Detecting Hidden Attacks in Apps through the Mobile App-Web Interface

Mrs.S.Revathy
Assistant Professor,Department of Computer Science and Engineering, Velammal Engineering College
Chennai, Tamil Nadu
Ashvitha.V
Department of Computer Science and Engineering, Velammal Engineering College
Chennai, Tamil Nadu
Vaishnavi.N
Department of Computer Science and Engineering, Velammal Engineering College
Chennai, Tamil Nadu

----------**ABSTRACT**----------

Mobile users are increasingly becoming targets of malware infections and scams. Some platforms, such as Android, are more open than others and are therefore easier to exploit than other platforms. In order to curb such attacks it is important to know how these attacks originate. We take a previously unexplored step in this direction and look for the answer at the interface between mobile apps and the Web. Numerous in-app advertisements work at this interface: when the user taps on an advertisement, she is led to a web page which may further redirect until the user reaches the final destination. Similarly, applications also embed web links that again lead to the outside Web. Even though the original application may not be malicious, the Web destinations that the user visits could play an important role in propagating attacks. In order to study such attacks we develop a systematic methodology consisting of three components related to triggering web links and advertisements, detecting malware and scam campaigns, and determining the provenance of such campaigns reaching the user. We have realized this methodology through various techniques and contributions and have developed a robust, integrated system capable of running continuously without human intervention. We deployed this system for a two-month period and analyzed over 600,000 applications in the United States and in China 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 as well as make several interesting findings, including a rogue antivirus scam, free iPadand iPhone scams, and advertisements propagating SMS trojans disguised as fake movie players. In broader terms, our system enables locating attacks and identifying the parties that intentionally or unintentionally let them reach theend users and, thus, increasing accountability from these parties.

Keywords- **Virus, Application Permissions, Redirect and Malicious URL**

I. 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 the Web, to study so-called malvertising or malicious advertising [4], [5]. The context of the problem here is however 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. Our methodology enables us to explore the Web that is

reachable from within mobile applications, something that, we believe, is not yet done by traditional search engines and website blacklistsystems such as Google Safebrowsing. We make the following contributions.

- We have developed a framework for analyzing the appweb 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.
- In order to assist with determining the provenance of identified malicious links, we conducted a systematic study to associate ad networks with ad library packages in existing applications. Our study reveals 201 ad networks and their associated ad library packages. To the best of our knowledge, this is the largest number of ad libraries identified.
- We deployed our system for a period of two months in two continents, with one location in Northwestern University campus and the other in Zhejiang University campus. 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 SMS trojans propagating through well-known ad networks.

II. LITERATURE SURVEY

Mobile users are increasingly becoming targets of malware infections and scams. Some platforms, such as Android, are more open than others and are therefore easier to exploit than other platforms. In order to curb such attacks it is important to know how these attacks originate. We take a previously unexplored step in this direction and look for the answer at theinterface between mobile apps and the Web. Numerous inapp advertisements work at this interface: when the user taps on an advertisement, she is led to a web page which may further redirect until the user reaches the final destination. Similarly, applications also embed weblinks that again lead to the outside Web. Even though the original application may not be malicious, the Web destinations that the user visits could play an importantrole in propagating attacks.

In order to study such attacks we develop a systematic methodology consisting of three components related to triggering web links and advertisements, detecting malware and scam campaigns, and determining the provenance of such campaigns reaching the user. We have realized this methodology through various techniques and contributions and have developed a robust, integrated system capable of running continuously without human intervention. We deployed this system for a two-month period and analyzed over 600,000 applications in the United States and in China 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 as well as make several interesting findings, including a rogue antivirus scam, free iPad and iPhone scams, and advertisements propagating SMS trojans disguised as fake movie players. In broader terms, our system enables locating attacks and identifying the parties (such as specific ad networks, websites, and applications) that intentionally or unintentionally let them reach the end users and,thus, increasing accountability from these parties.

III. METHODOLOGY

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.

3.1. Triggering App-Web interfaces

Recall from previous discussion that web links in applications are often dynamically generated (such as from advertisements). Thus a static approach of extracting web links is not sufficient. Therefore, in order to trigger web links from within the application, we run the applications in a custom dyanamic analysis environment. To enable scalability andcontinuous operation, running applications on real devices is not a feasible option. 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 GUIautomatically to trigger app-web interfaces. The rest of this subsection describes

thetechniques thatwe 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.

