Mobile Application Impersonation Detection Using Dynamic User Interface Extraction

Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun

Institute of Information Security
ETH Zurich
firstname.lastname@inf.ethz.ch, michael.och@alumni.ethz.ch

Abstract. In this paper we present a novel approach for detection of mobile app impersonation attacks. Our system uses dynamic code analysis to extract user interfaces from mobile apps and analyzes the extracted screenshots to detect impersonation. As the detection is based on the visual appearance of the application, as seen by the user, our approach is robust towards the attack implementation technique and resilient to simple detection avoidance methods such as code obfuscation. We analyzed over 150,000 mobile apps and detected over 40,000 cases of impersonation. Our work demonstrates that impersonation detection through user interface extraction is effective and practical at large scale.

Keywords: mobile, visual, repackaging, phishing, impersonation

1 Introduction

Mobile application *visual impersonation* is the case where one application intentionally misrepresents itself in the eyes of the user. Such applications impersonate either the whole, or only a small part of the user interface (Figure 1). The most prominent example of whole UI impersonation is application repackaging; the process of republishing an app to the marketplace under a different author. It's a common occurrence [39] for an attacker to take a paid app and republish it to the marketplace for less than it's original price. In such cases, the repackaged application is stealing sales revenue from the original developers.

In the context of mobile malware, the attacker's goal is to distribute a malicious application to a wide user audience while minimizing the invested effort. Repackaging a popular app, and appending malicious code to it, has become a common malware deployment technique. Recent work [40] showed that 86% of analyzed malware samples were repackaged versions of legitimate apps. As users trust the familiar look and feel of their favourite apps, such a strategy tricks the user into believing that she is installing, and interacting with, a known app.

Application fingerprinting [36,15,16,37,41,14,38,12,4] is a common approach to detect repackaging. All such works compare fingerprints extracted from some feature of the application, such as their code or runtime behaviour. However, an adversary that has an incentive to avoid detection can *easily modify* all such features without affecting the appearance of the app, as seen by the user. For

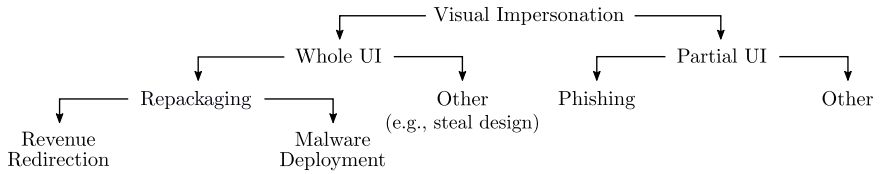

Fig. 1. Taxonomy of mobile app visual impersonation. Existing works primarily focused on detecting repackaging. Our goal is to detect all types of impersonation.

example, an attacker can obfuscate the app by adding dummy instructions to the application code. Code comparison approaches would fail because the new fingerprint would be significantly different, and we demonstrate that such detection avoidance is both effective and simple to implement.

Instead of impersonating the whole UI, malicious apps can also impersonate only a small part of the UI by, e.g., creating a fake login screen in order to phish login credentials. Phishing apps that target mobile banking have become a recurring threat, with serious incidents already reported [26,28]. Prior works on repackaging detection and common malware detection [4,24,32,31,13] are ill-suited for detecting such phishing cases as the malicious apps share little resources with the original, they don't exhibit specific system call patterns, nor do they require any special permissions — they only draw to the device screen.

Due to these inherent limitations of previous detection techniques, we propose a conceptually different approach. Our goal is to design an impersonation detection system that is resistant to common detection evasion techniques (e.g., obfuscation) and that can be configured to efficiently detect different types of impersonation; from repackaging (whole UI) to phishing (partial UI). We observe that, for visual impersonation to succeed, and irrespective of possible modifications introduced to the app, *the adversary must keep the runtime appearance of the impersonation app close to the original.* A recent study [19] showed that the more the visual appearance of a mobile app is changed, the more likely the user is to become alarmed. We propose a detection system that leverages this unavoidable property of impersonation.

Our system complements existing fingerprint-based approaches, runs on the marketplace and can analyze large amounts of Android apps using dynamic analysis and visual comparison, prior to their deployment onto the market. Our system runs the app inside an emulator for a specified time (e.g., 20 min), dynamically explores the mobile app user interface and extracts screenshots using GUI crawling techniques. If two apps have more than a threshold amount of screenshots in common (either exact or near matches), the apps are labeled as an instance of impersonation. In contrast to previous works, we do not base our detection on some easily modified feature of the app's resources, but rather on the final visual result of executing any app — the screenshot presented to the user. As a result, our system is robust towards attacker introduced perturbations to either application resources, or to the way the UI is created — *as long as the*

application looks the same at runtime, our system will detect the impersonation. No prior detection schemes offer this property.

To realize a system that is able to analyze a large number of apps, we had to overcome technical challenges. The existing GUI crawling tools [20,22,2] require application-specific knowledge or manual user input, and are therefore not applicable to automated, large-scale analysis. To address those challenges, we developed novel GUI crawling techniques that force the analyzed app to draw its user interface. Our system requires no user input, no application-specific knowledge, it supports the analysis of native applications, and thus enables automated analysis of apps at the scale of modern application marketplaces. Our system uses locality-sensitive hashing (LSH) [10] for efficient screenshots retrieval.

To evaluate our system, we dynamically analyzed over 150,000 applications downloaded from Google Play and other mobile app markets. Our system extracted approximately 4.3 million screenshots and found over 40,000 cases of impersonation; predominantly repacks (whole UI) but also apps that impersonate only a single registration screen (partial UI). These experiments demonstrate that impersonation detection through dynamic user interface extraction is effective and practical, even in the scale of large mobile application marketplaces.

