This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2809727, IEEE Transactions on Mobile Computing

Understanding In-app Ads and Detecting Hidden Attacks through the Mobile App-Web Interface

Rui Shao, Vaibhav Rastogi, *Member, IEEE,* Yan Chen, *Fellow, IEEE,* Xiang Pan, *Member, IEEE,*
Guanyu Guo, Shihong Zou, *Member, IEEE,* Ryan Riley, *Member, IEEE,*

Abstract—Mobile users are increasingly becoming targets of malware infections and scams. In order to curb such attacks it is important to know how these attacks originate. We take a previously unexplored step in this direction. Numerous in-app advertisements work at this interface: when the user taps on the advertisement, she is led to a web page which may further redirect until the user reaches the final destination. Even though the original applications may not be malicious, the Web destinations that the user visits could play an important role in propagating attacks.

We develop a systematic static analysis methodology to find ad libraries embed in applications and dynamic analysis methodology consisting of three components related to triggering web links, detecting malware and scam campaigns, and determining the provenance of such campaigns reaching the user. Our static analysis system identified 242 different ad libraries and dynamic analysis system was deployed for a two-month period and analyzed over 600,000 applications while triggering a total of about 1.5 million links in applications to the Web. We gain a general understanding of attacks through the app-web interface and make several interesting findings including a rogue antivirus scam, free iPad scams, and advertisements propagating SMS trojans.

Index Terms—malware, ad libraries, app-web interface.

1 Introduction

Android is the predominant mobile operating system with about 80% worldwide market share [1]. At the same time, Android also tops among mobile operating system in terms of malware infections [2]. Part of the reason for this is the open nature of the Android ecosystem, which permits users to install applications for unverified sources. This means that users can install applications from third-party app stores that go through no manual review or integrity violation. This leads to easy propagation of malware. In addition, industry researchers are reporting [3] that some scams which traditionally target desktop users, such as ransomware and phishing, are also gaining ground on mobile devices.

In order to curb Android malware and scams, it is important to understand how attackers reach users. While a significant amount of research effort has been spent analyzing the malicious applications themselves, an important, yet unexplored vector of malware propagation is benign, legitimate applications that lead users to websites hosting malicious applications. We call this the *app-web interface*. In some cases this occurs through web links embedded directly in applications, but in other cases the malicious links are visited via the landing pages of advertisements coming from ad networks.

A solution directed towards analyzing and understanding this malware propagation vector will have three components: triggering (or exploring) the application UI and following any reachable web links; detection of malicious content; and collecting provenance information, i.e., how malicious content was reached. There has been some related research in the context of Web to study so-called malvertising or malicious advertising [4], [5]. The context of the problem here is broader and the problem itself requires different solutions to triggering and detection to deal with aspects specific to mobile platforms (such as complicated UI and trojans being the primary kinds of malware).

In order to better analyze and understand attacks through app-web interfaces, we have developed an analysis framework to explore web links reachable from an application and detect any malicious activity. We dynamically analyze applications by exercising their UI automatically and visiting and recording any web links that are triggered. We have used this framework to analyze 600,000 applications, gathering about 1.5 million URLs, which we then further analyzed using established URL blacklists and anti-virus systems to identify malicious websites and applications that are downloadable from such websites. We need to mention that we could not trigger ads or links in about $5/6^{th}$ of the applications. Note that many applications do not have any ad libraries (we can statically check for this) but still have to be run as there may be other kinds of links present. To give an example, for a run of 200K applications in China, we obtained 400K chains with 770K URLs. However, there are only 30K unique applications and 180K unique URLs. The other applications either do not have any ads or links or, in some cases, we may not have been able to trigger those

- *R. Shao and G. Guo are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: {ruishao, guanyuguo}@zju.edu.cn,*
- *V. Rastogi is with the Computer Science Department, University of Wisconsin Madison, Madison, WI, 53706, USA. E-mail: vrastogi@wisc.edu*
- *Y. Chen and X. Pan are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA. E-mail: ychen@northwestern.edu, xiangpan2011@u.northwestern.edu*
- *Shihong Zou is with the Department of CyberSpace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China. E-mail: zoush@bupt.edu.cn*
- *R. Riley is with the Department of Computer Science and Engineering, Qatar University, Doha, Qatar. E-mail: ryan.riley@qu.edu.qa.*

ads or links. Our methodology enables us to explore the Web that is reachable from within mobile applications, something that is not possible for traditional search engines and website blacklist systems such as Google Safebrowsing. We are not aware of any previous work that enables this.

We make the following contributions.

- We have developed a framework for analyzing the app-web interfaces in Android applications. We identify three features for a successful methodology: triggering of the app-web interfaces, detection of malicious content, and provenance to identify the responsible parties. We incorporate appropriate solutions for the above features and have implemented a robust system to automatically analyze app-web interfaces. The system is capable of continuous operation with little human intervention.
- As part of our triggering app-web interfaces, we developed a novel technique to interact with UI widgets whose internals do not appear in the GUI hierarchy. We develop a computer graphics-based algorithm to find clickable elements inside such widgets.
- We deployed our system for a period of two months in two locations, one in North America and another in China. We studied over 600,000 applications from Google Play and four Chinese stores for a period of two months and identified hundreds of malicious files and other scam campaigns. We present a number of interesting findings and case studies in an attempt to characterize the malware and scam landscape that can be found at the app-web interface. As some examples, we have found rogue ad networks propagating rogue applications; scams enticing users by claiming to give away free products propagating through both in-app advertisements and links embedded in applications; and dangerous SMS trojans propagating through well-known ad networks.
- In order to assist with determining the provenance of the identified malicious links, we conducted a systematic study to associate ad networks with ad library packages in existing applications. We apply the *MinHash* [6] and *LSH* [7] techniques to greatly improve the efficiency. The system is also incremental, allowing new apps to be analyzed on demand. Our study reveals 242 ad networks and their associated ad library packages. To the best of our knowledge, this is the largest number of ad libraries identified. We also analyze the popularity of the applications to help us understand the distribution of ad libraries in the markets.

The manuscript extends our conference version [8] in the following important ways: (*a*)We apply the *MinHash* [6] and *LSH* [7] techniques to greatly improve the efficiency of finding ad libraries system. This demonstrates the scalability of the approach, even when applied to a large number of applications. We found 40 new ad libraries in 300,000 applications. The system is also incremental, allowing new apps to be analyzed on demand(Section 3). (*b*)We add the popularity part to help us know about the ad libraries distribution of markets.

In our findings, we have detected both applications embedding links leading to malicious content as well as advertisements that are malicious. We note that the two cases are different in terms of which party is to blame: the application developer, or others like the advertisement networks. Our results indicate that in both cases, the users can be offered better protection on the Android ecosystem by screening out offending applications that embed links leading to malicious content as well as making ad networks more accountable for their ad content.

The rest of this paper is organized as follows. Section 2 presents the necessary background. Our methodology is presented in Section 4 while Section 5 discusses implementation details. Section 6 and 6.6 presents our results and some interesting findings characterizing the studied malware and scam landscape. Related work is presented in Section 7. Finally, we conclude in Section 8.

2 Background

In this section we provide the necessary context in which our system and study fits as well as some details which led to important decisions in our methodology.

2.1 Android Ecosystem

Android is a dominant mobile operating system. The core operating system is developed primarily by Google and is used by many device vendors as the platform for their devices. Apart from system applications, Android also allows running third-party applications, which serve to enrich the functionality of user's devices.

Application stores serve as the primary venue for the users to find and install applications. Google maintains the official Android application store, called Google Play. However, there also exist other application stores. In some countries, such as China, Google services are not as popular and so the unofficial stores serve as the primary method of application distribution. Most devices and vendors allow application installation from unofficial sources, including third-party application stores and web links.

Apart from the discovery mechanisms built into the application stores, users may also discover applications through advertisements in other applications. These advertisements may be served through ad networks or may be directly embedded by the application developers without the involvement of intermediary ad networks. Furthermore, in some cases applications may include direct web links (i.e., not affiliated with any application store).

2.2 Advertising

In-app advertisements are a significant source of revenue for application developers, and as such form a significant portion of app-web interaction on mobile devices. As an ad-supported application runs, it shows advertisements from various ad networks. Advertisements take a variety of forms ranging from banners at top or bottom area of the screen, whole-screen interstitials during switching of activities (roughly equivalent to windows) in the application, and as system notifications.

In the context of mobile advertising, the *advertisers* are parties who wish to advertise their products, the *publishers* are mobile applications (or their developers) that bring advertisements to the users. *Ad networks* or *aggregators* link the publishers to the advertisers, being paid by the latter and paying the former. Ad networks themselves may have complex relationships with each other; Applications with advertisements em bed some code from ad networks. This code provides the glue between the ad network and the publisher. It is responsible for managing and serving advertisements and is called *ad library*. Each ad library may be attributed to an ad network. Clicking on advertisements may lead users to content on Google Play or to web links. This often happens through a chain of several web page redirections. We generally refer to all these URLs in these web page redirections as the *redirection chain* and the final web page as the *landing page.* Ad networks themselves may participate in complex relationships with each other. Certain parties, which may be ad networks themselves, run so-called *ad exchanges* where a given ad space is auctioned among several bidding ad networks so as to maximize profits for the publishers. Ad networks also have *syndication* relationships with each other: an ad network assigned to fill a given ad space may delegate that space to another network. Such delegation can happen multiple times through a chain of ad networks and is visible in the redirection chains.