- 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 VIII). 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 that we had developed earlier in AppsPlayground [6].
- 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.
- 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 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 [7], 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.In order to overcome this issue of essentially flat (i.e., with no hierarchical structure in the UI debug interfaces provided by Android) Webviews, we apply computer graphics techniques in order to detect buttons and widgets as a human would see them. Algorithm 1 presents the detection algorithm.

3.2. Algorithm 1Button detection algorithm

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

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 wellperforming 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.

3.3. 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.

3.3.1. 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 apparently use time-based checks 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 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 as 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.

3.3.2. 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.

3.3.3. 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 agreement among antiviruses 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.

3.3.4. 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 5 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.

3.4. *Ad Library Identification*

Applications that monetize with advertisements typically partner with ad networks and embed code called ad libraries from them in order to display and manage advertisements. Our goal is to comprehensively identify ad networks that participate in the Android ecosystem and their associated ad libraries. Such an identification is important for automatically classifying if a malicious activity is a result of an advertisement or is the responsibility of the application developer. Some simple domain knowledge, such as which ad networks are there in the market, may not provide a comprehensive list we are looking for. We instead resorted to two systematic approaches to do this identification based on the ad libraries embedded in the code.

3.4.1. Approach1

We exploit the fact that one ad network will likely be used by many applications and thus common ad library code will be found in all applications using an ad network. The native programmingplatform for Android applications is Java and Java packages provide mechanisms to organize related code innamespaces. Ad libraries themselves have packages that can serve as their identifying signatures. In our first approach, we collected packages from all applications in our dataset and created a package hierarchy together with the frequency of occurrence of each package. We sorted the packages and then manually searched the most frequent packages to identify ad libraries. For example, after sorting, packages such as com.facebook and com.google.ads appear at the top. Then we identified the nature of each package, i.e., whether it constituted an ad library, based on either prior knowledge or manually searching information about that package on the Web.

3.4.2. Approach 2

The previous approach became cumbersome when we reached frequencies of a few hundred because many non-ad packages also had such frequencies. Our alternative approach allows for comprehensive identification of ad libraries without depending on the frequency of occurrence of those ad libraries. Our second approach relies on the fact that the main application functionality is

only loosely coupled with the functionality of ad libraries. Thus, we use the technique described by Zhou et al. [8] to detect loosely coupled components in the applications. The coupling is actually measured in terms of characteristics such as field references, method references, and class inheritances across classes. Ideally, all the packages of one ad library will be grouped into one component. In reality, this does not always happen and it may also happen that classes that should have been in different components end up in the same components. However, the errors are tolerable and can be manually analyzed. The manual analysis is further eased by employing a clustering technique described as follows. We create a set of Android APIs called in an application component. This set of APIs forms a signature for the component. We map these APIs to integers to enable efficient set computations. Based on this, ad library instances with the same version have matchingAPI sets. For different versions, the sets will be similar but not identical. We run this analysis on components extracted from all applications and then use the Jaccard distance to compute dissimilarity between API sets. If it is below a certain threshold (we used 0.2), we place the components in the same cluster. Thus packages of different ad libraries end up in different clusters, and then clusters can be easily mapped to ad libraries.

3.4.3. Results

Using the two approaches, we were able to identify 201 ad networks in our dataset. To our knowledge, this is the highest number of ad networks identified. Some ad networks have ad libraries withseveral package names. For example,com.vpon.adon and com.vpadn belong tothe same network. We combine such instances together to be represented as a single ad network. More notably, Google's Admob and DoubleClick platforms are both represented as Google ads.

Note that 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 [9]. 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 likecom.cRDpXgdA.kHmZYqsQ70374and com.enVVWAar.CJxTGNEL99769.