To summarize, our main contributions are:

- We demonstrate that existing impersonation detection techniques can be easily avoided.
- We propose a novel approach for impersonation detection based on dynamic analysis and user interface extraction that is robust towards the way the adversary implements impersonation.
- We built a detection system for large-scale analysis of mobile apps and as a part of the system we developed novel UI exploration techniques.
- We analyzed over 150,000 applications and found both whole and partial UI impersonation instances.

The rest of this paper is organized as follows. In Section 2 we explain the problem of app impersonation in detail. We describe our solution in Section 3 and evaluate it in Section 4. We present our results in Section 5, and security analysis in Section 6. We discuss deployments aspects in Section 7 and review related work in Section 8. We conclude in Section 9.

2 Motivation and Case Study

Whole UI impersonation. The first type of impersonation we consider is whole UI impersonation (Figure 1). Since a typical repackaged app shares majority of its code and runtime behaviour with the original application, fingerprinting can be an effective way to detect such impersonation. Known techniques leverage static application features, such as code structure [12], imported packages [41] and dynamic features [4,24,32,31,13], such as system call [18] or network traffic patterns [25]. Using fingerprinting, large number of repacks have been detected

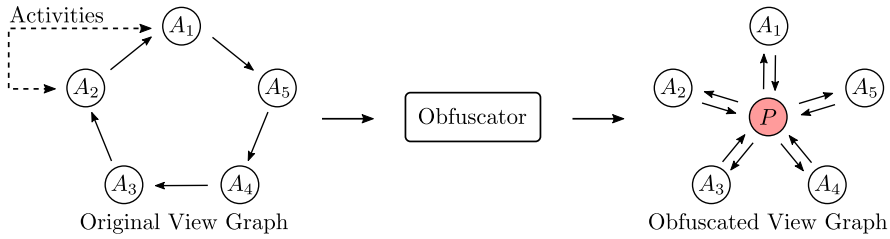

Fig. 2. We developed a small obfuscation tool that modifies the activity transition graphs of an app without affecting user experience. The obfuscation introduces additional proxy activities that mediate transitions between UI screens. In the original app, the user transitions (e.g., by clicking a button) $A_1 \rightarrow A_5$. Afterwards, the transition becomes $A_1 \rightarrow P \rightarrow A_5$. The transition occurs so fast that the user does not notice it.

from several Android application marketplaces [9]. The inherent limitation of fingerprinting approaches is that the adversary can easily modify the above listed (or any similar) features without affecting the runtime appearance of the application towards the user. In this section we demonstrate that such modifications are both easy to implement and effective in avoiding detection.

Case study. We present a case study on MassVet [9], a recent static analysis system that has demonstrated good results in detecting repackaging. MassVet extracts two types of static fingerprints from a mobile application: *m-cores* define application code flow graph and sequence of instructions and *v-cores* express UI state transitions from one activity to another. The rationale behind these fingerprints is that repackaged apps typically share majority of their functionality, as well as user interface structure.

We built a simple obfuscation tool (less than 200 lines of code) that modifies both extracted features (m-cores and v-cores). Our tool adds blocks of unused code, as well as proxy activities that modify activity transitions. Our modifications have no visible effect on the user experience (see Figure 2). To evaluate the tool, we took 1259 apps from the Android Malware Genome project [1]. We ran those apps through the MassVet tool online interface[1] and 559 applications were identified as potentially malicious repackaged apps. We obfuscated all of these apps using our tool and rerun them on the MassVet system — 92% of the obfuscated apps were no longer detected. We argue that similar limitations apply to any fingerprinting solution and adoption of obfuscation introduces little to no cost to repackaging adversaries, as it merely implies running an automated tool. We demonstrated only one possible approach, and such obfuscation tools can be implemented for all prior fingerprinting approaches (both static and dynamic), in a similar manner and level of complexity.

Partial UI impersonation. The second type of visual impersonation are apps that only impersonate a small portion of the UI. Application phishing is an example of impersonation, where a malicious application constructs a single screen that visually resembles one of a another app, but otherwise shares no

[1] `http://www.appomicsec.com/`

similarities (e.g., no shared code or resources) with the original app. For example, a malicious game asks the user to perform an in-app purchase, but instead of starting the legitimate banking app, the game presents a phishing screen that mimics the login screen of the bank. Distinguishing the phishing screen from the genuine login screen is difficult for the user, as modern mobile platforms do not enable the user to identify the application currently drawing on the screen.

Fingerprinting is not effective against such impersonation apps, as the malicious app does not share large parts of resources (e.g., code). Furthermore, traditional mobile malware detection schemes that, e.g., examine API call patterns [7] and permissions [6] are also ill-suited for the detection of phishing. Such applications differ from regular malware as they often require no special permissions, nor do they necessarily perform any suspicious actions. They only perform a single operation — *drawing on the device screen*.

Adversarial model. We consider a strong attacker that can implement impersonation in various ways. The attacker can create the UI by, e.g., using standard OS libraries, implement it in a custom manner, or show static images of the interface. On Android, the attacker could implement the app in Java, native code, or as a web app. For a more thorough introduction of the various ways of creating user interfaces in Android, we refer the reader to Appendix A. On top of that, the adversary can modify the application code or resource files in arbitrary ways (e.g., obfuscation). Such an adversary is both realistic and practical. The attacker can freely create the impersonated screens by any means allowed by the underlying Android system. Running an (off-the-shelf or custom) obfuscation tool comes at little to no cost to the adversary.

