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. 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. 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 work, we propose a novel approach for addressing mobile application spoofing attacks by leveraging the visual similarity of application screens. We use deception rate as a novel metric for measuring how many users would confuse a spoofing application for the genuine one. We conducted a large-scale online study where participants evaluated spoofing samples of popular mobile applications. We used the study results to design and implement a prototype spoofing detection system, tailored to the estimation of deception rate for mobile application login screens.

1. 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 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 [18, 30, 34]. For example, a recent mobile banking spoofing application infected 350,000 Android devices and caused significant financial losses [12]. More sophisticated attack vectors are described in recent research [6, 11, 33].

The problem of spoofing has been studied extensively in the context of phishing websites. However, despite numerous countermeasures proposed and deployed [1, 2, 15], web phishing remains a problem [9, 14]. Mobile application spoofing attacks are even harder to detect than web phishing attacks. Web applications run in browsers that provide visual cues, such as URL bars, SSL lock icons and security skins [8], that 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 phishing application — *the user is therefore unable to authenticate the contents of the screen.* Security indicators for smartphone platforms have been proposed [10, 28], 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 applications. Application-specific personalized indicators [33] require no platform changes, but increase the application setup effort.

An intuitive approach to address the problem of mobile application spoofing is to compare the similarity of mobile applications, for example, upon their deployment on the market or during their use. To detect web phishing attacks, various schemes and metrics have been proposed that compare website DOM trees and their elements [3, 16, 24, 35, 36] as well as the final visual appearance of websites [13, 20]. While these schemes can be effective in determining the amount of structural or visual similarity, the proposed metrics do not measure how likely the attack is to succeed.

In this paper, we propose a novel approach for addressing mobile application spoofing attacks by leveraging visual similarity, as perceived by the users. We design and build a prototype system, tailored for mobile login screen spoofing, that first trains on user perception data and then uses the obtained knowledge to measure the similarity of application visual appearances. Our system uses *deception rate*, a novel metric that measures how many users would confuse a spoofing application for a legitimate one. We consider a strong adversary that is capable of constructing phishing screens in arbitrary ways. For example, the adversary can create the spoofed user interface pixel by pixel to complicate structural similarity analysis and visual feature extraction.

Our system extracts visual features from the screenshot that is presented to the user and is therefore agnostic to the way the phishing screen is constructed. This is in contrast to approaches based on application code analysis.

Our system requires a good understanding of how users remember mobile application user interfaces and how they react to perceived changes within them. Perceived change has been studied extensively in general [22, 23, 29], 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 phishing 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 application feature. We notice that users are habituated to faulty mobile applications and that users are surprisingly tolerant to login screen distortions. For most of the visual modifications we experimented with, we noticed a systematic user behavior: the more a visual property is changed, the less likely the users are to consider the application genuine.

We used the results of our study as input into our spoofing detection system that estimates the deception rate for mobile applications. Our prototype system uses common supervised learning techniques and shows good accuracy: it is able to estimate the deception rate with 6% error margin on the applications we tested it with. The runtime detection overhead of our prototype is not excessive. These results show that detection of spoofing attacks based on user perception is a viable approach for mobile applications, where user interfaces are typically simple and their designs clean. Our results are also useful to other spoofing detection schemes, as they give insight into how users perceive visual change.

Contributions. To summarize, we make the following contributions:

- We propose a *novel approach* for the detection of mobile application spoofing attacks under a strong adversarial model by leveraging knowledge of users' similarity perception and using *deception rate* as a novel similarity metric.

- We conducted a *large-scale user study* on the perception of visual modifications in mobile application login screens.

- We designed and implemented a prototype of a runtime *spoofing detection system* for Android using common supervised learning techniques and leveraging our user study results.

- We developed novel *visual feature extraction* techniques that are agnostic to the way the spoofing screen is constructed.

The rest of this paper is organized as follows. In Section 2 we explain the problem of mobile application spoofing. Section 3 introduces our approach, login screen spoofing case study, and attacker model. In Section 4 we describe our user study and discuss its results. Section 5 explains the spoofing detection system design, implementation, evaluation, and directions for further research. Section 6 reviews related work, and we conclude in Section 7.

2. 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 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 application 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 cloned application. To the user, the application appears identical to the target application, except for subtle visual cues such as a different developer name. Such malicious applications are typically distributed via side-loading to avoid detection on the marketplace.

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, 11]. Background attacks are difficult to notice for the user. Static code analysis on the marketplace can be used to detect API calls that enable background attacks [4], but automated detection is complicated by the fact that the same APIs are frequently used by benign apps.

A malicious application can present a button to share information via another app. Instead of forwarding the user to the suggested target application, the button triggers a spoofing screen within the same, malicious application [11]. Fake forwarding requires no specific permissions or specific API calls which makes such attack vectors difficult to discover using code analysis. Further spoofing attack vectors are discussed in [4].