IV. PROPOSED SOLUTION

We implemented most of our system in Python. For UI exploration, we make use of the source code of the AppsPlayground tool [10]. 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 believe the study results are generally representative. Furthermore, not being able torun ARM native code is not a fundamental limitation of our approach: thirdparty Android emulators, e.g., Genymotion, or the use of a dynamic ARM-to-x86code translation library(libhoudini) can allow runningARM code on hardware-accelerated x86 architectures [11], [12].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.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. All the resulting analysis data is stored in MySQL and MongoDB databases. Since the framework works in a distributed, concurrent manner, server-based SQL engines such as MySQL were more appropriate than serverless implementations like SQLite. SQL commands are additionally wrapped with SQLAlchemy, a library that provides object-relational mapping (ORM), generally easing the programming. We implemented the analysis of the landing pages or the final URLs in the redirection chains on top of Chromium web browser using Watir and the Selenium Webdriver framework. We use Watir and Webdriver to scriptbrowser actions for automatically loading web pages, clicking on links, automatically download content that is available on clicking links, as well as going back to the original page if a new page loads after clicking on links. All the processing is done headlessly (i.e., without any GUI) using the Xvfb display server, which is an X server implementation that does not present a screen output. Applications are run in the virtualized environment for a maximum of five minutes, with the average running time less than two minutes. The post-trigger analysis, especially the analysis of landing pages, is allowed to run for a maximum of fifteen minutes. We allow for such a long time as our crawler may traverse many links and each link may have complex redirection mechanisms that may trigger only after a short wait. A

systematic static analysis methodology to find ad libraries embedded in applications and dynamic analysis methodology consisting of three components related to triggering web links, detecting malware and scan campaigns, and determining the provenance of such campaigns reaching the user.

V. EVALUATION AND RESULTS

5.1. *Application Collection*

Our application dataset consists of 492,534 applications from Google Play and 422,505 applications from four Chinese Androidapplication 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 two 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 toGoogle's servers randomly chooses from among twenty source IP addresses. 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 IV cannot be tested on our system. Our entire usable application dataset therefore consists of a little over 600,000 applications.

5.2. *Deployment*

We deployed our system to gather results over a period of about two months from mid-April 2015 to mid-June 2015 in two locations, one at Northwestern University campus in the US and the other at Zhejian University campus in China. 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 Northwestern University location ran applications from Google Play (only the applications available from theUS) while the Chinese university location ran applications from Chinese application stores. The location where the apps are run is important because much of advertising, which forms bulk of theapp-web interaction we are studying, is targeted based on location. Theadvertisements that are seen in one locationmay not be shown in another location.

5.3. *Overall Findings*

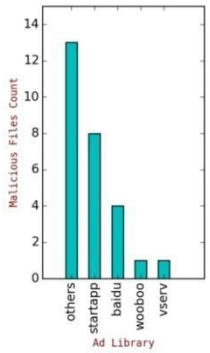

Fig. 4. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in US deployment. Libraries not resulting in malware downloads are not shown. Tapcontext malware numbers are not shown here as they are too high.

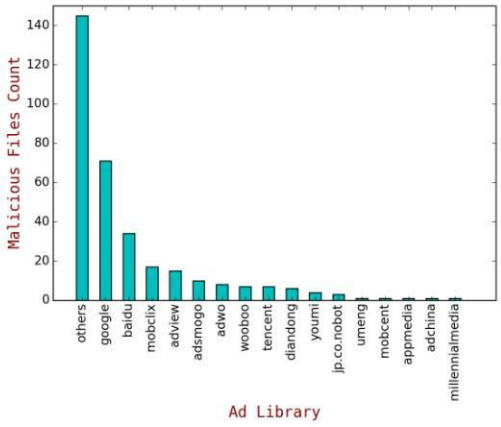

Fig. 5. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in Chinese deployment. Tapcontext and libraries not resulting in malware downloads are not shown.

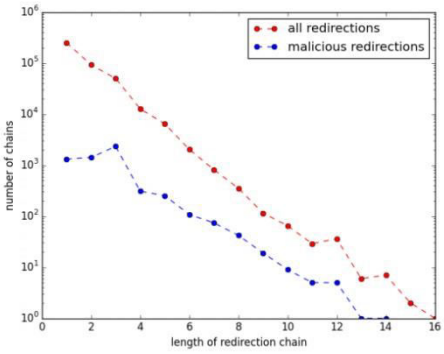

Fig. 7. Redirection Chain lengths in Chinese Deployment

Overall, we recorded a total of slightly over 1 million launches of app-to-web links in the US deployment. In the Chinese deployment, 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 the US, 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 filesof which many were simple text files or docx files. As for the number of Android applications, the US deployment collected 468 unique applications (from the Web,outside Google Play) of which 271 were found to be malicious. Alarge chunk (244)of these malicious applications comes from the antivirus scam reported in Section VI-A. Excluding this anomalous number of 244, we find that one in six applicationsdownloaded from the Web (outside Google Play) are malicious. The file numbers above do not include the applications applications hosted on Google Play. We accounted for suchseparately: 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 arelabeled as malicious (based on our criterion of being flaggedby at least 3 antiviruses) on VirusTotal. Based on our manual check of the antiviruslabels,however, all of these appear to beadware. On theChinese deployment side, we collected 1,097 unique files of which 435 are malicious. 102 of these files are from the antivirus scam of Section VI-A. Figures 4 and 5 present the distribution of malware downloads through various ad libraries in the US 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 applicationstores.

