales linearly with the number of protected apps: the decomposition of the extracted screenshots is performed once, and the remaining analysis steps are repeated for each reference value. Assuming five protected applications, the complete analysis takes 667 ms on Nexus 5.

We use application whitelisting to eliminate unnecessary processing. The detection system extracts and analyzes screenshots only when an untrusted (i.e., not whitelisted) app is active. For example, the platform provider can whitelist popular apps from trusted developers (Facebook, Twitter, Whatsapp). Below we discuss further performance improvement

	Galaxy S2	Nexus 5	Nexus 6
Screenshot extraction	10 ± 3 ms	21 ± 13 ms	19 ± 7 ms
Decomposition	99 ± 19 ms	41 ± 10 ms	42 ± 8 ms
Element matching	147 ± 35 ms	54 ± 16 ms	106 ± 16 ms
Feature extraction	150 ± 34 ms	67 ± 12 ms	94 ± 13 ms
Estimator	0.5 ± 0.9 ms	0.1 ± 0.3 ms	0.4 ± 0.5 ms
Total	407 ± 69 ms	183 ± 28 ms	261 ± 26 ms

TABLE III: Performance evaluation of our implementation.

techniques that can significantly reduce the required computation time for each screenshot. In Section VII we describe a collaborative deployment model that allows even further computation reduction by less frequent screenshot extraction.

E. Performance Improvements

The detection system can perform a less expensive *pre-filtering* operation to determine if the examined screenshot vaguely resembles a login screen, and if so, proceed with the full analysis. Screenshot decomposition might function as such pre-filtering operation, where the detection system could continue with the full screenshot analysis only if the decomposition phase provides a number of elements, or similar heuristic, that is close to the login screen model. Efficient pre-filtering would avoid the expensive analysis for the majority of the extracted screenshots, as most of them are benign and do not resemble a login screen. The adversary should not be able to create spoofing screens that are pre-filtered but still deceive many users, and we leave thorough evaluation of secure pre-filtering schemes as future work.

The on-device performance primarily depends on the size of the analyzed screenshot. Modern smartphones have high screen resolutions (e.g., 1080×1920) and analyzing such large images is expensive and does not increase system accuracy. It is important to note that screenshot extraction time depends only on the output screenshot resolution and not on the physical screen resolution itself. For all our measurements we extracted screenshots of size 320×455 pixels as the resolution provides a good ratio of element detection accuracy and runtime performance. Our initial experiments show that the image resolution (and with it, execution time) can be decreased even further, and determining the optimal resolution we leave as future work.

VII. Analysis

In the previous section we evaluated the computational cost of deception rate estimation for a single screenshot. In this section we explain how often screenshots can be extracted on the device, given a pre-defined amount of allocated system resources. If a spoofing attack takes place, we analyze the probability that at least one screenshot of the spoofing application is captured. We also present an efficient collaborative detection model that enables significantly fewer screenshot analysis operations per device.

A. Detection Probability on a Single Device

In Figure 14, we illustrate the intuition of our analysis. The system has two controllable parameters: the share of system resources s that are allocated for spoofing detection and the number of reference apps n_a the system protects. Together with device performance and the observed user habits (the share of time spent on unknown apps u), these two parameters define the screenshot rate r which in turn determines the detection probability for a single spoofing attempt p_s, as well as the number of devices n needed for efficient collaborative detection. In what follows, we introduce the rest of the terms gradually and, for ease of reference, summarize our terminology in Table IV.

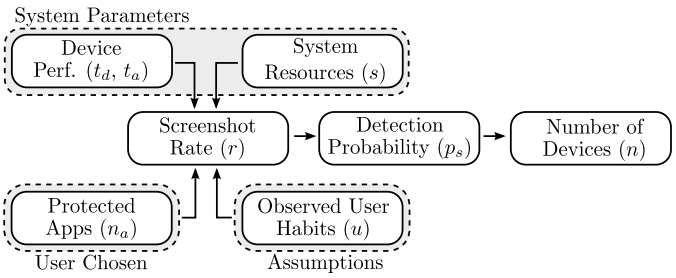

Fig. 14: Analysis intuition. System parameters, user behavior assumptions, and a user-chosen number of protected apps define the screenshot rate, the detection probability for a single spoofing attack, and the number of devices required for effective collaborative detection.