While spoofing attacks have not yet become as pervasive as other types of mobile malware, serious attacks have already taken place. Svpeng malware infected 350,000 Android devices and caused financial loss worth of nearly one million USD [12]. In addition to ransomware functionality, Svpeng presents a spoofed credit card entry dialog when the user starts the Google Play application. The malware also monitors startup of targeted mobile banking applications and performs background spoofing attacks for their login screens in order to steal mobile banking credentials.

Because of such serious attacks and challenges in marketplace code analysis, we believe it is useful to seek alternative ways to address the problem of mobile application spoofing.

3. OUR APPROACH

In this section we first describe the rationale behind our visual similarity approach and we 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 conclude this section by describing our attacker model.

3.1 Visual Similarity and Deception Rate

The problem of application spoofing can be approached in multiple ways. One approach is to perform static code analysis in order to detect API calls that enable spoofing attacks [4]. However, such approaches are limited to known attack vectors and do not address spoofing attacks that require no specific API calls (e.g., fake forwarding). A second approach is to analyze the application code or website DOM trees and identify applications with *structural* user interface similarity [3, 16, 24, 35, 36]. A limitation of this approach is that the adversary can complicate code analysis by, for example, constructing the user interface pixel by pixel. A third approach is to enhance the mobile platform with security indicators [10, 28]. 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 [33] 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 [13, 32, 35].

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 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 and would result in many false negatives.

The second straw-man solution is to flag all applications that have some visual similarity to the 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 and finding a similarity threshold that does not produce excessive false positives or negatives can be challenging. For accurate and efficient mobile application spoofing detection, more sophisticated techniques are needed.

In this paper, we 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 reference app, our system (Figure 1) 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 application user interfaces.

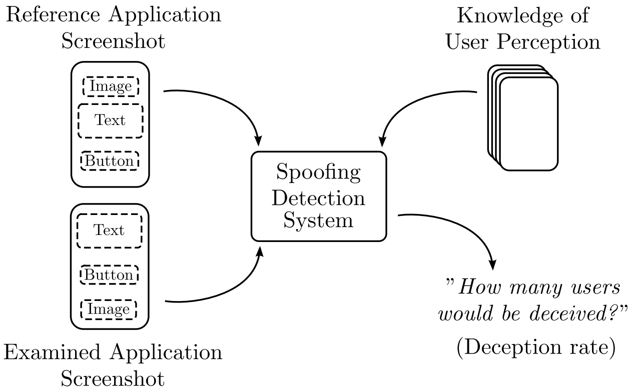
Reference Application Screenshot
Image
Text
Button
Text
Button
Image
Examined Application Screenshot
Spoofing Detection System
Knowledge of User Perception
"*How many users would be deceived?*"
(Deception rate)

Figure 1: Overview of our approach. 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.

The deception rate can therefore be seen as a risk measure that allows our system to determine if the examined application should be flagged as a potential spoofing application. An example policy would be to flag any application where the deception rate exceeds a threshold. By controlling the threshold, the system administrator can adjust the ratio between false positives and negatives.

Deception rate is a conceptually different similarity metric from the ones previously proposed for similarity analysis of phishing websites. These works extract structural [3, 16, 24, 35, 36] as well as visual [7, 13, 20] similarity features and combine them into a similarity score that alone is not expressive, but enables comparison to known attack samples [16, 21]. The extracted features can also be fed into a system that is trained using known malicious sites [13, 32, 35]. 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 one previously seen.

Deception rate has different semantics, as it captures the users' *perceived similarity* of spoofing screens. For example, a mobile application 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 known spoofing attack 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 4.

3.2 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 manually examined 20 popular Android social network apps and found that their login screens 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

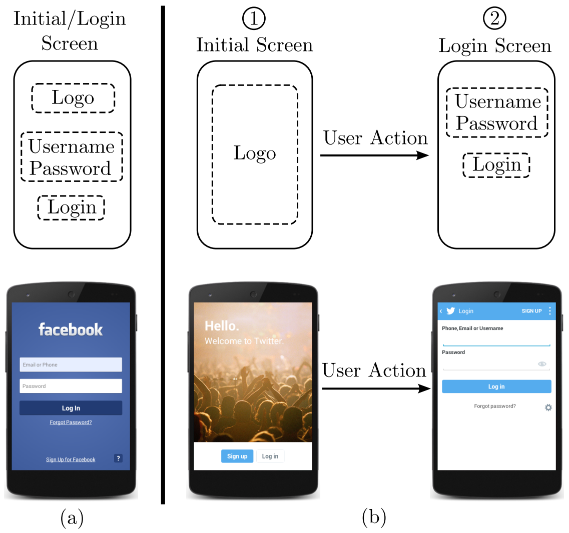

Figure 2: Model for mobile application login screens. The login screen has three main elements: logo, username and password input fields, and login button. The login functionality is either (a) standalone or (b) distributed.