Applications with advertisements embed some code from ad networks. This code provides the glue between the ad network and the publisher. It is responsible for managing and serving advertisements and is called *ad library*.

2.3 Android Malware

Among the mobile operating systems, Android is particularly troubled by malware. Part of the reason for this is the openness in the ecosystem: applications can be downloaded from the Web and through unofficial application stores. The stores may be checking for malware with varying strictness while for Web links, there may be very little the user can do to know whether the downloaded applications are trusted.

It is also noteworthy that most Android malware comes as trojans, i.e., applications that have a purported useful function as well as a hidden malicious function. Android implements a sandboxed application model, so that the compromise of one application does not directly mean compromise of the whole system. In the context of the Web and browsers, this means that drive-by-download attacks are difficult. Therefore, malware infections on Android happen not through drive-by-download attacks, which are fairly common on some other operating systems, but through trojans.

In our methodology, therefore, we do not attempt to detect drive-by-download attacks but rather scams that may entice users into downloading trojans or applications that charge users exorbitant amount of money.

3 Ad Network Identification

Applications that monetize with advertisements partner with ad networks and embed code called ad libraries from them to display and manage those advertisements. Our goal in this section is to comprehensively identify ad networks that participate in the Android ecosystem. Some simple domain knowledge, such as which ad networks are in the market, may not provide a comprehensive list. We instead resorted to a systematic approach to do this by analyzing ad libraries found in a large number of actual Android applications.

Our approach allows for comprehensive identification of ad libraries with very little manual effort. We begin by analyzing relationships among different entities in the application to identify independent code components, some of which could be ad libraries. We then map these components to robust feature sets derived from Android SDK APIs and then, based on these feature sets, cluster these components. The components can then be manually studied with little effort to identify if they correspond to some ad networks. In the following, we describe our approach in greater detail.

3.1 Component Decoupling

In general, the main application functionality is only loosely coupled with the functionality of ad libraries. The entire logic of fetching and displaying ads is implemented in the ad libraries while the other parts of the application may only occasionally make calls into the ad library code. Intuitively, in the application call graph and def-use graph, we would therefore see a densely connected region corresponding to the ad library which is only loosely connected to the other components of the application. Our goal here is to separate out these loosely connected components.

Specifically, in order to decouple components, we implement the technique described by Zhou et al. [9]. They measure coupling in terms of of characteristics such as field references, method references, and class inheritances across classes. We build a dependency graph among Java classes: two classes are connected by an edge if code in one of them refers to that in the other through field references, method references, and class inheritances. Edges have weights and multiple edges between two vertices are collapsed and replaced by a single edge with the total of the weights of those edges. How closely a class is connected to another class is quantified by the total weight on the edges between the vertices.

Having built such a graph, we iteratively collapse any vertices that are connected by an edge whose weight exceeds a threshold. The final result is a reduced graph whose vertex set is the desired loosely coupled components. Each component contains a group of classes, which are usually succinctly identifiable with a few packages (Java packages are hierarchical namespaces in which class definitions are organized). Such succinct identifiers are useful when performing manual analysis later.

Ideally, all the packages of one ad library will be grouped into one component while the non-related packages will be placed in other components. However, the errors are tolerable and can be manually analyzed.

3.2 Clustering Components

Once we have identified components in applications, we can make clusters of similar components over our entire application set. Ad libraries tend to be used by many

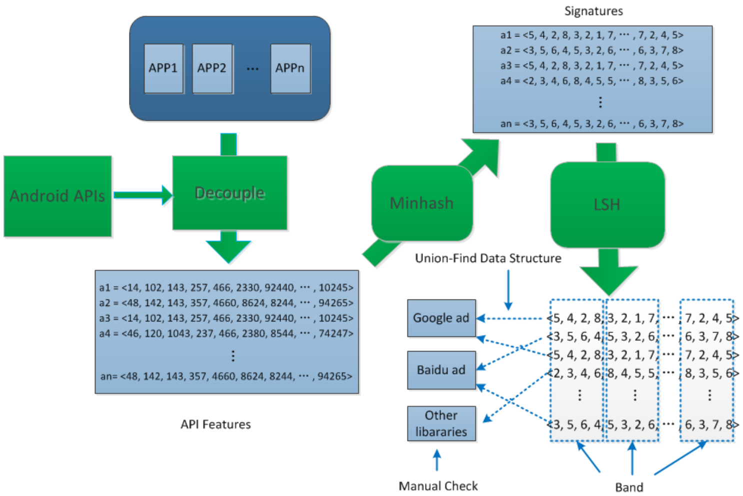

Signatures
a1 = <5, 4, 2, 8, 3, 2, 1, 7, ··· , 7, 2, 4, 5>
a2 = <3, 5, 6, 4, 5, 3, 2, 6, ··· , 6, 3, 7, 8>
a3 = <5, 4, 2, 8, 3, 2, 1, 7, ··· , 7, 2, 4, 5>
a4 = <2, 3, 4, 6, 8, 4, 5, 5, ··· , 8, 3, 5, 6>
⋮
an = <3, 5, 6, 4, 5, 3, 2, 6, ··· , 6, 3, 7, 8>
APP1
APP2
···
APPn
Android APIs
Decouple
Minhash
LSH
Union-Find Data Structure
a1 = <14, 102, 143, 257, 466, 2330, 92440, ··· , 10245>
a2 = <48, 142, 143, 357, 4660, 8624, 8244, ··· , 94265>
a3 = <14, 102, 143, 257, 466, 2330, 92440, ··· , 10245>
a4 = <46, 120, 1043, 237, 466, 2380, 8544, ··· , 74247>
⋮
an= <48, 142, 143, 357, 4660, 8624, 8244, ··· , 94265>
API Features
Google ad
Baidu ad
Other libararies
<5, 4, 2, 8, 3, 2, 1, 7, ··· , 7, 2, 4, 5>
<3, 5, 6, 4, 5, 3, 2, 6, ··· , 6, 3, 7, 8>
<5, 4, 2, 8, 3, 2, 1, 7, ··· , 7, 2, 4, 5>
<2, 3, 4, 6, 8, 4, 5, 5, ··· , 8, 3, 5, 6>
<3, 5, 6, 4, 5, 3, 2, 6, ··· , 6, 3, 7, 8>
Manual Check
Band

Fig. 1. Overview of the structure of ad library identification

applications at once and thus bigger clusters are more likely to correspond to ad libraries that smaller clusters.

Our clustering should be robust against minor differences in code of components as well as renaming of classes and packages. This would, for example, enable us to cluster different ad library versions together. To do this, we first map our components to the Android APIs that the code in these components call. These APIs thus form our feature sets. Note that such features are representative of the functionality of the code: Different pieces of code performing similar functions are likely to call the same Android APIs.

A well-known measure of similarity between two sets is the Jaccard coefficient, which is defined for sets A and B as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Note that $0 \leq J(A,B) \leq 1$ and $J(A,B)$ is higher when A and B are similar.

Pair-wise computation of the Jaccard coefficient for the different components, however, has a runtime that is quadratic in the number of components and linear in the size of the sets. A popular approach to reduce the runtime is to estimate the pairwise Jaccard coefficients for a number of sets through a *locality sensitive hashing (LSH)* [7] technique, such as *MinHash* [6], where each set is represented as a constant-sized signature.

We give a brief overview of MinHash here. Let h be a random hash function mapping elements of sets A and B to integers. Let us define $h_{min}(S) = \min_{x \in S} h(x)$. It can be shown that the probability that $h_{min}(A) = h_{min}(B)$ is the same as $J(A,B)$. Thus, by increasing the number of hash functions used and considering them together, we can obtain an arbitrarily good estimate of the Jaccard coefficient. For each API feature, we use p hash functions to calculate hash values of the set and use the minimum value as an element of the signature vector. We repeat it q times, therefore, each signature has q elements for each API feature.(p = 80 and q = 80 in our system).

Lets give a detail description of LSH. LSH divides a signature into multiple lower-dimensional vectors, called a band. We choose to divide the signature into b bands where each band has r elements. We can compare the same parts of the band between signatures to know whether they are identical. For each band of a signature, we use a hash function to calculate their hash values. We use the same hash function for the same column. If their hash values are the same, we add them to the same bucket. For example, as we can see in Figure 1, the first and the third signature of the first column has the same value [5, 4, 2, 8], indicating their hash values are the same, so they will be pushed into the same bucket. If their hash values are not the same in the first column but the same in other columns, we also push them to the same bucket.

Algorithm 1 gives the pseudocode of LSH. Sig is a vector set of signatures. Each vector has b bands and each band has r elements. $hashValue$ is a r values hash function, and $Union$ is a Union-Find data structure. We initialize a union-find data structure with each component representing one cluster, and then we union the similar component according to the LSH algorithm.