3 Visual Impersonation Detection System

As demonstrated in the previous section, the adversary has significant *implementation freedom* in performing an impersonation attack and, at the same time, the adversary has a clear incentive to keep the visual appearance of the app close to the original. A previous study [19] has shown that the more the adversary deviates from the appearance of the original mobile application user interfaces, the more likely the user is to become alarmed. We say that the adversary has limited *visual freedom* and our solution leverages this property.

Our goal is to develop a robust visual impersonation detection mechanism that is based on visual similarity — a feature the attacker cannot modify without affecting the success of the attack. More precisely, the system should: (i) detect both whole (e.g., repackaging) and partial UI impersonation (e.g., phishing), (ii) be robust towards the used impersonation implementation type and applied detection avoidance method (e.g., obfuscation), (iii) analyze large numbers of apps in an automated manner (e.g., on the scale of modern app stores).

Figure 3 shows an overview of our system that works in two main phases. In the first phase, the system extracts user interfaces from a large number of mobile apps in a distributed and fully autonomous manner. The system takes as input only the application binary, and requires no application-specific knowledge or

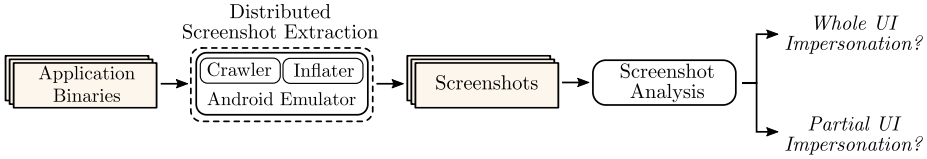

Fig. 3. An overview of the impersonation detection system that works in two phase. First, we extract user interfaces from a large number of applications. Then we analyze the extracted screenshots to detect repackaging and impersonation.

manual user input. We run each analyzed app in an emulated Android environment, and explore its user interface through crawling. An attacker could develop a malicious app that detects emulated environments. However our system can be executed on real hardware as well (Section 7). In a best-effort manner, we extract as many screenshot as possible within a specified time limit. Full exploration coverage is difficult to achieve [5,30], and as our results in Section 4 show, not necessary for effective impersonation detection. During crawling, our system also automatically identifies reference screens (e.g., ones with login and registration functionality) that benefit from impersonation protection.

In the second phase, the system examines the extracted screenshots to find cases of impersonation. To detect whole UI impersonation, our system finds applications that share the majority of their screenshots with a queried app. To detect partial UI impersonation, our system finds applications that share a similar (e.g., login or user registration) screen with the reference app, but have otherwise different screenshots.

Our system can be deployed on the marketplace and used to detect impersonation, e.g., upon submitting the app to the market. We emphasize that our system can be combined with existing approaches to enhance impersonation detection. For example, only apps that are considered as benign by a fast fingerprint-based approach could be submitted to our, more costly analysis.

3.1 Design Challenges

To realize the system outlined above, we must overcome a number of technical challenges. First, the existing GUI crawling approaches were designed for development and testing, and a common assumption is that the test designers have access to application-specific knowledge, such as source code, software specifications or valid login credentials. Login screens are a prominent example of how crucial application-specific knowledge is in GUI exploration, as such tools need to know a valid username and password to explore beyond the login screens of apps. Another example are games where, in order to reach a certain GUI state, the game needs to be won. In such cases, the exploration tool needs to be instructed how to win the game. Previous crawling tools address these issues of reachability limitations using application-specific exploration rules [5,33] and pre-defined crawling actions [34]. As our system is designed to analyze a large

number of mobile apps, similar strategies that require app-specific configuration are not possible. In Section 3.2 we describe a mobile app crawler that works fully autonomously, and in Section 3.3 we describe new user interface exploration techniques that increase its coverage.

Second, dynamic code analysis is significantly slower than static fingerprinting. In Section 3.2 we describe a distributed analysis architecture that enables us to analyze applications in a fully scalable manner. And third, many repackaged apps and known phishing malware samples [28] contain minor visual differences to their target apps. Our system must efficiently find screenshots that are exact or near matches, from a large set of screenshots. In Section 3.4 we describe a system that uses locality-sensitive hashing [10] for efficient screenshot analysis.

3.2 Automated Crawling

We designed and implemented a UI crawler as part of the Android core running inside an emulator (Figure 3). The crawler uses the following basic strategy. For each new activity, and every time the view hierarchy (tree of UI elements) of the current activity changes, our crawler takes a screenshot. The crawler continues exploration in a depth-first manner [5], as long as there are clickable elements (views) in the user interface. To support autonomous user interface exploration, our crawler must determine which views are clickable without prior knowledge. We implemented the crawler as a part of the Android core, which gives it full access to the state of the analyzed app, and we examine the current state (i.e., traverse the view tree) to identify clickable elements. We observed that in many apps, activities alter their view hierarchy shortly after their creation. For example, an activity might offload a lengthy initialization process to a background thread, show a temporary UI layout first, and the final one later. To capture such layout changes, our crawler waits a short time period after every transition.

To increase the robustness of crawling, we made an additional modification to the Android core. If the crawled application creates an Intent that triggers another app to start, we immediately terminate it, and resume the execution of the analyzed app. In practice this approach turned out to be an efficient way to continue automated user interface exploration.

Reference screen identification. To enable partial UI impersonation detection, our system automatically identifies screens that benefit from impersonation protection. We have tailored our implementation to identify screens that contain login or registration functionality. While crawling an app, we traverse the view hierarchy tree of each screen and consider the screen a possible login or user registration screen, when the hierarchy contains at least one password field, one editable text field, and one clickable element or meets other heuristics, such as the name of the activity contain word “login” or “register”. If such a screen is found, we save it as a *reference screen* for partial UI impersonation detection (Section 3.4). Reference screen identification is intended for benign apps that have no incentive to hide their UI structure, but we repeat the process for all crawled apps, since we do not know which apps are benign.