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 also the first screen the user sees, i.e., the *initial screen*. Other applications distribute these elements across two screens. The initial screen contains the logo, or a similar visual identifier, as well as a button or a link that leads to the login screen, where the rest of the main elements reside.

Their common structure enables us to model mobile application login screens, and their simple and clean designs provide a good opportunity to experiment on user perception. Such simple 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 2 that captures both standalone and distributed logins screens. Five of the 20 apps we examined had a standalone login screen, while 15 had a distributed one. All apps conformed to our model.

Our focus is to 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. For example, the adversary could create a spoofed login screen where the background color or the application logo has been modified. While such a login screen would be visually different from the legitimate one, it would still conform to the model. We do not experiment on changes that fall outside the model.

3.3 Attacker Model

We assume a strong attacker capable of creating arbitrary spoofed login screens. This also includes login screens that deviate from our model which we discuss in Section 5.6.

We distinguish between two spoofing attack scenarios regarding user expectations and goals. In all the spoofing attacks listed in Section 2, 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. Alternatively, the adversary could 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.

We assume an attacker that controls a malicious spoofing application running on the user smartphone. The attacker can construct a mobile application spoofing screen statically (e.g., through use of 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 in a pixel-by-pixel manner, in which case the operating system sees only one element, a displayed picture. Furthermore, mobile applications can collude and create collaborative screens where, e.g., each malicious app creates a portion of the spoofed screen.

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 still perceive the complete spoofing screen.

4. 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) [23, 29]. 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 applied to the spoofing detection system outlined above. To the best of our knowledge, no previous studies on the 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 two-fold. First, we wanted to understand how users perceive changes in login screens. We were especially interested in any insights that could be used to guide the design of our spoofing detection system. Second, we wanted to gather training data for the spoofing detection system.

The study was performed as a set of 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.

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 recognized application. This study had no *a priori* hypothesis, i.e., the study was exploratory. After that we carried out follow-up studies where we tested similar visual modifications on Skype and Twitter apps and combinations of visual changes. Below, we describe the initial 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 our study and informed us that it does not need approval.

4.1 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 modi-

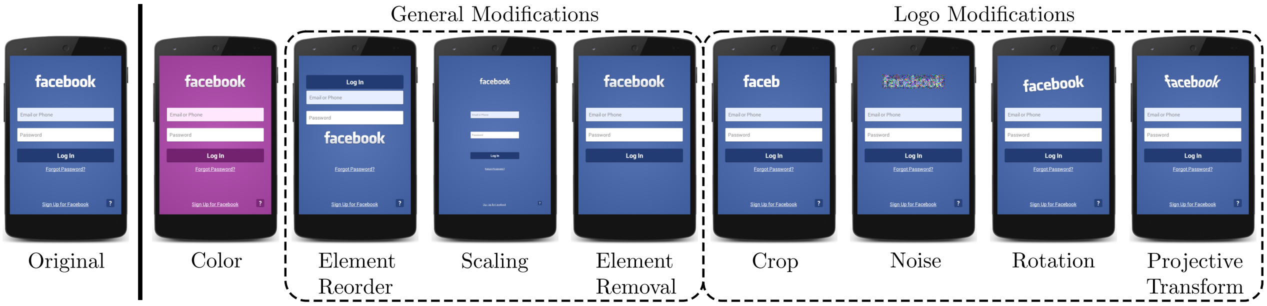

Figure 3: Examples from Facebook login screen samples. The original login screen is shown on the left. We show an example of each type of visual modification we performed: color, general modifications, and logo modifications.

fied 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 5), and in what follows, 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 3 (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 login screen. (3) We removed any extra elements from the login screen. Figure 3 (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 synthetic spoofing samples because no extensive mobile phishing application dataset is available. Moreover, the goal of our approach is not only to detect spoofing attacks similar to existing samples, but also previously unseen attacks. We therefore want to understand how users perceive as many different visual changes as possible.

4.2 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. We accepted participants globally through the crowd sourcing platform and all participants were at least 18 years old. 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. The same participant was not allowed to complete multiple surveys on color modification. In total 2,910 unique participants evaluated 5,900 Facebook samples, and Table 1 provides study statistics.

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 1: Statistics on our Facebook user study.

4.3 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. After that, 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?"* with Yes and No reply alternatives.
- *"If you would see this screen, would you login with your real Facebook password?"* with Yes and No reply alternatives.

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

4.4 Results

We discarded survey responses where the study participants did not indicate active usage of the Facebook application 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 1). Table 2 shows demographics of our study participants.