Algorithm 1 LSH algorithm

```
for $j \leftarrow 1$ to $length[Band]$
    do
        for $i \leftarrow 2$ to $length[Sig]$
            do
                $Vaule1 \leftarrow hashValue(Sig[i][j])$
                $Vaule2 \leftarrow hashValue(Sig[i-1][j])$
                if $Value1 = Value2$
                    then $Union(Sig[i], Sig[j])$
```

3.3 Manual analysis

Recall that ad libraries are embedded in multiple applications. Once separated into components by decoupling, the components belonging to the same ad libraries will likely be clustered together. We examine manually whether a cluster represents an ad library and if so, which one. Since ad libraries appear in a number of applications, we examine clusters with a size above a threshold, which we choose to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2809727, IEEE Transactions on Mobile Computing

be 10. This screens numerous clusters that may represent application-specific code. Next, we choose the most common package names in a cluster and check if they belong to an ad library – this can be done by search for those package names on the Web.

3.4 Incremental analysis

Our technique easily supports incremental analysis to identify new ad libraries in newly published applications. To accomplish this, we provide two features. First, we can persist the clusters and features on storage and instantiate our runtime data structures, e.g., union-find, from these clusters. Handling new applications is simply a matter of decoupling components in them, creating clusters for them, and merging them with the previous clusters. Second, to aid manual analysis, we save a list of package names that we had confirmed earlier to be or to not be ad libraries. This saves redundant effort in examining package names.

3.5 Complexity comparison

Suppose that we get N modules and need to cluster them next. In the previous algorithm, we calculate the Jaccard coefficient between modules and cluster the modules based on the the value. The complexity is $O(x^2)$. When applying MinHash and LSH, as described in 3.2. We consider the time of getting hash value is a constant because the input value of hash function is less than the number of Android SDK APIs. So the complexity of getting signatures of API feature is $O(\mathrm{N})$. For the LSH part, the complexity is $O(\mathrm{N})$ because the length of the signature is q = 80 and q = b * r. (b and r refers to the number of bands and the number of elements for each band). We also validated the reduction in time complexity empirically. The previous algorithm took about three days to cluster 10K applications modules, with quadratic increase in complexity. However, it took only about 15 hours to cluster 300K applications modules when using MinHash and LSH. Our new algorithm is obviously more efficient.

4 Studying malvertising

Our methodology for analyzing app-web interfaces will involve the following three conceptual components:

- *Triggering.* This involves interacting with the application to launch web links, which may be statically embedded in the application code or may be dynamically generated (such as those in the case of advertisements).
- *Detection.* This includes the various processes to discriminate between malicious and benign activities that may occur as a result of triggering.
- *Provenance.* This is about understanding the cause or origin of a detected malicious activity, and attributing events to specific entities or parties. Once a malicious activity is detected, this component provides the information required in order to hold the responsible parties accountable.

Different processes and techniques may be plugged-in to these different components almost independently of what goes into the other components.

The rest of this section elaborates on these three components, describing the various processes we incorporate into each of them. An overall schematic depiction of all the involved processes is presented in Figure 2.

4.1 Triggering App-Web interfaces

In order to trigger web links from within the application, we run the applications in a custom dynamic analysis environment. To enable scalability and continuous operation, running applications on real devices is not a feasible option. We deployed our system using multiple AVDs (20 in our test) in parallel for large-scale testing. If we use multiple real phones to run apps, it will increase the costs. Besides, with an application installed on a real phone, it may affect the results of other applications in spite of uninstalling it before installing another application. If we use emulators, we can kill the previous emulator and start a clean emulator to for a new application. Our system can easily support real phones for analyzing apps although we dont choose it. Therefore, each application is run in a virtual machine based on the Android emulator. The applications we are interested in are primarily GUI oriented and therefore we need to navigate through the GUI automatically to trigger app-web interfaces. The rest of this subsection describes the techniques that we leverage from past research in order to accomplish this, as well as some new techniques designed to overcome issues specific to the app-web interface.

4.1.1 Application UI Exploration

Application user interface (UI) exploration is necessary to trigger app-web interfaces. Researchers have come up with a number of systems for effective UI exploration catering to varied applications and incorporating different techniques (Section 7). An effective UI explorer will offer high coverage (of the UI, which may also translates to code coverage) while avoiding redundant exploration. For our work, we used the heuristics and algorithms developed in AppsPlayground [10]. We briefly describe these next.

UI exploration generally involves extracting features (the widget hierarchy) from the displayed UI and iteratively constructing a model or a state machine of the application's UI organization, i.e., how different windows and widgets are connected together. A black-box (or grey-box) technique, such as AppsPlayground, may apply heuristics to identify which windows and widgets are identical to prevent redundant exploration of these elements. Window equivalence is determined by the activity class name (an activity is a code-level artifact in Android that describes one screen or window). Widget equivalence is determined by various features such as any associated text, the position of the widget on the screen, and the position in the UI hierarchy. In order to prevent long, redundant exploration, thresholds are used to prune the model search.

4.1.2 Handling Webviews

While studying advertisements, we faced a significant challenge: most of the in-app advertisements are implemented as customizations of Webviews (these are special widgets that render Web content, i.e., HTML, JavaScript, and CSS). Webviews and some custom widgets are opaque in the UI

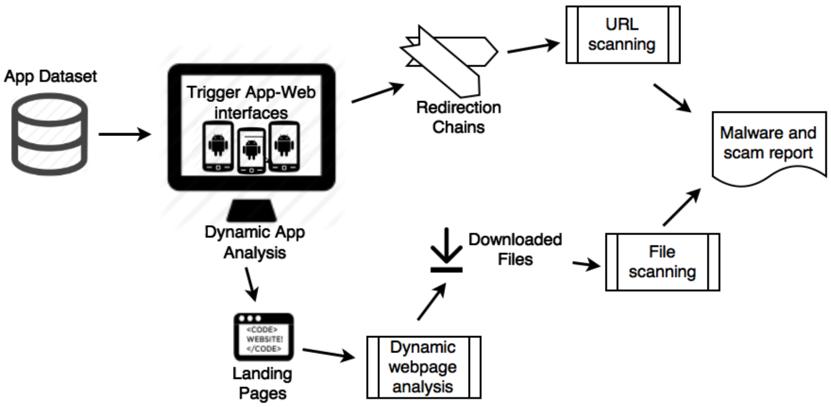


Fig. 2. Overview of measurement methodology

hierarchy obtained from the system, i.e., the UI rendered inside them cannot be observed in the native UI hierarchy and thus interaction with them will be limited. To the best of our knowledge, previous research has not proposed a satisfactory solution to this problem.

Certain open source projects, such as Selendroid [11], may be used to obtain some information about the internals of the Webview. We developed code around Selendroid to interact with Webviews. However, our experience was that it is difficult to use the information provided from Webviews to trigger advertisements. Advertisements often include specific buttons (actually decorated links) that should be clicked to trigger the ads. They may also present other features such as those relating to users' preferences, but which are irrelevant for our purposes. The relevant links cannot easily be distinguished from the irrelevant ones. Often times the click-able link is represented by images instead of text. If we click the irrelevant links, ads may not get triggered, resulting in low click-through rates.

Algorithm 2 Button detection algorithm

```
1. Perform edge detection on the view's image
2. Find contours in the image
3. Ignore the non-convex contours or those with very small area
4. Compute the bounding boxes of all remaining contours
```

In order to overcome this issue of essentially flat Webviews, we apply computer vision techniques in order to detect buttons and widgets as a human would see them. Algorithm 2 presents the detection algorithm.

The first step, edge detection, is the technique of identifying sharp changes in an image. Fundamentally, it works by detecting discontinuities in image brightness. We specifically use the Canny edge detection algorithm, a classical, yet generally well-performing edge detection algorithm. In the second step we compute contours of images, using the computed edges, to obtain object boundaries. Since buttons typically have a convex shape and a large enough area so that a user can easily tap on them, we ignore non-convex contours and those with a small area within a threshold parameter. Numerous contours such as those arising out of text or the non-convex or open contours in embedded images are eliminated in this step. For the remaining contours, we compute the bounding boxes, or the smallest rectangles that would contain those contours. This step is simply to identify a central point where a tap can be made to simulate a button click.

The resulting bounding boxes signify the buttons that would be visible to a human being. We have not performed a thorough evaluation of the accuracy of our technique but the results are good in the cases we have examined. Figure 4 presents some cases related to ads as well as other views. We note that this technique depends only on computer graphics and vision algorithms, is completely black box as it does not even need to extract the UI hierarchy from the system. It can therefore be generally used for any widget whose internals are opaque to the UI hierarchy extraction. This technique also achieves a slightly better click rate on advertisements (measured by the number of redirection chains generated on exploring a given large number of applications) than the previous technique based on Selendroid. In our deployment, we therefore employ this second technique only.