Distributed architecture. We built a distributed analysis architecture that leverages cloud platforms for analysis of multiple apps. Our architecture consists of a centralized server and an arbitrary number of analysis instances. The server has a database of apps, orchestrates the distributed analysis process, and collects the extracted user interfaces. Each analysis instance is a virtual machine that contains our dynamic analysis tools.

3.3 Coverage Improvements

In this section we describe techniques we implemented to increase the coverage of our user interface exploration.

Out-of-order execution. The crawler starts from the entry-point activity defined in the manifest file. For some apps only a small part of the user interface is reachable from the entry point without prior application-specific knowledge (e.g., a login screen requires valid credentials to proceed). To improve our crawling coverage, we additionally force the application to start all its activities out of order. This is a best-effort approach, as starting the execution from an arbitrary activity may crash the app without correct Intent or application state.

Layout file inflation. We implemented a tool that automatically renders (inflates) mobile app user interfaces based on XML layout files and web resources. As many apps customize the XML layouts in code, the final visual appearance cannot be extracted from the resource file alone. We perform resource file inflation from the context of the analyzed app which ensures that any possible customization will be applied to UI elements defined in the resource file. We implemented a dedicated enforcer activity and force each app to load it at startup. This activity iterates through the app's layout files, renders them one by one and takes a screenshot. We noticed that increasingly many mobile apps build their user interface using web technologies (e.g., HTML5). To improve the coverage of such apps, we perform similar inflation for all web resources. Our enforcer activity loads all local web resources of the application one by one.

Layout file inflation is conceptually different from, e.g., extracting fingerprints from resource files. The layout files can be modified by the attacker without affecting our analysis, as we take a screenshot of the final rendered layout.

User interface decomposition. Our crawling approach relies on the assumption that we can determine all clickable UI elements by analyzing the view hierarchy of the current activity. While this assumption holds for many apps, there are cases where clickable elements cannot be identified from the view tree, and therefore our crawler cannot proceed. For instance, mobile apps that implement their user interfaces in OpenGL (mostly games) and malicious apps that intentionally attempt to hide parts of their user interface from crawling.

We integrated a user interface decomposition outlined in [19] to our crawler to improve its coverage and attack resistance. Our experiments (Section 4) show that by using this extension we are able to crawl a number of mobile apps whose user interfaces we would not be able to crawl otherwise.

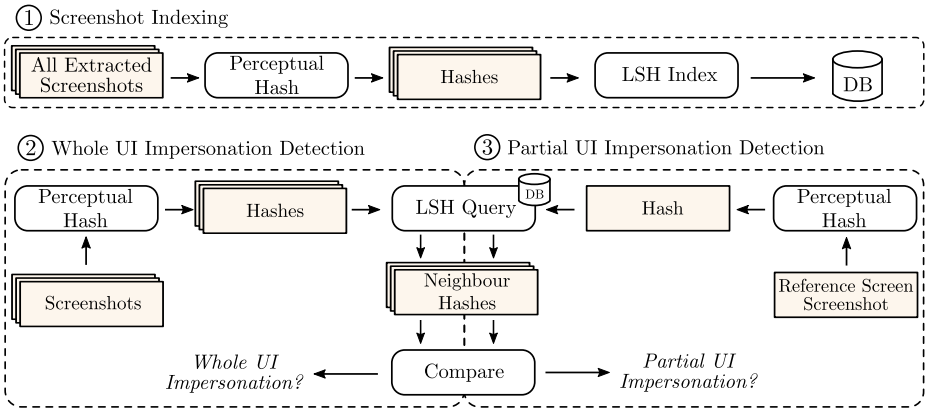

Fig. 4. Screenshot analysis system for impersonation detection. All extracted screenshots are indexed to a database using locality-sensitive hashing (LSH). To find impersonation applications from a large dataset, we first find nearest neighbor screenshots using the LSH database, and then perform an more expensive pairwise comparison.

3.4 Screenshot Analysis

Our initial experiments showed that many screenshots extracted from repackaged apps have minor visual differences in them. Some of the observed differences are caused by the application (e.g., different version, language or color scheme) while others are artifacts of our exploration approach (e.g., a blinking cursor visible in one screenshot but not in the other). Also the phishing screens seen in known malware samples contain minor visual differences to their target apps [28].

Our system handles visual differences in the extracted screenshots using *perceptual hashing*. The goal of a perceptual hash is to provide a compact representation of an image that maintains its main visual characteristics. In contrast to a cryptographic hash function, a small modification in the image produces a hash value that is close to the hash of the original image. Perceptual hashing algorithms reduce the size and the color range of an image, average color values and reduce their frequency. The perceptual hash algorithm we use [8] produces 256-bit hashes and the visual similarity of two images can be calculated as the Hamming distance between two hashes. To enable analysis of large number of screenshots, we leverage *locality-sensitive hashing* [11] (LSH). LSH algorithms are used to perform fast approximate nearest neighbour queries on large amounts of data.

Using these two techniques (perceptual hashing and LSH), we built a screenshot analysis system, as shown in Figure 7. The system consists of three operations: (1) indexing, detection for (2) whole and (3) partial UI impersonation.

Indexing. In the indexing phase, a perceptual hash is created for each extracted screenshot and all hashes are fed to our LSH implementation. We use bit sampling for reducing the dimensionality of input items, as our perceptual hashing algorithm is based on the Hamming distance. We set the LSH parame-

ters through manual experimentation (Section 4). To reduce false positives, we manually created a list of screenshots belonging to shared libraries (e.g., ad or game frameworks), and we removed all such screenshots from the database.