System Presets	s	Share of allocated system resources
	t_d	Decomposition time (device perf.)
	t_a	Analysis time (device perf.)
Observed User Habits	t_l	Time spent on login screen
	u	Share of time spent on unknown apps
User Chosen	n_a	Number of protected applications
Detection Properties	r	Screenshot rate
	p_s	Detection probability, single spoofing
	p	Detection probability, collaborative system
	n	Number of devices with spoofing app

TABLE IV: Summary of analysis terminology.

In a typical deployment, the share of system resources allocated for the detection system would be chosen by the platform provider. For our analysis, we use $s = 1\%$, as we assume that one percent overhead does not hinder user experience nor overly drain the battery. The number of protected applications is chosen by the user. We assume that in most cases the user would choose to protect a small number of important services (e.g., banking, e-mail, Facebook, Skype, Twitter) and use the value $n_a = 5$ for our analysis.

For analysis simplicity, we assume that the user spends a constant time t_l on the spoofed login screen. In a recent study [21], users spent 4–28 seconds on the login screen, so $t_l = 5$ seconds is a safe assumption. We also assume that the user spends a constant share u of her time on unknown (non-whitelisted) apps. According to [9], smartphone users spend 88% of their time on five of their favorite apps, so setting $u = 0.25$ is a safe assumption. The detection system can monitor the runtime usage of unknown apps and adjust a user-specific u accordingly.

For device performance, we use the values from our implementation evaluation on Nexus 5: the screenshot extraction and decomposition time t_d is approximately 60 ms, while the remaining screenshot analysis time t_a that needs to be repeated for each reference app is approximately 120 ms. Using such device performance, system parameters and analysis assumptions, we can compute the screenshot rate r as follows:

$$r = \frac{u}{s}(t_d + n_a t_a) \approx 16.5 \text{ s}$$

That is, given 1% of allocated system resources, a screenshot can be analyzed on the average once per 16.5 seconds when an unknown app is active.

The detection probability for a single spoofed login screen p_s is the probability that, when a spoofed login screen is shown to the user for $t_l = 5$ seconds, the detection system captures, and analyzes, at least one screenshot during that time. To avoid simple detection evasion where the malware never shows spoofed screens at pre-determined screenshot extraction times, we assume that screenshots are taken at random points in time, according to the chosen screenshot rate. Given the randomized screenshot extraction model, we model p_s as a random number from the Poisson distribution $P(x; \mu)$, where x is the number of successes in a given time period (zero successes means that no screenshots are taken in the time period) an μ is the mean of successes in the same time period. The number of screenshots taken on the average can be calculated as t_l/r (e.g., $5/16.5$ in our example scenario). The detection probability p_s becomes:

$$p_s = 1 - P(0, \frac{t_l}{r}) \approx 0.26$$

We observe that the probability of detecting a single spoofed login operation is low. Moreover, the adversary does not have an incentive to repeat a successful attack on the same device. Once the users login credentials have been stolen, the malicious app can, e.g., remove itself. For these reasons we focus on a more effective collaborative deployment model that leverages the *many eyes principle*.

B. Collaborative Detection

An instance of the detection system can be running on a large number of devices (e.g., all devices from the same platform provider), where each device takes screenshots at random points in time, according to the chosen screenshot rate. When one of the devices finds a potential spoofing login screen, the identity of the application is reported to the platform provider (or the app marketplace) which can examine the application and remove it from all of the devices, if confirmed malicious. For analysis simplicity, we assume that all participating devices have similar performance and use the same, previously chosen system parameters, but deployments where devices are configured differently are, of course, possible. The detection probability p of the collaborative system, i.e., the probability that at least one device will detect the spoofing attack, is defined as:

$$p = 1 - (1 - p_s)^n$$

System Resources (s)	**Number of Devices** (n)		
	$n_a = 1$	$n_a = 5$	$n_a = 10$
0.5%	9	31	59
1.0%	5	16	30
2.0%	3	8	15

TABLE V: The number of devices n needed in collaborative detection depends on the allocated system resources s and the number of protected apps n_a. We illustrate the number of devices required for detection probability $p = 0.99$ in example scenarios.