4.2 Detection

As the links are triggered, they may be saved for further analysis and detection of malicious activity such as spreading malware or scam. We would like to capture the links, their redirection chains, and their landing pages. The links, redirection chains, and the content of the landing pages may then be further analyzed using various methods.

4.2.1 Redirection chains

Advertisements redirect from one link to another until they finally arrive at the landing page. As discussed earlier, the redirection may be a result of ad syndication and auction or may even be performed within an ad network itself or by the advertisers themselves. An example redirection chain of length five is shown in Figure 3. Redirection chains may also be observed in non-ad links. Redirection may be performed using several techniques, including HTTP 301/302 status headers, HTML meta tags, and at the JavaScript level. Furthermore, we found that certain ad networks such as Google ads use time-based checks (preventing following the chain too quickly) in order to reduce possibility of click fraud. The result of this is that the links must be launched in real-time to obtain redirection messages. In

```
http://mdsp.avazutracking.net/tracking/redirect.php?bid_id=8425..&ids=BMjgzfjI1..&_m=%07
  publisher_name%06%07ad_size%06320x50%07campaign_id%0625265%07carrier%06%07category%06IAB7%07
  country%06..%07exchange%06axonix%07media%06app%07os%06android&ext=
http://track.trkthatpaper.org/path/lp.php?trvid=10439&trvx=f3ea3ff0&clickid=XVm..&pub_name=
  {publisher_name}&ad_size=320x50&camp_id=25265&carrier={carrier}&iab_category=IAB7&country=..&
  exchange=axonix&media=app&os=android
http://com-00-usa5.com/lps/thrive/android/hp/win/us/congrats_blacksmrt/index.php?isback=1&backid1
  =10451&backid2=90ca7507&sxid=b2f..&tzt=..&devicename=&mycmpid=10439&iphone_o=2199&ipad_o=2198&
  os=android&isp=..&country=US&clk=fln&trkcity=..&clickid=X..Q&pub_name=%7Bpublisher_name%7D&
  ad_size=320x50&camp_id=25265&carrier=%7Bcarrier%7D&iab_category=IAB7&exchange=axonix&media=app
http://track.trkthatpaper.org/path/lp.php?trvid=10608&trvx=2721e17a&clk_ip={clk_ip}&clk_campid=
  {clk_campid}&clk_country={clk_country}&clk_device={clk_device}&clk_scr=480x800&clk_tch=true&
  clk_campname={clk_campname}&clk_tzt=0&clk_code=fln
http://com-00-usa5.com/lps/thrive/android/hp/sweeps/us/iphone-winner/index_ipad.php?isback=1&
  backid1=10451&backid2=90ca7507&sxid=377..&tzt=..&devicename=&mycmpid=10608&os=Android&
  devicemodel=Android+4.2&devicetype=mobile&isp=..
```

Fig. 3. An example redirection chain. Lengthy query parameters and those that are could reveal authors' identity (through location/ISP) have been redacted. This example chain is also useful in understanding the case study presented in Section 6.6.2.

order to ensure that our approach accurately follows the redirection chain regardless of the redirection technique used, we use an instrumented web browser to follow the chain, just a real user would. We implemented a custom browser that runs inside the virtualized execution environment so that the ads are loaded completely realistically inside the browser allowing full capture of the redirection chains. Our browser implementation is based on the Webview provided in Android. With Javascript enabled and a few other options tweaked, it behaves completely like a web browser. We additionally hook onto the relevant parts to log every URL (including redirected ones) that is loaded in it while freely allowing any redirections to occur.

4.2.2 Landing pages

Landing pages, or the final URLs in redirection chains, in Android may contain links that may lead to application downloads. Malicious landing pages may lure the users into downloading trojan applications. We load the landing pages in a browser configured with a realistic user agent and window size corresponding to a mobile device, so that the browser appears to be the Chrome browser on Android. We then collect all links from the landing page and click each to see if any files are downloaded. Simulating clicks on pages loaded in a browser ensures that links are found and clicked properly in the presence of Javascript-based events. The downloaded files are analyzed further as below.

4.2.3 File and URL scanning

The collected URLs and files may be analyzed in various ways for maliciousness. In this paper, rather than developing our own analysis, we used results from URL blacklists and antiviruses from VirusTotal. VirusTotal aggregates results from over 50 blacklists and a similar number of antiviruses. Each URL collected, either the landing page or any other URL involved in the redirection chain, is scanned through URL blacklists provided by VirusTotal. This includes blacklists such as Google Safebrowsing, Websense Threatseeker, PhishTank, and others. Files that are collected as a result of downloads from the landing pages are scanned through the antiviruses provided on VirusTotal. Antivirus systems and blacklists are known to have false positives. In order to minimize the impact of this, we use corroboration to reduce the false positive rate: we say a URL or a file is malicious only if it is flagged by at least three different blacklists or antiviruses.

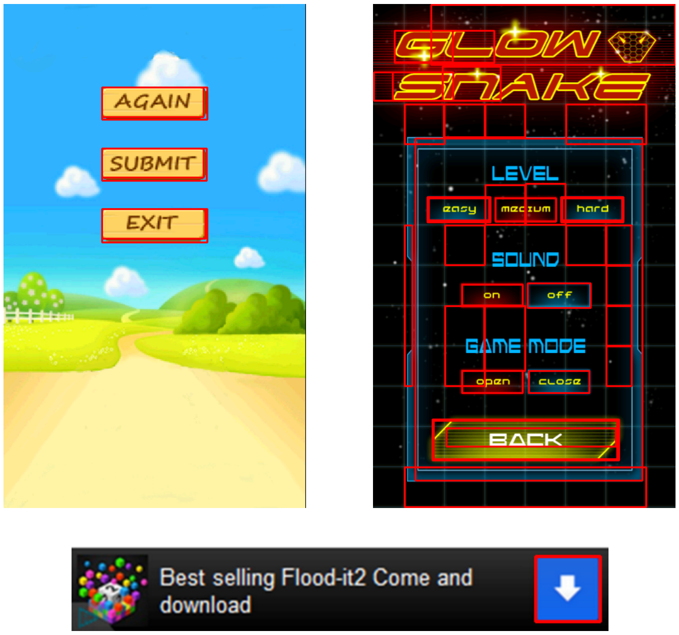


Fig. 4. Examples of detecting buttons with bounding boxes. The bounding boxes are depicted as red rectangles. The top two figures contain the whole screen while the bottom figure is just an ad. Note the detection of buttons.

4.3 Provenance

Once a malicious event is detected, it is necessary to make the right attributions to the parties involved so that these parties can be held responsible and proper action may be taken. In our system, we use two aspects as part of provenance.

- *Redirection chain.* The redirection chain, which is already captured as part of the detection component. The redirection chain can be used to identify how the final landing page was reached: if the landing page contains something malicious, the parties owning the URLs leading up to the landing URL can be identified.

- *Code-level elements.* The application itself may include code from multiple parties such as the primary application developer as well as ad libraries from a variety of ad networks. In order to launch one application from another, Android uses what are called intents. URLs may be opened by applications in the system's web browser by submitting intents to the system with specific parameters. We modify the system to log specific intents that are indicative of URL launches together with which part of the code (the Java class within which the launching code lies) that submitted the intent. This allows us to determine which code with an application launched the malicious URL.

It is important to identify the owners of the code classes captured as part of provenance: do they belong to the application developer or an ad library, and if they belong to an ad library, which one is it? In order to assist us in doing this, we therefore perform the one-time task of identifying prevalent ad libraries and their associated ad networks.

5 Implementation

We implemented most of our system in Python. For UI exploration, we make use of the source code of the AppsPlayground tool [12]. However, the existing version of the tool is unable to run on current versions of Android, and we therefore reimplemented the system to work on current Android versions with the same heuristics as are described in the AppsPlayground paper. Furthermore, instead of using HiearchyViewer for getting the current UI hierarchy of the application, we used UIAutomator, which is based on the accessibility service of Android. This had a significant and positive effect on the speed of execution. The graphics algorithms used for button detection were provided by the OpenCV library and appropriate thresholds were chosen after repeated testing.

To improve speed of dynamic analysis, we take advantage of KVM-accelerated virtualization. To use this, we use Android images that can run on the x86 architecture. About 70% Android applications have no native code and so can run without problem on such targets. Other applications contain ARM native code and cannot run on x86 architecture without proprietary library support. We therefore excluded applications containing native code. Despite this we still believe the study results are generally representative.

For post-trigger analysis, our entire framework is managed through Celery [13], which provides job management with the ability to deploy in a distributed setting. In our implementation the app UI exploration as well as the recording of redirection chains with a real browser happens in tandem. Once this stage is completed, any recorded redirection chains are queued through a REST API into the Celery-managed queue together with information about the application and part of the code that was responsible for the triggering of the intent that led to the redirection chain. Tasks are pulled from the queue to perform further analysis on the landing pages and scan the files and URLs with VirusTotal as described above. The whole system has proper retry and timeout mechanisms in place and could run for multiple months without significant need of human attention.