Whole UI impersonation detection. To detect impersonation, we use the intuitive metric of *containment* (C), i.e., how many screenshots of one app are included in the screenshot set of another app. To find whole UI impersonation apps, our system takes as input a queried app, and finds all apps whose user interface has high similarity to it. We note that, while our system is able to detect application pairs with high user interface similarity, it cannot automatically determine which of the apps in the relationship (if any) is the original one.

Our system analyzes one app at a time, and the first step in whole UI impersonation detection is to obtain the set of all screenshots (Q) of the current queried app. For every screenshot, we hash it and query the indexed LSH database. For each query, LSH returns a set of nearest neighbour screenshot hashes, their app identifiers and distances to the queried hash. For returning the nearest neighbours, we use the cutoff hamming distance $d = 10$, as explained in Section 4.

We sort this dataset based on the application identifiers to construct a list of candidate applications, where P_i is the set of their screenshots. For each candidate app we find the best screenshot match to the queried app. As a result, we get a set of matching screenshots for each candidate app. We consider the candidate app as a potential impersonation app when (1) the ratio (containment C) between the number of matched app and reference app screenshots is larger than a threshold T_w, and (2) the number of considered screenshots meets a minimum threshold T_s. Without loss of generality, we assume $|P_i| \leq |Q|$.

$$C = \frac{|P_i \cap Q|}{|P_i|} \geq T_w \tag{1}$$

$$|P_i| \geq T_s \tag{2}$$

In Section 4 we describe the experiments we used to set these thresholds (T_w, T_s). To find all potential repacks from a large dataset of applications, the same procedure is repeated for each application.

Partial UI impersonation detection. For partial UI impersonation, we no longer require a significant visual similarity between the two apps (target and malware). Only specific screens, such as login or registration screens, must be visually similar to perform a convincing impersonation attack of this kind. To scan our dataset for such applications, we adjusted the search criteria accordingly. Given our set of potential reference screens (Section 3.2) extracted during our dynamic analysis phase, we target applications that contain the same or a very similar screen but otherwise do not share a significant visual similarity in other aspects of the application. We only consider applications to be of interest if their containment with the queried application is less than a threshold ($C \leq T_p$), as long as the app contains the queried login or registration screen.

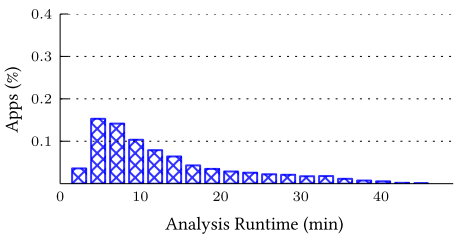

Fig. 5. The distribution of analysis time.

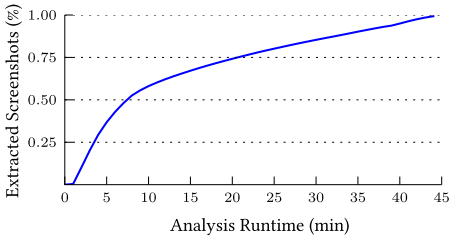

Fig. 6. Average number of screenshots extracted from an app, as a function of time.

4 Evaluation

In this section we evaluate the detection system. For evaluation we downloaded 158,449 apps from Google Play and other Android application repositories (see Table 1 in the Appendix). From Google Play we downloaded approximately 250 most popular apps per category. Our rationale was that popular apps would be likely impersonation targets. We also included several third-party markets to our dataset, as repacks are often distributed via third-party stores [9].

4.1 User Interface Extraction

Analysis time. We deployed our system on the Google Cloud Compute platform. The analysis time of a single application varies significantly, as apps have user interfaces of different size and complexity. While the majority of apps have less than 20 activities, few large apps have up to 100 activities. Furthermore, some applications (e.g., games) may dynamically create a large number of new user interface states and in such cases the dynamic user interface exploration is an open ended process. To address such cases, in our tests we set the maximum analysis time to generous 45 minutes. On the average, the analysis of one mobile app took only 7 minutes and we plot the distribution of analysis time in Figure 5. Extracting screenshots is the most time-consuming part of our system. Once the screenshots are extracted, querying LSH and deciding if any impersonation exists is fast (few seconds per app).

On 1000 computing instances on the Google Cloud Compute platform the entire analysis of over 150,000 apps took 36 hours. On the same rate and similar computing platform, the entire Google Play market (1.6 million apps) could be analyzed in approximately two weeks. We consider this a feasible one-time investment and the overall analysis time could be further reduced by limiting the maximum analysis time further (e.g., to 20 minutes as discussed below).

Analysis coverage. From the successfully analyzed 139,656 apps we extracted over 4.3 million screenshots after filtering our duplicates from the same application and low-entropy screenshots, e.g., single-color backgrounds that are not useful in impersonation detection. The majority of applications produced less than 50 screenshots, but we were able to extract up to 150 screenshots from

some apps. Figure 6 plots the percentage of extracted screenshots as a function of analysis time. We extract approximately 75% of all the screenshots during the first 20 minutes of analysis. The steeper curve during the first 7 minutes of the analysis is due to the fact that we run the inflater tool first and after that we start our crawler. The crawler tool extracted 58% of the screenshots and the inflater contributed additional 42%. Majority (97%) of the extracted screenshots come from user interfaces implemented using standard Android user interface elements and a small share originates from user interfaces implemented using web techniques or as an OpenGL surface. Figure 10 summarizes the user interface extraction results.