Color modification. The color dataset results are illustrated in Figure 4. We plot the observed login rate in green

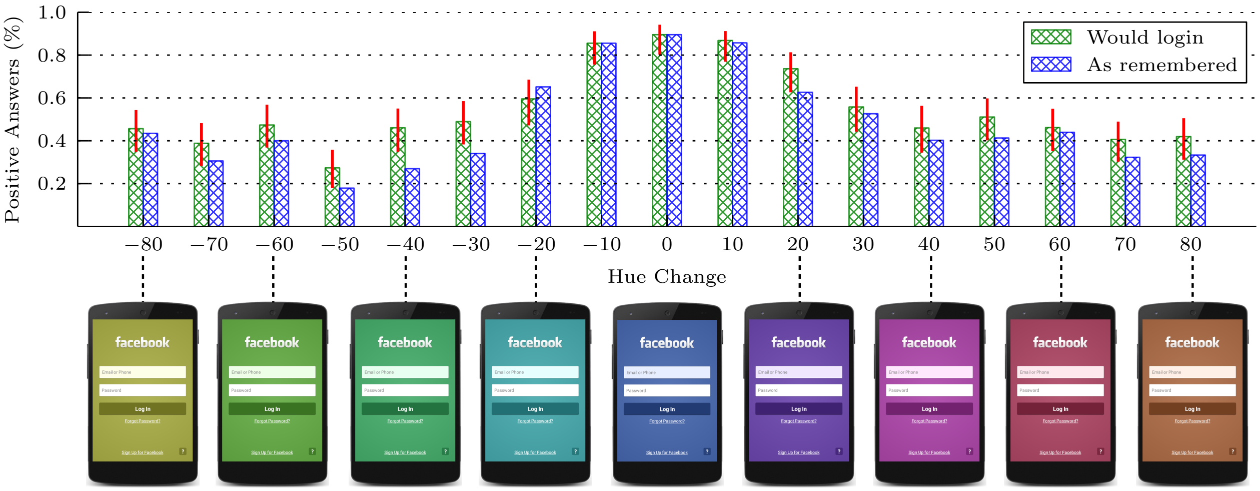

Figure 4: Percentages of users that perceived a Facebook login screen sample with modified color as genuine (*as-remembered rate*) and would login to the application if such a screen is shown (*login rate*). Color has a significant effect on both rates.

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 2: Demographics on Facebook user study.

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 significance level p=.05 to examine the relation between login responses and the sample color. The relation between these variables was significant ($\chi^2(16, N = 1551) = 194.44), p < .001$) 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 5. Both element reordering ($\chi^2(5, N = 546) = 15.84, p = .007$) as well as scaling ($\chi^2(9, N = 916) = 245.56, p < .001$) had an effect on 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 smaller than half of their original size, 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 6. The relation between the login behavior and the amount of crop was significant ($\chi^2(5, N = 540) = 83.75, p < .001$). As an interesting observation 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 only a partial logo. However, our study was not designed to prove or reject such hypothesis.

Amount of noise in the logo had an effect on login rates ($\chi^2(4, N = 460) = 75.30, p < .001$), and 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 ($\chi^2(4, N = 462) = 57.25, p < .001$), and even a modest rotation of five degrees caused the login rate to drop noticeably. Similarly, study participants were less likely to login to screens with projected logo ($\chi^2(5, N = 542) = 102.45, p < .001$).

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 visual features are treated the same, are unlikely to be effective.

4.5 Follow-up Studies

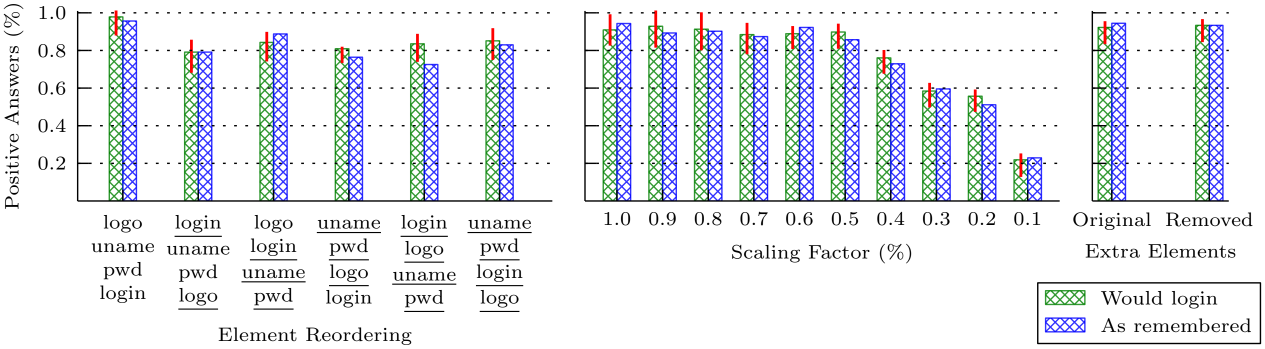

Figure 5: Percentages of users that perceived a Facebook login screen sample with general modifications as genuine (*as-remembered rate*) and would login to the application if such a screen is shown (*login rate*). Element reordering modification had a small but statistically significant effect, scaling caused a major effect, and extra element removal showed no effect.