6 Results

6.1 Application Collection

Our application dataset consists of 492,534 applications from Google Play and 422,505 applications from four Chinese Android application stores: 91, Anzhi, AppChina, and Mumayi. Google Play has a proprietary API for searching and downloading applications from the store and it further requires Google account credentials to do these tasks. We used PlayDrone, which is an open source project to crawl Google Play [14]. Google implements rate limiting based on Google accounts and IP addresses and bans accounts and IP addresses if there are too many requests in a given period of time. PlayDrone mitigates this problem by seamlessly allowing the use of multiple Google accounts and deploying the crawler over multiple machines in a distributed manner. We used the multiple Google accounts feature but simplified the system by using a single machine and setting multiple IP addresses for that machine. In our deployment, every new connection to Google's servers randomly chooses from among twenty source IP addresses. We used PlayDrone to download over half of the free applications on Google Play at the time of download. We met resource limitations in our PlayDrone setup that prevented us from being able to download rest of the applications. Nonetheless, given the large percentage of applications downloaded, we are confident these applications are representative of all application categories. Based on manual sampling from these applications, this set of applications also not biased on application popularity, ratings, downloads, permissions, and other such common metrics.

To crawl applications from Chinese application stores, we used our own in-house tool. These third-party stores have a much simpler API than Google Play and typically have a public http/https URL associated with each application. While there can be sophisticated ways to search for each application, the technique we employed was based on the observation that applications in all these stores have identifiers in a small integer range. Requesting URLs constructed for each possible identifier sufficed to completely scrap these applications stores. After removing applications that were redundant among these stores, the total number amounts to 422,505. About 30% applications have native code and due to implementation reasons mentioned in Section 5 cannot be tested on our system. Our entire usable application dataset therefore consists of a little over 600,000 applications.

6.2 Deployment

We deployed our system to gather results over a period of about two months from mid-April 2015 to June 2015 in two locations, one in a North American university and the other in a Chinese university. The deployment ran continuously with little manual intervention, and restarts were necessary only when we needed to update the system for fixing bugs or adding features. To have a realistic setting, the North American university location ran applications from Google Play while the Chinese university location ran applications

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2809727, IEEE Transactions on Mobile Computing

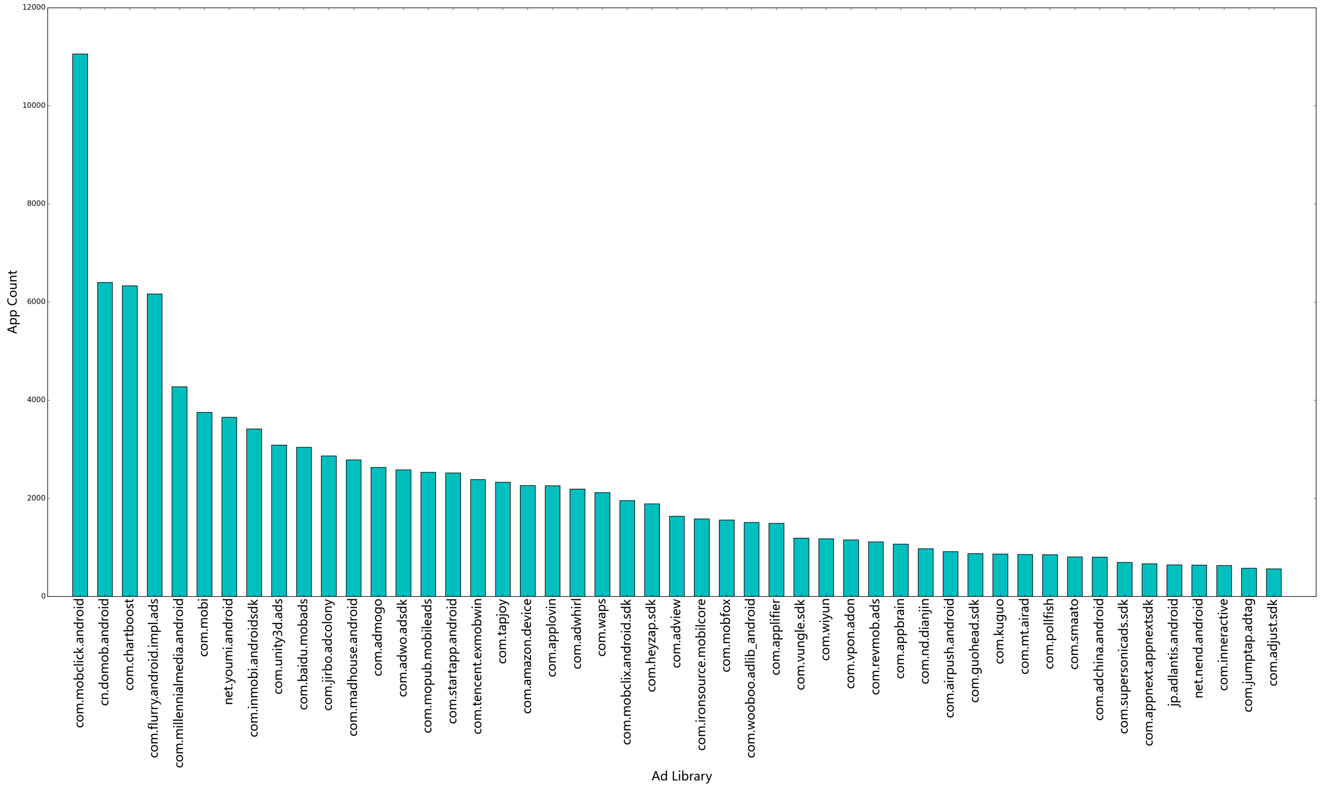


Fig. 5. Top 50 popular ad library from 150,000 applications.

from Chinese application stores. The location where the apps are run is important because much of advertising, which forms bulk of the app-web interaction we are studying, is targeted based on location. The advertisements that are seen in one location may not be shown in another location.

6.3 Popularity

We studied how different ad networks share the market based on a variety of metrics. We quantify ad network usage with the number of applications.

We crawled 150,000 applications in July 2016 including 50,000 from Google Play and 100,000 from the Baidu markets. We identify ad libraries by package names. We decompiled applications to get all Java package names and matching them with ad libraries we found. Figure 5 presents the number of applications using an ad network. The ad networks on the x-axis are ranked (sorted) by the number of applications that use them. As the total number of Google ad networks is very high, it is not feasible to show on the figure. Google ads ranks at the top, being used in 41% applications. Domob and Chartboost come next but are an order of magnitude behind Google ads.

6.4 Detection of ad libs

Using the approach from Section 3, we were able to identify 242 ad networks in 300,000 applications which were downloaded between April 2015 and July 2016. The analysis required about 15 hours spending on compute time. This demonstrates the scalability of the approach, even when applied to a large number of applications.

Some ad networks have ad libraries with several package names. For example, `com.vpon.adon` and `com.vpadn` belong to the same network. We combine such instances together to be represented as ad network for later measurements. More notably, Google's Admob and DoubleClick platforms are both represented as Google ads.

Our approach to use package names to identify ad libraries is contingent upon the assumption that ad library packages are not obfuscated. This is true for most cases that we know of: the top-level packages work quite well to identify most ad libraries. However, Airpush is one known ad network that obfuscates its ad libraries such that they are no longer identifiable with package names [15]. While applying our second approach, which is immune to lexicographic obfuscations, we also detected obfuscated Airpush packages, all ending up in a few clusters. The clusters have the non-obfuscated package com.airpush.android as well as obfuscated ones like com.cRDpXgdA.kHmZYqsQ70374 and com.enVVWAar.CJxTGNEL99769.

6.5 Overall Findings

Overall, we recorded a total of slightly over 1 million launches of app-to-web links in the North American location. In the Chinese location, this number was 415,000. Note that this is not a direct correspondence with the applications: some applications may result in more than one launch while others may not result in any. In North America, we detected a total of 948 malicious URLs coming from 64 unique domains. For the Chinese deployment we detected 1,475 malicious URLs that came from 139 unique domains. We also downloaded several thousands of files of which many were

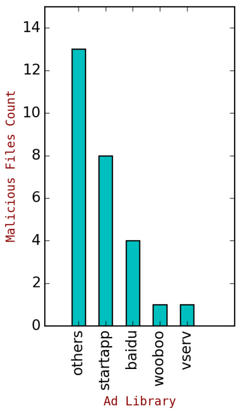


Fig. 6. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in North American deployment. Libraries not resulting in malware downloads are not shown. Tapcontext malware numbers are not shown here as they are too high.