We use *activity coverage* as a metric to compare the coverage of our crawler to previous solutions. Activity coverage is defined as the number of explored activities with respect to the total number of activities in an app [5]. While activity coverage does not account for all possible user interface implementation options, it gives a good indication of the extraction coverage when analyzing large number of apps. Our tool achieves 65% activity coverage, and is comparable to previous solutions (e.g., 59% in [5]). However, previous crawling tools that achieve similar coverage require application-specific configuration or manual interaction.

We separately tested our user interface decomposition extension on 200 apps (mostly games) that implement their UI using OpenGL surface. Without the decomposition extension, our crawler was only able to extract a single screenshot from each of the tested apps (the entry activity). With decomposition, we were able to extract several screenshots (e.g., 20) from 30% of the tested apps. This experiment demonstrates that there is a class of mobile apps whose user interfaces can be crawled with better coverage using decomposition.

4.2 Screenshot Comparison

Perceptual hash design. We investigated two different perceptial hashing techniques, based on image moments [29] and DCT [8] (pHash). Preliminary analysis of the results from both hashing techniques revealed that the image moments based approach is not suitable for the hashing and comparing of user interface screenshots, as both very similar as well as very dissimilar pairs of screenshots resulted in almost equivalent distances. The *pHash* [35] based approach yielded promising results with a good correlation between visual similarity of user interface screenshots and the hamming distance between their respective hashes. Screenshots of user interfaces are quite different from typical natural images: clearly defined boundaries from UI elements, few color gradients, and a significant amount of text. To account for these characteristics and improve the performance of the hashing algorithm, we performed additional transformations (e.g., dilation) to the target images as well as the final hash computation.

To find a good cutoff hamming distance threshold below which we consider images to be visually similar, we randomly selected pairs of images and computed their distance until we had 100 pairs of screenshots for the first 40 distances. We then manually investigated each distance bucket and counted the number of

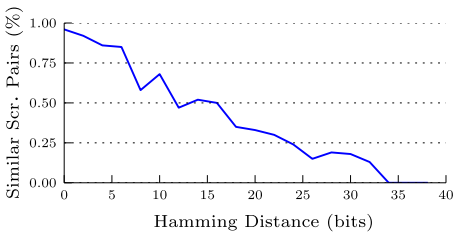

Fig. 7. Manual evaluation of hamming distances of screenshot hashes.

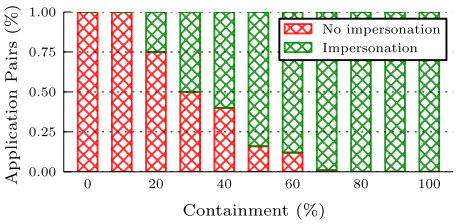

Fig. 8. Manual evaluation of false positives and false negatives.

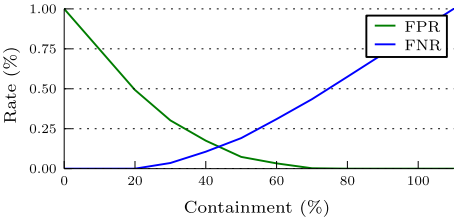

Fig. 9. False positive and false negative rates. Equal-error rate is at cont. 45%.

Number of successfully analyzed apps	**139,656**
Number of extracted screenshots	**4,302,413**
Extracted by the crawler	57.89%
Extracted by the inflater	42.11%
Originated from widget-based UI	96.95%
Originated from a HTML asset	2.17%
Originated from an OpenGL surface	0.88%

Fig. 10. User interface extraction results.

screenshot pairs we considered to be visually similar. The result are shown in Figure 7, and we concluded that a distance $d = 10$ is a reasonable threshold.

Containment threshold. Similar to the distance threshold evaluation, we randomly selected pairs of apps from our dataset and computed their containments, creating 10 buckets for possible containment values between $[0, 100]$ percent, until we had 100 unique pairs of apps for all containment values. We then manually examined those apps and counted the number of application pairs which we consider to be whole UI impersonations, as shown in Figure 8. The false negative and false positive rates are shown in Figure 9, yielding $T_w = 0.45$ containment to be a reasonable threshold for whole UI impersonation detection, above which applications within a pair are considered impersonation cases of each other. To verify that our manually derived false negatives rates were representative, we performed an additional check outlined in Appendix B. To detect partial UI impersonation, in our experiments, we found that setting the containment threshold to $T_p = 0.10$ gave good results.

5 Detection Results

To demonstrate that our system is able to catch impersonation attacks, we analyzed each application in our dataset and we report the following results.

Whole UI impersonation. Using perceptual hash distance $d = 10$ and containment threshold $T_w = 0.45$, our system detected 43,904 impersonating apps out of 137,506 successfully analyzed apps. Such a high number of impersonation instances is due to the fact that a large part of our apps are from third-party

markets. From the set of detected apps, our system does not automatically detect which apps are originals and which are impersonators.

At $T_w = 0.45$ our system has an estimated 15% false negatives. We remind the reader that these false negatives refer to missed *application pairs*, and not missed repacks. To illustrate, let us consider a simple example where our system is deployed on a market, and 3 apps are submitted (the original Facebook app, and 2 apps that impersonate it). The first app will be added and no impersonation can be detected. On the second app, our system has a 15% to miss the relation between the queried app and the one already in the store. However, the third app has approximately only $0.15^2 = 0.02$ chance of missing both relations and not being identified as impersonation with regards to the other two apps in the store. During our experiments, we found instances of clusters containing up to 200 apps, all repackaging the same original app.