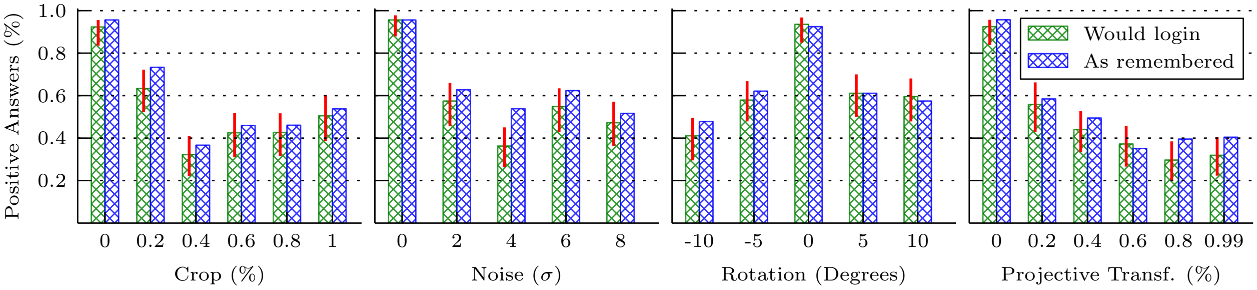

Figure 6: Percentages of users that perceived a Facebook login screen sample with logo modifications as genuine and would login to the application if such a screen is shown. All logo modifications caused a significant effect.

We performed similar follow-up studies for Skype and Twitter applications and, 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 use the collected data for training of our detection system.

4.6 Study Method

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 samples, for globally-distributed participants. A common approach in phishing studies is to observe participants during a login operation to a phishing screen. Scaling this method for the number of samples we wanted to evaluate 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 also influenced the results of our study.

5. SPOOFING DETECTION SYSTEM

Through our user study, we gained insight into what kind of visual modifications users notice, and fail to notice. In this section we design and implement a prototype spoofing detection system to leverage this knowledge. The proposed system is applicable to different platforms and deployment models, and in what follows, we instantiate the system for Android devices.

Our system extracts screenshots on a mobile device and uses the results of the user-study to estimate their deception rate, with respect to a set of reference applications (e.g., all apps installed on the phone). Figure 7 shows a high-level system design. The system consist of four main components: (1) reference application analysis, (2) screenshot analysis, (3) estimator training and (4) deception rate estimation. The reference application analysis and estimator training can be performed offline (e.g., at the application marketplace). We have implemented those components as a modified Android emulator environment and as Python tools using the OpenCV [5] library for image processing and scikit-learn for estimator training. The deception rate estimation and screenshot analysis are performed on the mobile device at runtime. We have implemented those components on Android in Java using OpenCV, and we proceed by describing each part of the system in detail.

5.1 Reference Application Analysis

To analyze a screenshot with respect to a reference app, we first obtain the reference application login screen and identify its main elements (reference elements) according to our login screen model (Figure 2). We assume reference application developers that have no incentive to obfuscate their login screen implementations. This analysis is a one-time operation that can be done offline, e.g., on every app update, and its results distributed to the mobile devices.

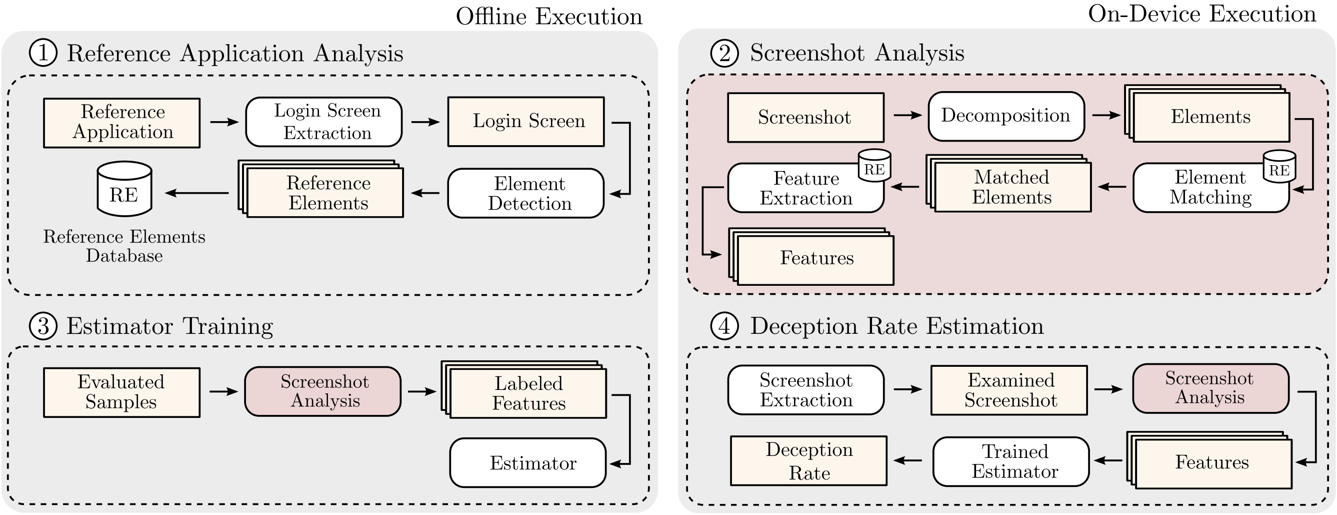

Figure 7: Overview of our spoofing detection system that leverages knowledge of user mobile application similarity perception. The system consist of four parts: (1) reference application analysis, (2) screenshot analysis, (3) estimator training and (4) deception rate estimation.