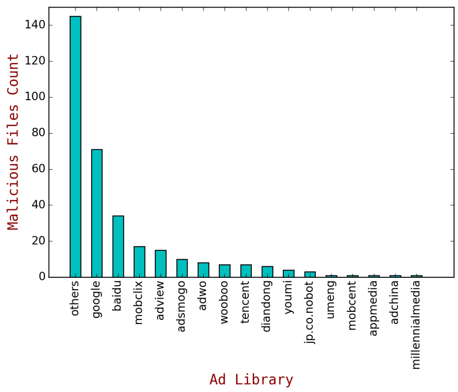


Fig. 7. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in Chinese deployment. Tapcontext and libraries not resulting in malware downloads are not shown.

simple text files or docx files. As for the number of Android applications, the North American deployment collected 468 unique applications (from the Web, outside Google Play) of which 271 were found to be malicious. A large chunk (244) of these malicious applications comes from the antivirus scam reported in Section 6.6.1. Excluding this anomalous number of 244, we find that one in six applications downloaded from the Web (outside Google Play) are malicious. The file numbers above do not include the applications hosted on Google Play. We accounted for such applications separately: there were 433,000 landing Google Play landing URLs, i.e., http URLs with play.google.com domain or URLs with market scheme (beginning with "market://"). These Google Play landing URLs led to a little over 19,000 applications on Google Play. About 5% of these labels are labeled as malicious (based on our criterion of being flagged by at least 3 antiviruses) on VirusTotal. Based on our manual check of the antivirus labels, however, all of these appear to be adware. On the Chinese deployment side, we collected 1,097 unique files of which 435 are malicious. 102 of these files are from the antivirus scam of Section 6.6.1.

Figures 6 and 7 present the distribution of malware downloads through various ad libraries in the North American deployment and in the Chinese deployment respectively. The "others" bar presents the downloads through web links not embedded in advertisements. Both the higher diversity and higher number of malicious downloads in the Chinese deployment are noteworthy. This is likely because the North American Android ecosystem is centered around Google Play and application downloads outside it are rare. However, the Chinese ecosystem depends much more on the Web and third-party Android application stores.

6.6 Case Studies

In this section we describe some interesting cases of scams and malicious applications.

6.6.1 Antivirus Scam

We discuss here an antivirus scam campaign. We found the antivirus, Armor for Android, to be heavily campaigned for through multiple applications in both the North American and the Chinese experiments. In our traces, the entire campaign is running off an ad network known as Tapcontext. In fact, based on our observation lasting a few months, the entire ad inventory of this ad network appears to be related to Armor for Android only.

Applications show the antivirus advertisements as any normal advertisement. In addition, they also sometimes automatically begin scanning for malware on the device (Figure 8 (a)). Our investigations on the Web seemed to clarify this: an apparent Tapcontext representative admits that the ad library has a tie up with an antivirus company that conducts a real scan of the device (perhaps by gathering application checksums and getting information about them from their server) [16]. The scan does show real results but labels minor adware also as threats while still not revealing additional information to the user what threats were found unless a purchase is made.

The next aspect to the scam-like operation is that when the user clicks on an advertisement to download the application, the ad library launches a web page that looks very similar in appearance to a native Android dialog box prompting the user to download and install the antivirus application through the "Download & Scan FREE Now" button. Upon clicking this button, a file by the name of "Scan-For-Viruses-Now.apk" is downloaded. We note that Tapcontext embeds a unique identifier to each click so that the URL of the web page is different every time while the appearance is the same. However, all the URLs come from two domains only: www.fastermobile.org and www.fastermobiles.com. Furthermore, each downloaded Scan-For-Viruses-Now.apk file is the same application (has the same functionality) but is slightly different so that their MD5 and SHA digests never match.

The antivirus application is considered a scam by several antiviruses and some Internet outlets [17] and is variously called as FakeApp, Fakealert, Fakepay, and FakeDoc by antiviruses in their malware labellings. The application charges a hefty subscription fee of 0.99 GBP a day. While the application is also hosted on Google Play (discussing whether

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2809727, IEEE Transactions on Mobile Computing

this is compliant with Google Play's policies is beyond the scope of this paper), the advertisements we saw directed users to download applications from outside Google Play.

Our detection of this campaign was through the "Scan-For-Viruses-Now.apk" files that were detected by antiviruses on VirusTotal. Manual analysis after these detections led us to also discover how the web page with the appearance of an Android dialog box was designed to phish users. We note that we had already detected this scam campaign and identified this phishing behavior at least twenty days prior to Google Safebrowsing and a few other URL blacklists on VirusTotal incorporating www.fastermobile.org URLs as phishing URLs.

The above highlights the importance of running such frameworks on a continuous basis. It is likely that the phishing web pages we detected are not discoverable directly through the Web and hence inaccessible to either search engines or URL safety evaluation infrastructures like those of Google Safebrowsing. By exploring the Web that is reachable from mobile applications, the doors for further analysis are opened and it becomes easier to identify and blacklist phishing websites leading to previously known malware and thus protecting the users.

This case study also offers a good example of how frameworks such as ours can be used to understand and expose scamming ad networks such as Tapcontext. The Tapcontext ad network is being used by more than 1,800 applications in our dataset. Application developers incorporate ad networks for making money; however, such scam networks jeopardize the applications' reputation and are likely to do more harm than good to the developers' revenue. Furthermore, such evidence may also be used by application markets and law enforcement groups to hold ad networks more accountable for the content they present.

6.6.2 Free iPad Scams

In our experiments, we encountered several instances of win-free-iPhone or win-free-iPad advertisements. In our traces, these advertisements had a few landing pages with domains such as com-00-usa5.com and 1.cdna.com, possibly from unrelated parties (based on Whois records). These landing pages present the user in flashy language that they have been lucky, an iPhone (or some other electronic) is theirs if they go to the next step. The example figures are shown [8]. all the users seeing the particular page are "lucky" and "randomly selected to qualify for the special offer". The tricked users upon continuing are lead to a page asking for some questions. This same page may itself come from different URLs such as http://www.electronicpromotion.com/Flow.aspx and http://www.promotionalsurveys.com/Flow.aspx. The page collects the users' personal information such as name, email address, physical address, and phone number and then leads to a website called http://www.amarktflow.com/. The user ends up answering lengthy surveys, confirming the personal information already provided, and then prompted to install an app or a browser toolbar.

None of the above websites themselves are flagged by URL blacklists on VirusTotal. WOT, a crowd-sourced reputation system for websites, however presents a "very poor" reputation for http://www.amarktflow.com/ and considers it a possible scam [18]. The users are simply enticed to give away their personal information, which could be sold or abused, and it is not clear if even a single iPhone or iPad is distributed out to any of the users. Similar scams have been covered in the past in other contexts. Sophos reported a free iPad scam being run through a Facebook application [19]. Similar scams propagating through spam email and SMS messages and over the Web have been covered and discussed elsewhere [20], [21].

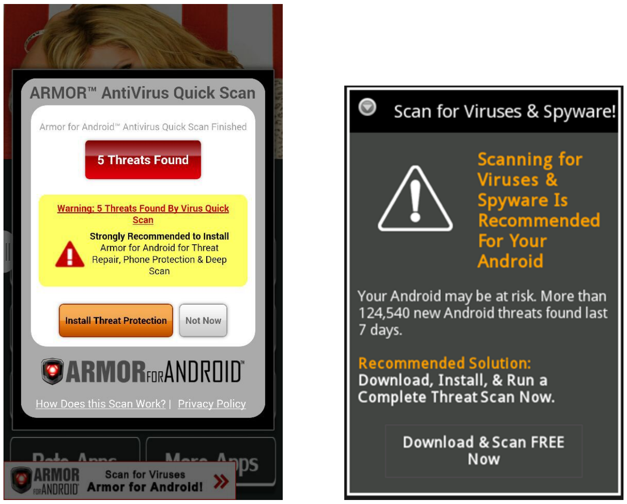
We next bring the reader's attention to how this scam shows up in mobile advertisements. The URL blacklists on VirusTotal flagged some of the intermediate redirection URLs as malicious or phishing websites. The concerned domains here include avazutracking.com and track.trkthatpaper.org. Based on our results, all URLs relating to these domains are not actually bad. These domains appear to be parts of some advertisement networks and exchanges and do show non-malicious content also. Likewise, the com-00-usa5.com mentioned earlier also presents non-malicious advertisements.

The developers are actually unaware that they are using ad services that may show scam content. In our experiments, all the free iPhone and iPad scams appear from two ad libraries: Mobclix and Tapfortap. These libraries retrieve ad content from so-called ad exchanges where multiple networks participate and bid to show advertisements in the given ad space. The bidding ad networks may further have syndication relationships with other ad networks and may allow those networks to show ads on their behalf. In many of these cases of free iPad scams, we believe that Mobclix leverages Axonix, which is another ad exchange. Consider the example redirection chain shown in Figure 3. In between it redirects through multiple domains belonging to ad exchanges and networks with the URLs passing information to those following them through query parameters. Because of this complicated infrastructure of multiple networks involved, it becomes difficult for the developers, ad libraries like MobClix and Tapfortap, and perhaps even the ad networks on top to ensure the quality of the content presented.

Our system is again useful here. If deployed by a responsible party, such as Google or a government agency, which can hold the content publishers accountable, the collected traces can be of invaluable help in getting to the offending parties and gathering evidence against them. In this way, it may be possible to limit the scam content shown to the users.