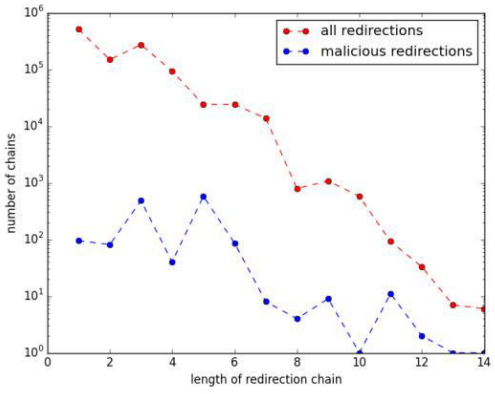

Fig. 6. Redirection Chain lengths in US Deployment

We also plot the length of redirection chains in both North American and Chinese Deployments. Note as the length of the chains increases, the two curves come closer, i.e., we have a greater fraction of malicious chains when they are longer. This was also observed by [5] and can possibly used to enhance our detection in future work.

VI. 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 explored the app-web interface, whereina 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 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.

REFERENCES

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

[2] "Malware infected as many androiddevices 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] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in Proceedings of ACM CODASPY, 2013.

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

[8] 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.[9] Symantec, "Airpush begins obfuscating ad modules," November 2012, http://www.symantec.com/connect/blogs/ airpush-begins-obfuscating-ad-modules.

[9] 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.

[10] "Genymotion," https://www.genymotion.com/.
[11] "Android-x86 running arm apps thanks to libhoudini and buildroid.org," 2012, http://forum.xda-developers.com/showthread.php?t=1750783. [13] "Celery: Distributed task queue," http://www.celeryproject.org/.
[12] 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.
[13] http://forums.makingmoneywithandroid.com/advertising-networks/ 1868-tapcontext-shit-breaking-policy-making-loosing-active-users. html#post12949
[14] http://www.androidauthority.com/armor-for-android-342192/.
[15] "Reputation of amarktflow.com," https://www.mywot.com/en/scorecard/ amarktflow.com.
[16] "Free iPad mini scam spreads via facebook rogue application," https: //nakedsecurity.sophos.com/2012/10/31/free-ipad-mini-facebook/.
[17] "Apple iPad scam," http://blog.spamfighter.com/software/ apple-ipad-scam.html.
[18] "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/.
[19] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," ACM SIGPLAN Notices, vol. 48, no. 10, pp. 641–660, 2013.
[20] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in ACM SIGPLAN Notices, vol. 48, no. 10. ACM, 2013, pp. 623–640.
[21] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: detecting and characterizing ad fraud in mobile apps," in Proc. of NSDI, 2014.
[22] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smvhunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in Proceedings of Network and Distributed Systems Security (NDSS), 2014.
[23] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014, pp. 190–203.
[24] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in 23rd USENIX Security Symposium (USENIX Security 14). USENIX Association, 2014, pp. 1021–1036.
[25] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014, pp. 204–217.
[26] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in ACM CCS, 2013. [29] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in IEEE Security and Privacy, 2015. [30] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," inProceedings of the 12thannual international conference on Mobile systems, applications, and services. ACM, 2014, pp. 123–134.
[27] 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.
[28] 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.
[29] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in USENIX Security, 2011.
[30] 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.
[31] 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.
[32] 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.
[33] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications." in USENIX Security Symposium, 2012, pp. 553–567.
[34] 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.
[35] H. Lockheimer, "Android and security," February 2012, http:// googlemobile.blogspot.com/2012/02/android-and-security.html.
[36] "Protect against harmful apps," https://support.google.com/accounts/ answer/2812853?hl=en.
[37] 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.

[38] 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 androidmalware behaviors," in Proceedings of the the 3rd International Workshop on BuildingAnalysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

[39] 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.

[40] 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.

[41] 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.[46] D. Arp, M. Spreitzenbarth, M. Hubner, 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.

[42] 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.

[43] 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.

[44] 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.

[45] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM, 2013, pp. 329–334.