Similarly, at $T_w = 0.45$ our system has also 15% false positives, which is arguably high. However, due to the fact that apps are repacked over and over again, on a deployed system we can set the containment threshold to be higher (e.g., $T_w = 0.60$). At that value, our system has only 3% false positives, and 31% false negatives. However, if the app is repacked a modest number of 5 times, the chances of missing all impersonation relationships becomes less than half a percent $0.31^5 = 0.002$.

In the above examples, we assumed that the analysis of each app is an independent event. In reality, this may not be the case. For example, if one impersonation app is missed, a closely related impersonation app may be missed with higher probability (i.e., the events are not independent). However, the more apps impersonate an original app, the higher the chances of our system catching it.

Partial UI impersonation. Using the metric described in Section 3.4, and $T_p = 0.10$, we found approximately 1,000 application pairs that satisfy the query. We randomly selected and manually inspected 100 pairs of apps to understand their relationships. In most cases, we found repackaged applications with a large number of additional advertising libraries attached.

Among these results, we also found an interesting case of a highly suspicious impersonation application. In this sample, the impersonation target is a dating app. The registration screen of the impersonation app appears visually identical to original. However, manually inspecting the code reveals that a new payment URL has been added to the application, no longer pointing to the dating website, but instead to a different IP. We uploaded the sample to virustotal.com to confirm our suspicions, and the majority of scanners indicated maliciousness. The code similarity between the original and impersonating apps (according to Androguard) is only 22%, largely due to added advertising libraries.

Our similarity metric allows us to find specialized kinds of impersonation attacks that deviate from the more primitive repackaging cases. Interesting user interfaces with certain characteristics (e.g. login behaviour) can be queried from a large data-set of analysed applications to find various kinds of impersonation, drastically reducing the necessary manual verification done by humans.

6 Security Analysis

Detection avoidance. Our user interface extraction system executes applications in an emulator environment. A repackaging adversary could try to fingerprint the execution environment and alter the behavior of the application accordingly (e.g., terminate execution if it detects emulator). The obvious countermeasure is to perform the user interface exploration on real devices. While the analysis of large number of apps would require a pool of many devices (potentially hard to organize), user interface exploration would be faster on real devices [21] compared to an emulator. Our user interface extraction system could be easily ported from the emulator environment to real Android devices.

A phishing adversary could construct the phishing screen in a way that complicates its extraction. To avoid the inflater, the adversary can implement the phishing screens without any resource files. To complicate extraction by crawling, the adversary could try to hide the phishing screen behind an OpenGL surface that is hard to decompose and therefore explore or make the appearance of the phishing screen conditional to an external event or state that does not necessarily manifest itself during the application analysis. Many such detection avoidance techniques reduce the likelihood that the phishing screen is actually seen by the user (a necessary precondition of any phishing attack). While our solution does not make impersonation impossible, it raises the bar for attack implementation and reduces the chances that users fall for the attack.

7 Discussion

Improvements. Our crawler could be extended to create randomized events and common touch screen gestures (e.g., swipes) for improved coverage. Our current out-of-order execution could be improved as well. Currently, startup of many activities fails due to missing arguments or incompatible application state. One could try to infer these from the application code through static analysis. Static code analysis could be also used to identify sections of application code that perform any drawing on the screen and the application could be attempted to force the execution of this code segment, e.g., leveraging symbolic execution. We consider such out-of-order execution a challenging problem.

Deployment. The primary deployment model we consider is one where a marketplace, mobile platform provider, anti-virus vendor or a similar entity wants to examine a large number of mobile apps to detect impersonation in the Android ecosystem. As shown in Section 4, the analysis of large number of apps requires significant resources, but is feasible for the types of entities we consider. Once the initial, one-time investment is done, detection for new applications is inexpensive. User interface extraction for a single application takes on the average 7 minutes and finding matching repacks or phishing apps can be done in the matter of seconds. As a comparison, the Google Bouncer system dynamically analyzes each uploaded app for approximately 5 minutes [23]. Our system streams screenshots to the central analysis server as they are extracted. The system can therefore stop further analysis if a decision can be made.

Post-detection actions. Once an application has been identified as a potential impersonation app, it should be examined further. Ad revenue stealing repacks could be confirmed by comparing the advertisements libraries and their configurations in two apps with matching user interfaces; sales revenue stealing repacks could be confirmed by comparing the publisher identities; repacks with malicious payload could be detected using analysis of API calls [3], comparison of code differences [9], known malware fingerprints or manual analysis; and phishing apps could be confirmed by examining the network address where passwords are sent. Many post-detection actions could be automated, but implementation of these tools is out of scope for this work.

8 Related Work

Repackaging detection. The previous work on repackaging detection is mostly based on static fingerprinting. EagleDroid [27] extracts fingerprints from Android application layout files. ViewDroid [36] and MassVet [9] statically analyze the UI code to extract a graph that expresses the user interface state and transitions. The rationale behind these works is that while applications code can be easily obfuscated, the user interface structure must remain largely unaffected. We have showed that detection based on static fingerprinting can be easily avoided (Section 2) and our solution provides more robust repackaging detection, at the cost of increased analysis time.

Phishing detection. The existing application phishing detection systems attempt to identify API call sequences that enable specific phishing attacks vectors (e.g., activation from the background when the target application is started [7]). While such schemes can be efficient to detect certain, known attacks, other attacks require no specific API calls. Our solution applies to many types of phishing attacks, also ones that leverages previously undiscovered attack vectors.

User interface exploration. The previous work on mobile application user interface exploration focuses on maximizing GUI coverage and providing means to reason about the sequence of actions required to reach a certain application state for testing purposes [2,5,17,33]. These approaches require instrumentation of the analysis environment with application-specific knowledge, customized companion applications, or even source code modifications to the target application. Our crawler works autonomously and achieves similar (or better) coverage.