Login screen extraction and element detection are platform specific operations. On Android applications, windows and their contained elements are represented as *activities*. To find which activity represents the login screen, we developed a tool in the form of a modified Android runtime environment that is executed inside an emulator and that hooks activity and user interface element creation events. Similar analysis can be implemented by instrumenting the reference application, but we chose to modify the runtime environment in order to make the analysis more robust.

When the reference application is started, the tool hooks the creation of the first activity and then searches the element tree list of the activity for a password input field. We identify the first encountered text box element 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 with respect to our model. 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 base 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 by, e.g., pressing a button. The tool identifies buttons, triggers each of them, and hooks any new created activity. For each of the new activities, the tool searches for a password field and, if one is not found, it 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 data structure and, for each element, stores its type, location, size, and content as a screen capture over the element area.

Our automatic tool was designed to analyze benign reference apps. We performed a preliminary test, and in all 10 tested apps the tool correctly identified the activity holding the login screen as well as its type (standalone or distributed). Benign developers have an incentive to make their login screens easy to detect, and can specify the activity that is the login screen (e.g., in the Android manifest file).

5.2 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 (see Figure 7).

Decomposition. Screenshot decomposition is illustrated in Figure 8. First, we perform common edge detection and then dilate all detected edges to fill small areas such as text. We then 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 which share a border get detected as two separate elements. We run a connected components algorithm to identify the regions. Finally, we filter regions smaller than a threshold value, and we place a bounding box around each detected area. 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 bounding box of another element. For each element, we store its location, size, and screenshot of its area.

Element matching. The next step is to match the detected elements to the reference elements, as illustrated in Figure 9. To identify which element is the closest match to the reference logo, we use a known image feature extractor. While SIFT extractors [17] have been successfully applied for detection of logos in natural images [26], we found SIFT to be ill-suited for mobile application logos, especially in cases where only partial (e.g., cropped) logos were present. The shapes of mobile app logos are typically smooth, compared to the ones seen in natural images, and have small dimen-

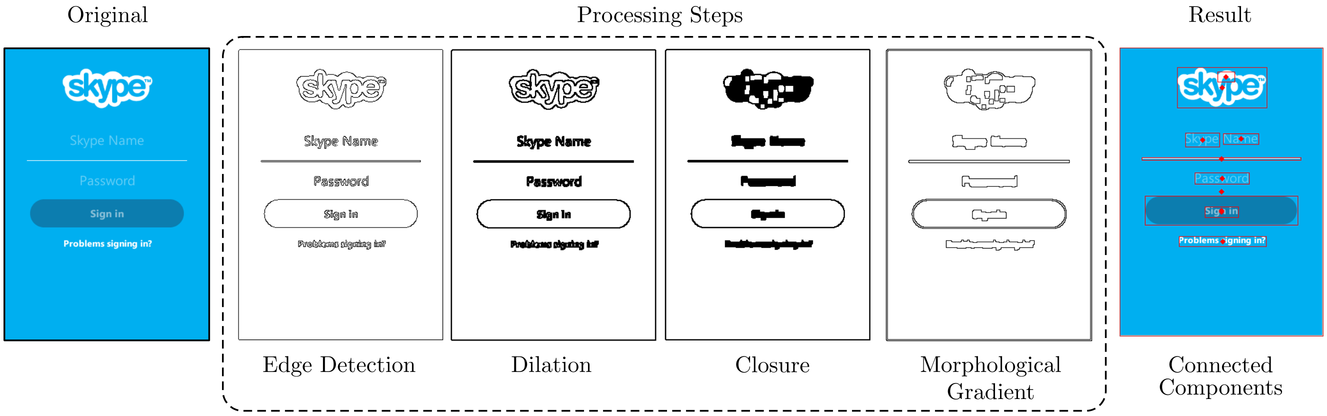

Figure 8: Decomposition example on the Skype login screen. The processing steps in the middle includes common image analysis techniques. The final step is a connected components algorithm as well as filtering out smaller regions. For visual clarity, we inverted the colors in the processing steps.

sions, and SIFT was unable to identify enough keypoints for accurate detection. We found that the ORB feature extractor [25] performed better in our context.

Matching the reference application logo to an element in the examined application works as follows. We compute ORB keypoints over the reference logo as well as the whole examined screenshot and we match the two sets. The element that matches with the most keypoints 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 9).