6.6.3 Scams Through Direct Links

We also encountered scams the result of which is very similar to the free iPad scams described in the previous section. However, how they originate is different. Rather than an advertisement embedded in the application leading to the scam page, in these cases, a web link statically embedded in the application leads to the scam page. The web link appears to link to a benign website not related with advertisements or scam; however, it contains code that loads an advertisement, which then redirects through a series of URLs to a scam landing page. An example is shown in Figure 9. When the user taps on the button labeled "Fiestas de hoy" (Parties Today), a web page opens in the browser and redirects to the scam webpage. As an aside, note the scam page actually shows the user's city (hidden here for authors' anonymity), derived from the client's IP address, perhaps to engender



ARMOR™ AntiVirus Quick Scan

Armor for Android™ Antivirus Quick Scan Finished

5 Threats Found

Warning: 5 Threats Found By Virus Quick Scan

Strongly Recommended to Install Armor for Android for Threat Repair, Phone Protection & Deep Scan

(a)

Scan for Viruses & Spyware!

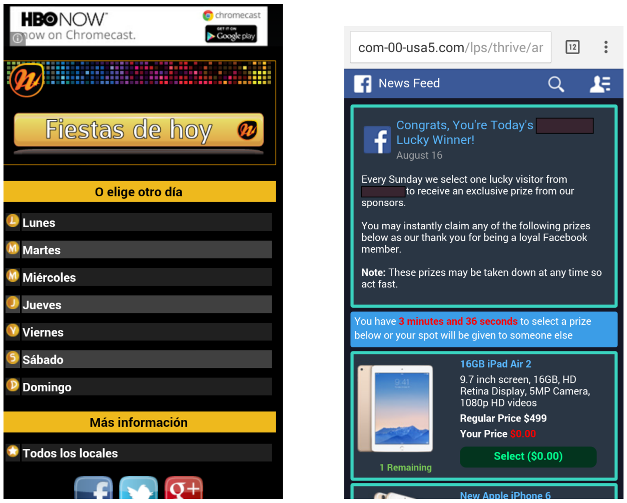
Scanning for Viruses & Spyware Is Recommended For Your Android

Your Android may be at risk. More than 124,540 new Android threats found last 7 days.

Recommended Solution: Download, Install, & Run a Complete Threat Scan Now.

(b)

Fig. 8. Armor for Android antivirus scam. (a) Application conducting gratuitous virus scan; (b) A web page imitating Android dialog box asking user to install the antivirus.



(a)

com-00-usa5.com/lps/thrive/ar

Congrats, You're Today's [redacted] Lucky Winner!
August 16

Every Sunday we select one lucky visitor from [redacted] to receive an exclusive prize from our sponsors.

You may instantly claim any of the following prizes below as our thank you for being a loyal Facebook member.

Note: These prizes may be taken down at any time so act fast.

You have 3 minutes and 36 seconds to select a prize below or your spot will be given to someone else

16GB iPad Air 2
9.7 inch screen, 16GB, HD Retina Display, 5MP Camera, 1080p HD videos
Regular Price $499
Your Price $0.00

(b)

Fig. 9. Another free iPad scam. The scam originates not through an ad in the app but through a link statically embedded in the application (in this case, "Fiestas de hoy"). Upon clicking this link an ordinary URL is launched and as the web page loads, it is redirected to web-based ad providers that show the scam. Note also the presence of the Facebook icon on the web page even though there is no association between Facebook and this website.

confidence in the user. More importantly, it also shows the Facebook logo even though it is not affiliated to Facebook, bringing the scam at the brink of phishing as well.

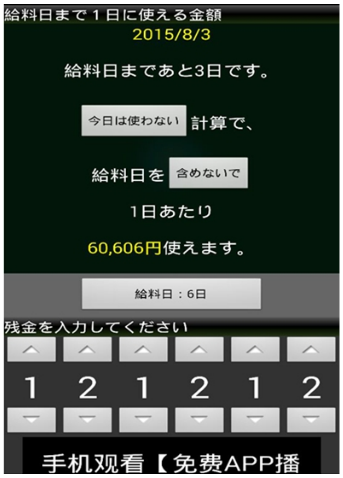
We found a number of applications having such behavior of leading to scams through links embedded in them. The applications we found do not exist on Google Play anymore (although Google's VerifyApps service does not label them as malicious, so removal due to being malicious is unlikely). Our detection of such scam was based on certain URLs whose domains (e.g., zb1.zeroredirect1.com) are nearly always flagged by VirusTotal blacklists. In our automatic attribution of the attack, we found that this scam is not attributed to any of the ad libraries that we detected in Section 3. Looking manually, some of the application's own classes were involved, and it was static links embedded in the app that led to scam pages.

We are not sure if the developers themselves are aware that these applications are participating in propagating scams. It is possible that the developers simply embedded some links and host advertisements on those web pages without knowing that advertisements could lead to scam. On the other hand, some of these applications always seem to lead to scammy advertisements (during the time we tried them); developers may thus knowingly be participating in such scams. The link in the application discussed here being name "Fiestas de hoy" or "Parties today" seems to also signify this.

6.6.4 Fake Movie Player Malware

Our deployment in China also detected several instances of advertisements on Baidu and Nobot ad networks. These advertisements tell the user that they can play videos for free. An example screenshot is shown in Figure 10.

Advertisements like the one at the bottom of the screenshot lead the user to either directly download a video player application, or take to a web page containing pornographic images and prompt the user to download a video player from there. Our system was able to trigger the ads and download the video player applications. These applications are however malicious and, more specifically, SMS trojans, i.e., they send SMS messages to premium numbers without users' consent. On VirusTotal nearly 30 antiviruses detect these applications under various names such as SMSSend and SMSPay. Based on their permissions, some of these applications can also, apart from sending SMS messages, make calls without user confirmation, read and write SMS messages, and monitor applications running on the system.



給料日まで１日に使える金額
2015/8/3
給料日まであと3日です。
今日は使わない 計算で、
給料日を 含めないで
1日あたり
60,606円使えます。
給料日：6日
残金を入力してください
1 2 1 2 1 2
手机观看【免费APP播

Fig. 10. A screenshot with an ad from Nobot at the bottom. The ad says in Chinese that it is free to play video using your mobile phone. It leads to download a video player. The purported video player is actually an SMS trojan.

The number of instances of such advertisements we found was not small either. Our system had triggered 30 advertisements on the Baidu ad network and 3 on the Nobot network. We also note that many of these advertisements do not have any redirection chains: the ad just directly leads to the apk or the landing page. Therefore, we believe it should have been easy for them to spot the malware and block the advertisements. In some cases, there would be a one-level redirection only, going through a site such as http://csu.ssooying.com/QnqQvy. This site is now blacklisted by four URL blacklists, including Google Safebrowsing, on VirusTotal. However, it was not detected by those blacklists at the time these advertisements were seen by our system.

We are pursuing this case, together with the provenance collected wit our system, with CNCERT, an non-profit, non-government organization in China that handles cybersecurity emergency response and to which ad networks are answerable for their content. As an aside, when we manually studied these two ad networks, we were able to see a pharmaceutical campaign that sell alternative therapy drugs for sexual fitness. Based on the content, the campaigns' claims seem dubious so that they could very well be classified as another scam. Even though VirusTotal URL blacklists do not flag the campaign's website, other vendors such as Qihoo 360 flag it as fake and trick website.

7 Related Work

7.1 Advertisement Security and Privacy

Mobile advertisements have been studied in the past from multiple security and privacy perspectives such as ad fraud and security and privacy implications of using ad-supported applications. Liu et al. [22] study a king of ad fraud in which the developer places ads and the main application widgets in such a way that it becomes easy for the user to mistakenly click on ads. Crussell et al. [23] study ad fraud in mobile applications from a network perspective. They identify repackaged applications with the purpose to direct ad revenue away from the original developers and to the persons who repackaged the applications and study the prevalence and implications of this kind of ad fraud. Our main concern in this paper is not ad fraud but the propagation of malicious content through advertisements and web links embedded in applications.

Several researchers have also studied privacy leakages through ad libraries. TaintDroid [24] and some follow-up works [10], [25] all present results in which a large majority of privacy leakages happen through ad libraries included in the applications. While the previous list of works uses dynamic analysis, researchers have also used static analysis to identify privacy leaks in applications, and through ad libraries in particular [26], [27]. Privacy leakages in ad libraries are not in the scope of this paper. However, we do study scams that extract personal information of the users, even with their consent. Grace et al. [28] perform static analysis of ad libraries to discover a number of implications such as private data leakage and execution of untrusted advertisement code in applications. Industry researchers also detected vulnerabilities in ad libraries that can provide escalated privileges to the advertisement code that these libraries execute [29]. AdSplit [30] discusses that ad libraries should be separated from the main application, running in a different sandbox, so that they can have different permissions from the applications, and vulnerabilities and privacy leakages in them do not affect the main application. Quire [31] also proposed techniques that can achieve a similar effect. The goal of this paper is not to identify vulnerabilities due to the inclusion of ad libraries or to fix such problems. The web links or advertisements embedded in applications may themselves not be malicious but their end result is.