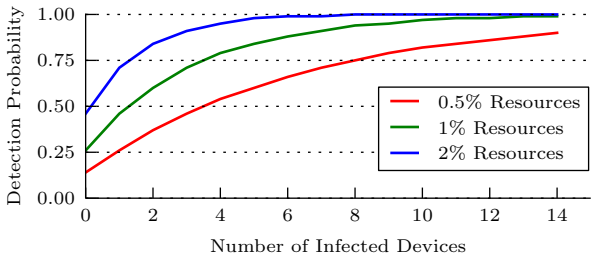

Fig. 15: The detection probability p as a function of infected devices n. We consider allocated system resources $s = \{0.5, 1, 2\}\%$ and assume $n_a = 5$. Detection is practical even with very low number of infected devices.

where n is the number of devices infected with the spoofing app. Assuming our example parameters, to reach detection probability $p = 0.99$, we need the malicious application to be installed and active on only 16 devices:

$$n = \lceil log_{1-p_s}(1 - p) \rceil = 16$$

Spoofing apps that infected thousands of devices have been reported [14], so we consider this a very low number for common wide-spread attacks that target globally used apps, such as Facebook, Skype or Google. In Table V, we provide example values for the number of infected devices required to reach detection probability $p = 0.99$ given different values of our system parameters s and n_a. Detection on a single spoofing attempt is hard to guarantee, but if the spoofing app is active even in a small number of devices (e.g., 10) it can be detected with high probability using only a small share of system resources in participating devices. Figure 15 illustrates the detection probability p as a function of infected devices n, and we observe that detection is practical even with very few infected devices.

The goal of the collaborative detection system is to keep a constant, high detection probability at all times. This can be achieved with fewer devices sampling more often or more devices sampling less often. For example, the screenshot rate can be controlled based on the popularity (global install count) of the currently running, unknown app. The marketplace can send periodic updates on the popularity of each application installed on the device. If an app is present on many devices (e.g., 50 or more), the detection system can safely reduce the screenshot rate to save system resources without sacrificing detection probability. If an application is installed in only a

small number of devices (e.g., less than 10), the system can increase the screenshot rate for better detection probability. Such adjustments can be done so that, in total, no more than the pre-allocated amount of system resources are spent for spoofing detection. Our analysis has shown that collaborative detection provides an efficient way to detect spoofing attacks in the majority of practical spoofing scenarios.

C. Detection Avoidance

The adversary can try to avoid runtime detection by leveraging the human perception property of averaging images that change frequently. For example, the adversary could quickly and repeatedly alternate between showing the first and second halves of the spoofing screen. The user would perceive the complete login screen, but any acquired screenshot would cover only half of the spoofing screen. Such attacks can be addressed by extracting screenshots frequently (screenshot extraction is fast) and averaging them out, prior to analysis. Such an acquisition method would mimic the human perception.

While the adversary has an incentive to create spoofing screens that resemble the original login screen, the adversary is not limited to these modifications. To test how well our system is able to estimate deception rate for previously unseen visual modifications and spoofing samples that differ from the login screen model, further tests are needed. This limitation is analogous to the previously proposed similarity detection schemes that compare website similarity to known phishing samples — the training data cannot cover all phishing sites.

Our current implementation has difficulties in decomposing screenshots with background noise, and consequently the adversary could try to avoid detection by constructing noisy spoofing screens. Developers could be encouraged to create clean login screen layouts for improved spoofing protection. While we did not experiment with noisy backgrounds, our study shows that typically the more the adversary deviates from the legitimate screen, the less likely the attack is to succeeded.