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

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.

Our analysis is designed to extract features from screenshots that follow the login screen model.

5.3 Estimator Training

As deception rate, or 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.

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) participants. We omitted study samples from the training data that express visual modifications that our current system implementation is not able to extract (noise).

We experimented with regression models of different complexities and trained two linear models (Lasso and linear regression), a decision tree, as well as two ensemble learning methods (gradient boosting and random forests). 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. This was the most observed range in our study.

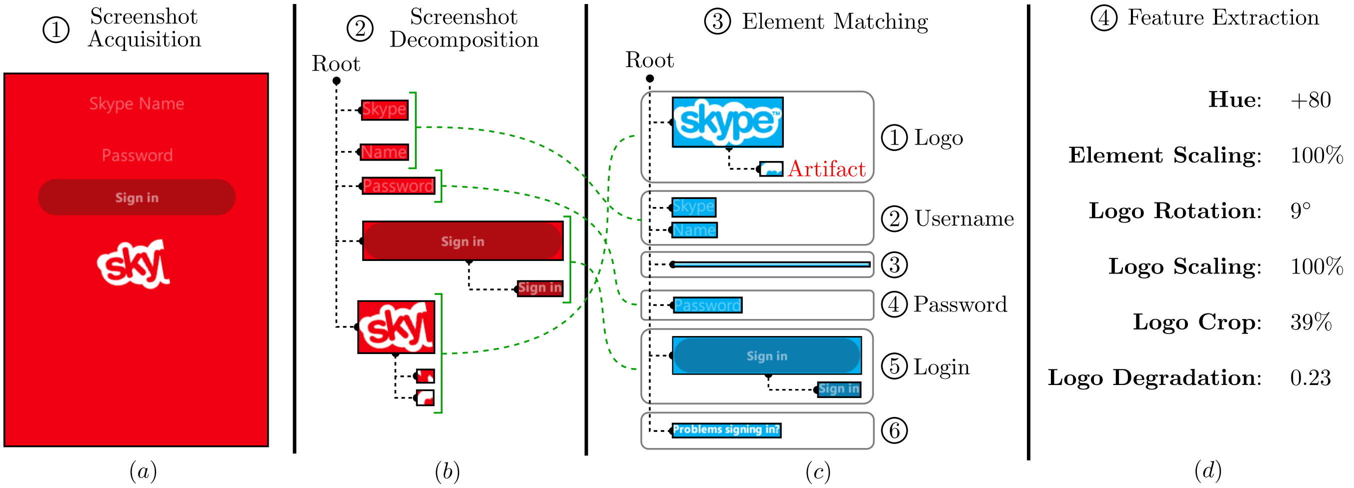

Figure 9: Summary of screenshot analysis. (a) The starting point is a mobile application login screenshot. (b) We decompose the screenshot to a tree hierarchy. (c) We match the detected elements to reference elements. (d) Finally we extract features from the detected elements with respect to the reference elements.

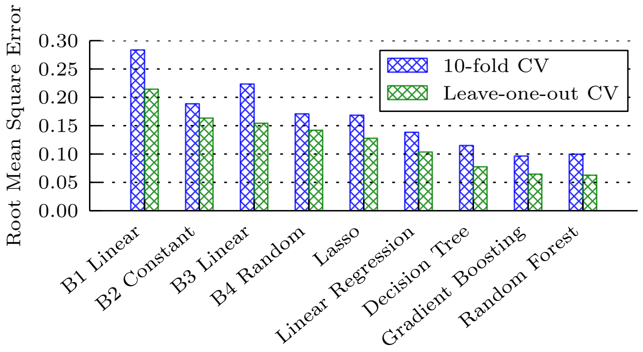

Figure 10: Evaluation of five regression and four baseline models (B1–B4) on combined datasets of Facebook and Skype. The random forest regressor performs the best.

5.4 Deception Rate Estimation

The goal of the final part of the system is to extract (and pre-process) screenshots on the mobile device and then estimate how many users would confuse the examined screenshot with the reference application.

Screenshot extraction. Obtaining the current contents of the screen is a platform specific operation. On Android, we grab the contents directly from the frame-buffer device.

Deception rate estimation. To estimate the deception rate, we extract features from the obtained 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. For example, if the deception rate is higher than a chosen threshold, the system can flag the examined application as fraudulent.

To evaluate how well our trained models generalize, we performed two types of model validation: leave-one-out and 10-fold cross-validation. We report the results in Figure 10 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.

	Galaxy S2	**Nexus 5**
Screenshot extraction	10 ± 3 ms	**NA**
Decomposition	99 ± 19 ms	41 ± 10 ms
Element (logo) matching	147 ± 35 ms	54 ± 16 ms
Feature extraction	150 ± 34 ms	67 ± 12 ms
Estimator	0.5 ± 0.9 ms	0.1 ± 0.3 ms
Total	407 ± 69 ms	162 ± 28 ms

Table 3: On-device Android implementation performance. At submission time, our Nexus 5 device was not rooted and we were not able to directly access the frame-buffer device. We expect the screenshot extraction time on a Nexus 5 to be less than on Galaxy S2 (e.g., less than 5ms).