A more related aspect of advertising security research is the so-called web malvertising. An important part of our study is malicious advertising in mobile applications. The analogous problem of malicious advertising on the Web, dubbed as malvertising, has been studied in the past. Li et al. [5] use a systematic methodology to crawl websites and load ad content in them. They then analyze the redirection chains and landing pages for malicious activity. Zarras et al. [4] have also studied web malvertising. Our work is different from these works in several aspects. First, our focus is on mobile applications; a similar study on mobile apps has not been done earlier. Moreover, we broadly study all app-web interaction and not just advertisements. Second, a study on mobile applications needs an additional triggering component in the methodology. Work on web malvertising has a different set of challenges. In our domain, we need to discover links by driving the application UI and then click/trigger the links and then follow the redirection chains. Such challenges of discovering the links are not present in previous work. Triggering increases the complexity of the methodology and we have also made an important contribution to enhance it. Finally, the malware propagation vectors through web malvertising are different from what we see on mobile. Drive-by-downloads are virtually non-existent on mobile platforms such as Android due to sandboxing at the process level. Similarly link hijacking, i.e., advertisement or other malicious code embedded in a web page automatically redirecting users to a page they did not intend without any user interaction, is also not possible on mobile apps. Rather the main propagation vector for malware is trojans. Collecting trojans again complicates our methodology as we need to automatically download content from the landing pages.

7.2 Malware Analysis and Detection

Both the industry and the academia are interested in analyzing potentially malicious or malicious applications to understand their behavior. We discuss here works related to mobile platforms only. Google has a service called Bouncer in place to analyze any applications that get uploaded to Google Play for malicious activity [32]. More recently, Google also introduced the VerifyApps service that collects all the applications from the Web, including those not from Google Play, and curates analysis results on those applications. The details of analysis are not public but it is likely to be a mix of both static and dynamic analysis. The results are used to warn the users whenever they install an application of which the VerifyApps is suspicious [33].

Mobile Sandbox [34] and Andrubis [35] are some of the dynamic analysis sandboxes proposed by the academia. They incorporate several different analyses and produce a report for the analyzed application, such as the permissions, the servers contacted while running, and so on. We are not aware of any analysis system that incorporates the kind of analysis we do: understanding the app-web interfaces and following the web links from applications and analyzing if they host any malicious content. If such analysis is supported by the industry or the government, it will be very helpful in curbing down instances of malicious content reachable from mobile applications. Moreover, by using their results, it may be possible for us as well to enhance our detection.

Another avenue of related work is honeypots. Honeypots interact with attackers allowing them to exploit the honeypots. This way, valuable information, such as malicious servers and websites as well as previously unknown vulnerabilities, can be identified. HoneyMonkey [36] is an active honeypot, i.e., it actively crawls and seeks out websites to connect. It analyzes the differences in the system state before and after visiting to determine if it was exploited. Such systems also need to perform triggering and detection; however triggering in case of mobile UI is more complicated. Moreover, our detection also does not seek to identify exploits but to recognize scams and download trojans.

Researchers have also proposed several techniques to perform Android malware detection. Zhou et al. [37] analyzed mobile applications from Play and third-party application stores and detected several instances of malware. Grace et al. [38] perform static analysis on Android applications to systematically detect malware. Arp et al. [39] introduce a machine-learning based system to detect and classify Android malware of previously known families. Zhang et al. [40] propose a dynamic analysis based on permission use to detect malicious applications. Feng et al. [41] and Zhang et al. [42] propose semantics-aware static analyses of applications so as to defeat malware obfuscation attacks such as

8 Conclusion

In order to curb malware and scam attacks on mobile platforms it is important to understand how they reach the user. In this paper, we found 242 ad libraries and explored the app-web interface, wherein a user may go from an application to a Web destination via advertisements or Web links embedded in the application. We used our implemented system for a period of two months to study over 600,000 applications in two continents and identified several malware and scam campaigns propagating through both advertisements and web links in applications. With the provenance gathered, it was possible to identify the responsible parties (such as ad networks and application developers). Our study shows that that should such as system be deployed, the users can be offered better protection on the Android ecosystem by screening out offending applications that embed links leading to malicious content as well as by making ad networks more accountable for their ad content. A regulatory authority like CNCERT(National Internet Emergency Center) could use our tool to understand the prevailing trends in mobile malvertising and hold the ad networks accountable. Similar techniques could also be used by the ad networks themselves to find malvertising in their own networks (note that this is a non-trivial issue due to multiple ad networks involved in serving a single ad).

Acknowledgments

We thank our reviewers for their valuable comments. This paper was made possible by the National Natural Science Foundation of China under Grant No.61472209, by the U.S. National Science Foundation under Grant CNS-1408790. The statements made herein are solely the responsibility of the authors.

References

[1] "Smartphone os market share, q1 2015," http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] "Malware infected as many android devices as windows laptops in 2014," http://bgr.com/2015/02/17/android-vs-windows-malware-infection/.

[3] "Android phones hit by 'ransomware'," http://bits.blogs.nytimes.com/2014/08/22/android-phones-hit-by-ransomware/?_r=0.

[4] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 373–380.

[5] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, "Knowing your enemy: understanding and detecting malicious web advertising," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*. ACM, 2012, pp. 674–686.

[6] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of Sequences 1997. Proceedings*, Jun 1997, pp. 21–29.

[7] J. Buhler, "Efficient large-scale sequence comparison by locality-sensitive hashing," vol. 17, no. 5, pp. 419–428, 2001.

[8] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces," 2016.

[9] 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*. ACM, 2013, pp. 185–196.

[10] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proceedings of ACM CODASPY*, 2013.

[11] "Selendroid: Selenium for android," http://selendroid.io/.

[12] 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*. ACM, 2013, pp. 209–220.

[13] "Celery: Distributed task queue," http://www.celeryproject.org/.

[14] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM international conference on Measurement and modeling of computer systems*. ACM, 2014, pp. 221–233.

[15] Symantec, "Airpush begins obfuscating ad modules," November 2012, http://www.symantec.com/connect/blogs/airpush-begins-obfuscating-ad-modules.

[16] http://forums.makingmoneywithandroid.com/advertising-networks/1868-tapcontext-shit-breaking-policy-making-loosing-acti html#post12949.

[17] http://www.androidauthority.com/armor-for-android-342192/.

[18] "Reputation of amarktflow.com," https://www.mywot.com/en/scorecard/amarktflow.com.

[19] "Free iPad mini scam spreads via facebook rogue application," https://nakedsecurity.sophos.com/2012/10/31/free-ipad-mini-facebook/.

[20] "Apple iPad scam," http://blog.spamfighter.com/software/apple-ipad-scam.html.

[21] "How to spot a 'free iPhone or iPad' scam: Why 'free iPhone' and 'free iPad' stories are always bogus, and how to avoid getting ripped off," http://www.macworld.co.uk/feature/iphone/free-iphone-ipad-scam-fake-auction-site-facebook-3608522/.

[22] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: detecting and characterizing ad fraud in mobile apps," in *Proc. of NSDI*, 2014.

[23] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 123–134.

[24] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010.

[25] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of ACM CCS*, 2011.

[26] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *USENIX Security*, 2011.

[27] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," *Trust and Trustworthy Computing*, 2012.

[28] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[29] Y. Zhang, D. Song, H. Xue, and T. Wei, "Ad vulna: A vulnaggressive (vulnerable & aggressive) adware threatening millions," 2013, https://www.fireeye.com/blog/threat-research/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html.

[30] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications." in *USENIX Security Symposium*, 2012, pp. 553–567.

[31] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, 2011, p. 24.

[32] H. Lockheimer, "Android and security," February 2012, http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[33] "Protect against harmful apps," https://support.google.com/accounts/answer/2812853?hl=en.

[34] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1808–1815.

[35] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis-1,000,000 apps later: A view on current android malware behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[36] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated web patrol with strider honeymonkeys," in *Proceedings of the 2006 Network and Distributed System Security Symposium*, 2006, pp. 35–49.

[37] 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 *Proceedings of the 19th Network and Distributed System Security Symposium*, ser. NDSS '12, 2012.

[38] 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*, ser. MobiSys '12. ACM, 2012.

[39] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[40] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 611–622.

[41] 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*. ACM, 2014, pp. 576–587.

[42] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1105–1116.

Rui Shao is pursuing a computer science Ph.D. at Zhejiang University, China. His research interests are in mobile security.

Vaibhav Rastogi is a postdoctoral research associate at the University of Wisconsin-Madison. He received a Ph.D. in computer science from Northwestern University. His research interests are in computer security.

Yan Chen is a professor of Electrical Engineering and Computer Science at Northwestern University. His research interests include security, measurement and diagnosis for large networks and distributed systems.

Xiang Pan is a Ph.D. student at Northwestern University. His research interests are web security and Android security.

Guanyu Guo is pursuing a computer science master at Zhejiang University, China. His research interests are in mobile security.

Shihong Zou is an Associate Professor at School of CyberSpace Security in Beijing University of Posts and Telecommunications. His research interests include mobile security, IoT security, wireless networking

Ryan Riley is an Associate Professor of Computer Science at Qatar University. His research interests include intrusion detection, operating systems security, and computer architecture.