VIII. Related Work

Spoofing detection systems. Bianchi et al. [4] propose a static analysis tool for mobile app spoofing detection that identifies API calls that enable spoofing attacks. The tool detects apps that query device state (e.g., running tasks) and after that perform UI related operations (e.g., create a new activity). Code analysis can be very effective in detecting attacks that leverage known attack vectors. Our approach is more agnostic to the attack implementation technique, but has a narrower focus: protection of login screens. We consider our work complementary to mobile app spoofing detection using code analysis.

Many web phishing detection systems analyze a website DOM tree and compare its elements and structure to the reference site [3, 18, 26, 38, 39]. While similar code analysis is possible for mobile applications, we assume an adversary that constructs spoofing applications in arbitrary ways (e.g., per pixel), and thus complicates structural code analysis. Our screenshot analysis techniques can help such approaches to infer user interface structure under strong adversarial models.

Another proposed approach is to consider the visual presentation of a spoofing application (or a website), and compare its similarity to a reference value [8, 15, 22]. The main difference between these schemes and our work is that they derive a similarity score for a website and compare it to the ones of known malicious sites. Our similarity metric determines how many users would confuse the application for another one. Unlike these previous works, we also extract visual features for the similarity analysis by decomposing the user interface from its visual presentation. The results of our user study could be used to determine appropriate visual similarity metrics in the above discussed approaches.

Spoofing detection by users. Two types of techniques have been proposed to help the user to detect spoofing attacks. First, similar to web browsers, the mobile OS can be enhanced with security indicators. For example, the OS can show the name, or comparable identifier, of the running application in a dedicated part of the screen, such as on the status bar [4, 13, 30]. Such schemes require that parts of the mobile device screen are made unavailable to applications or need hardware changes to the mobile device. Second, a mobile application can allow the user to configure a personalized security indicator (e.g., a personal image) that is shown by the application during each login [21]. Such application-specific security indicators require no platform changes, but increase application setup user effort.

User perception of spoofing attacks has been studied extensively in the context of web phishing. Several studies show that many users ignore the absence of security indicators, such as SSL locks or personalized images [11, 29, 34]. A recent study shows that personalized security indicators can be more effective on mobile apps [21]. We are the first to study how likely the users are to notice spoofing attacks, where the malicious application resembles, but is not a perfect copy of, the legitimate application.

IX. Conclusion

We have proposed a novel mobile app spoofing detection system that in collaborative fashion extracts screenshots periodically and analyzes their visual similarity with respect to protected login screens. We express the similarity in terms of a new metric called deception rate that represents the fraction of users that would confuse the examined screen for one of the protected login screens. We conducted an extensive online user study and trained our detection system using its results. Our system estimates deception rate with good accuracy (6-13% error margins) and very low overhead (only 1%). Essentially, our system tells how likely the user is to fall for a potential attack. We consider this a powerful and interesting security property that no previous similarity comparison scheme provides. In addition to supporting a spoofing detection system, the results of our user study, on their own, provide an insight into the perception and attentiveness of users during the login process.

REFERENCES

[1] Google safe browsing. http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html.

[2] Spoofguard. http://crypto.stanford.edu/SpoofGuard/.

[3] S. Afroz and R. Greenstadt. Phishzoo: Detecting phishing websites by looking at them. In *Fifth IEEE International Conference on Semantic Computing (ICSC)*, 2011.

[4] 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 *Symposium on Security and Privacy (SP)*, 2015.

[5] G. Bradski. *Dr. Dobb's Journal of Software Tools*.

[6] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, volume 15, 2015.

[7] 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 *USENIX Security Symposium*, 2014.

[8] T.-C. Chen, S. Dick, and J. Miller. Detecting visually similar web pages: Application to phishing detection. *ACM Trans. Internet Technol.*, 10(2):1–38, 2010.

[9] comScore. The 2015 u.s. mobile app report, 2015.

[10] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Symposium on Usable Privacy and Security (SOUPS)*, 2005.