5.5 Performance Evaluation

In this section we provide the performance evaluation for the on-device Android implementation or our system. We do not evaluate the performance of the offline components as they are not executed on the phone, and are not time-critical operations. We performed our measurements on a mid-range (Samsung Galaxy S2) as well as a high-end smartphone device (Nexus 5), and Table 3 shows execution times averaged over 100 runs. In total, a single deception rate estimation takes 407 ms (Galaxy S2) and 162 ms (Nexus 5).

A simplistic example deployment model for on-device spoofing detection is to extract and analyze screenshots periodically when the phone is in-use, as analysis of every frame is too expensive. A recent study showed that mobile banking application login operations were measured to last from 4 to 28 seconds [19]. Therefore, extracting screenshots every 4 seconds would allow analysis of most spoofed login screens and the performance overhead would be in the order of 10% and 4% for the Galaxy S2 and Nexus 5 phones respectively. This simple deployment model incurs significant performance overhead, but can be made more efficient in the following ways.

Pre-filtering. One approach is to first perform a less expensive pre-filtering operation to determine if the examined screenshot resembles a login screen and, if so, perform further examination. Decomposition of a lower resolution screenshot could be such a pre-filtering operation. The detection system could continue with full screenshot analysis only if decomposition provides a number of elements (or similar heuristic) that is close to the login screen model. This approach would avoid the expensive analysis for all screenshots, as most of them are, after all, benign.

Reduced resolution. Performance speed 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.

Application whitelisting. A user can select a set of trusted applications (e.g., Facebook, Twitter, Skype). The system would perform screenshot analysis only during times when a non-trusted app is running (e.g., there is possibility of a spoofing attack taking place).

Collaborative detection. To further reduce the detection system performance penalty, the phone can utilize the *many-eyes* principle and toggle down the screenshot acquisition frequency based on application marketplace popularity. For example, if the app currently drawing on the screen has high popularity, our system can acquire screenshots less often, as the same screenshot analysis would be performed by a larger number of different phones. If any one of those phones detects a potential spoofed screen, it informs the marketplace and the information is then subsequently distributed to the remaining app users.

The goal of our implementation was to demonstrate the feasibility of our new approach, rather than to provide a thorough performance evaluation of a fully integrated system. Our initial tests and evaluation show that user perception based spoofing detection can be both accurate and practical to deploy, and the exact deployment model and scheduling of screenshots we leave as future work.

5.6 Avoiding Detection

In this section we discuss possible strategies an attacker can use to evade detection.

Fixed screenshot times. In case the screenshots are taken at fixed time intervals, an attacker can avoid detection by displaying the spoofed screen at all times, other than the ones when screenshots are acquired. To prevent such an attack, the screenshot acquisition can be performed in a randomized manner, e.g., every $t \pm r$ seconds, where r is a random time offset.

Averaging attacks. 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 phishing 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.

Different types of spoofing screens. While the adversary has an incentive to create phishing 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.

6. RELATED WORK

Spoofing detection systems. In a recent work, Bianchi et al. [4] have developed a static analysis tool for mobile device application spoofing detection that identifies API calls that enable background spoofing attacks, and their work is complementary to ours. 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). A noteworthy difference is that while this tool is tailored to efficient detection of known attack vectors, our system takes as an input the end product of any spoofing attack—a screenshot presented to the user—and thus our approach applies also to the detection of previously unseen attack types.

Many web phishing detection systems analyze a website DOM tree and compare its elements and structure to the reference site [3, 16, 24, 35, 36]. 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 approach is to consider the visual presentation of a spoofing application (or a website), and compare its similarity to a reference value [7, 13, 20]. 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.

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, 11, 28]. 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 [19]. 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 [9, 27, 31]. Recent studies show that personalized security indicators can be more effective on mobile applications [19]. We are the first to study how likely the users are to notice spoofing attacks, where the malicious application resembles the legitimate application.

7. CONCLUSION

In this work we addressed the problem of mobile application spoofing detection. We proposed a novel application spoofing detection system that analyses the visual appearance of application login screens and measures their similarity to reference login screens. We express this similarity in terms of a new metric — deception rate, which represents the fraction of users that would confuse the candidate for the reference login screen. To form a basis for an accurate estimation of the deception rate we leveraged an extensive on-line user study. Our results show that deception rate estimation is a viable approach for mobile application spoofing detection, and that our system is able to estimate the deception rate with 6% error margins on our dataset. In addition to supporting a spoofing detection system, the results of our user study, on their own, provide an insight into perception and attentiveness of users to login screen features 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] 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.

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

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

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

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

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

[18] 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/.

[19] 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.

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.