9 Conclusions

In this paper we have proposed and demonstrated a novel approach for mobile app impersonation detection. Our system extracts user interfaces from mobile apps and finds applications with similar user interfaces. Using the system we found thousands of impersonation apps. In contrast to previous fingerprinting systems, our approach provides improved resistance to detection avoidance, such as obfuscation, as the detection relies on the final visual appearance of the examined application, as seen by the user. Another benefit of the system is that it can

be fine-tuned for specialized cases, like the detection of phishing apps. The main drawback of our approach is the significantly increased analysis time compared to static analysis. However, our experiments show that impersonation detection at the scale of large application stores is practical. Finally, the novel user interface exploration techniques that we have developed as part of this work, may have numerous other applications besides impersonation detection.

10 Acknowledgements

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

A Android UI Background

In this section we provide a concise primer on Android application user interfaces. Android apps with graphical user interfaces are implemented using activity components. A typical app has multiple activities, one for every separate functionality. For example, a messaging app could have activities for reading a message, writing a message, and browsing received messages. Each activity provides a window to which the app can draw its user interface, that usually fills the entire device screen, but it may also be smaller and float on top of other windows. The Android system maintains the activities of recently active apps in a stack. The window surface from the top activity is shown to the user or if the window of the top activity does not cover the entire screen, the user sees also the window beneath it. The user interface within a window is constructed using views, where each view is either a visible user interface element (e.g., a button or an image) or a set of subviews. The views are organized in a tree hierarchy that defines the layout of the user interface.

The recommended way to define user interface layouts in Android is using XML resource files, but Android apps can also construct view hierarchies (UI element trees) programmatically. Furthermore, Android apps can construct their user interfaces using OpenGL surfaces and WebViews. The OpenGL user interfaces are primarily used by games, while WebView is popular with many cross-platform apps. Inside a WebView, the user interface is constructed using common HTML. All of the methods stated above can be implemented in Java, as well as native code.

Every Android application contains a manifest file that defines the application's activities. One activity is defined as the application entry point, but the execution of the application can be started from other activities as well. When an activity is started, the Android framework automatically renders (inflates) the layout files associated with the activity.

B Containment Threshold Verification

To further verify our manual containment threshold evaluation performed in Section 4.2, we created a large ground truth of application pairs known to be trivial repacks of each other. We define a trivial repacks as any pair of applications where both applications have at least 90% identical resource files as well as a code similarity (computed using Androguard) of 90% or higher. For such pairs, we can be confident that the apps are indeed repacks. Over the course of several days, we compiled a list of 200,000 pairs to serve as a ground truth of known repacks. We then ran our system against every pair in this list, querying either app of the pair. If the system did not find the corresponding app, or if their containment is below the threshold value of 45%, we consider this a false negative. We repeated this exercise for different threshold values and compared the results to the expected false negative rates, confirming that the pairs considered in our manual verification are indeed representative for our large data-set.

C Application Dataset

Here we present the statistics of all the applications we used in our experiments.

Marketplace	**apps**
play.google.com (US)	14,323
coolapk.com (CN)	5666
m.163.com (CN)	24,069
1mobile.com (CN)	24,173
mumayi.com (CN)	29,990
anzhi.com (CN)	36,202
slideme.org (US)	19,730
android.d.cn (CN)	4635
Total	**158,449**

Table 1. Application dataset.

References

1. Android malware genome project. `http://www.malgenomeproject.org`.
2. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *International Conference on Automated Software Engineering (ASE)*, 2012.
3. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Network and Distributed System Security (NDSS)*.
4. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

5. T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.
6. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.
7. 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.
8. J. Buchner. `https://pypi.python.org/pypi/ImageHash`.
9. 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 Symposium*, 2015.
10. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Annual symposium on Computational Geometry (CG)*, 2004.
11. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.
12. Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.
13. P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011.
14. M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*.
15. K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Recent advances in intrusion detection*, pages 101–120. Springer, 2009.
16. S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2012.
17. M. Kropp and P. Morales. Automated gui testing on the android platform. In *International Conference on Testing Software and Systems (ICTSS)*, 2010.
18. Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.
19. L. Malisa, K. Kostiainen, and S. Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. Cryptology ePrint Archive, Report 2015/709, 2015. `http://eprint.iacr.org/`.
20. A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Working Conference on Reverse Engineering (WCRE)*, 2003.
21. S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015.

22. B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1), 2014.
23. J. Oberheide and C. Miller. Dissecting the android bouncer. In *SummerCon*, 2012.
24. V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.
25. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.
26. L. Stefanko. Android Banking Trojan, March 2016. `http://www.welivesecurity.com/2016/03/09/android-trojan-targets-online-banking-users/`,.
27. M. Sun, M. Li, and J. Lui. Droideagle: seamless detection of visually similar android apps. In *Conference on Security and Privacy in Wireless and Mobile Networks (Wisec)*, 2015.
28. Symantec. Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna?, April 2015. `http://goo.gl/xh58UN`.
29. Z. Tang, Y. Dai, and X. Zhang. Perceptual hashing for color images using invariant moments. *Appl. Math*, 6(2S):643S–650S, 2012.
30. M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2002.
31. T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*.
32. L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.
33. W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering (FASE)*. 2013.
34. H. Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing, 2013.
35. C. Zauner. *Implementation and benchmarking of perceptual image hash functions*. na, 2010.
36. F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *ACM conference on Security and privacy in wireless and mobile networks (Wisec)*, 2014.
37. Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 411–420. IEEE, 2007.
38. W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.
39. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*.
40. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (S&P)*, May 2012.
41. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.