[11] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Conference on Human Factors in Computing Systems (CHI)*, 2006.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[13] A. P. Felt and D. Wagner. Phishing on mobile devices. In *Web 2.0 Security and Privacy Workshop (W2SP)*, 2011.

[14] Forbes. Alleged 'Nazi' Android FBI Ransomware Mastermind Arrested In Russia, April 2015. http://goo.gl/c91izV.

[15] A. Fu, L. Wenyin, and X. Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (EMD). *IEEE Transactions on Dependable and Secure Computing*, 3(4):301–311, 2006.

[16] J. Hong. The state of phishing attacks. *Communications of the ACM*, 55(1), 2012.

[17] International Secure Systems Lab. Antiphish, last access 2015. http://www.iseclab.org/projects/antiphish/.

[18] W. Liu, X. Deng, G. Huang, and A. Fu. An antiphishing strategy based on visual similarity assessment. *IEEE Internet Computing*, 10(2), March 2006.

[19] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 2004.

[20] MacRumors. Masque attack vulnerability allows malicious third-party iOS apps to masquerade as legitimate apps. http://www.macrumors.com/2014/11/10/masque-attack-ios-vulnerability/.

[21] C. Marforio, R. Jayaram Masti, C. Soriente, K. Kostiainen, and S. Capkun. Personalized Security Indicators to Detect Application Phishing Attacks in Mobile Platforms. *ArXiv e-prints*, Feb. 2015.

[22] M.-E. Maurer and D. Herzner. Using visual website similarity for phishing detection and reporting. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, 2012.

[23] E. Medvet, E. Kirda, and C. Kruegel. Visual-similarity-based phishing detection. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2008.

[24] W. Metzger. *Laws of Seeing*. The MIT Press, 2009.

[25] R. A. Rensink. Change detection. *Annual review of psychology*, 53(1), 2002.

[26] A. P. Rosiello, E. Kirda, C. Kruegel, and F. Ferrandi. A layout-similarity-based approach for detecting phishing pages. In *Conference on Security and Privacy in Communications Networks (SecureComm)*, 2007.

[27] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *International Conference on Computer Vision (ICCV)*, 2011.

[28] H. Sahbi, L. Ballan, G. Serra, and A. Del Bimbo. Context-dependent logo matching and recognition. *Image Processing, IEEE Transactions on*, 22(3), March 2013.

[29] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *IEEE Symposium on Security and Privacy (SP)*, 2007.

[30] M. Selhorst, C. Stuble, F. Feldmann, and U. Gnaida. Towards a trusted mobile desktop. In *International Conference on Trust and Trustworthy Computing (TRUST)*, 2010.

[31] D. J. Simons and R. A. Rensink. Change blindness: past, present, and future. *TRENDS in Cognitive Sciences*, 9(1), 2005.

[32] Spider Labs. Focus stealing vulnerability in android. http://blog.spiderlabs.com/2011/08/twsl2011-008-focus-stealing-vulnerability-in-android.html.

[33] Symantec. Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna?, April 2015. http://goo.gl/xh58UN.

[34] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Conference on Human Factors in Computing Systems (CHI)*, 2006.

[35] G. Xiang, J. Hong, C. P. Rose, and L. Cranor. Cantina+: A feature-rich machine learning framework for detecting phishing web sites. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):21, 2011.

[36] Z. Xu and S. Zhu. Abusing notification services on smartphones for phishing and spamming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[37] J. Zhai and J. Su. The service you can't refuse: A secluded hijackrat. https://www.fireeye.com/blog/threat-research/2014/07/the-service-you-cant-refuse-a-secluded-hijackrat.html.

[38] H. Zhang, G. Liu, T. Chow, and W. Liu. Textual and visual content-based anti-phishing: A bayesian approach. *IEEE Transactions on Neural Networks*, 22(10), Oct 2011.

[39] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: A content-based approach to detecting phishing web sites. In *International Conference on World Wide Web (WWW)*, 2007.

[40] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Conference on Data and Application Security and Privacy (CODASPY)*, 2012.