On the Origin of Mobile Apps: Network Provenance for Android Applications

Ryan Stevens
UC Davis
rcstevens@ucdavis.edu

Jonathan Crussell
UC Davis
jcrussell@ucdavis.edu

Hao Chen
ShanghaiTech University
chenhao@shanghaitech.edu.cn

ABSTRACT

Many mobile services consist of two components: a server providing an API, and an application running on smartphones and communicating with the API. An unresolved problem in this design is that it is difficult for the server to authenticate which app is accessing the API. This causes many security problems. For example, the provider of a private network API has to embed secrets in its official app to ensure that only this app can access the API; however, attackers can uncover the secret by reverse-engineering. As another example, malicious apps may send automatic requests to ad servers to commit ad fraud.

In this work, we propose a system that allows network API to authenticate the mobile app that sends each request so that the API can make an informed access control decision. Our system, the Mobile Trusted-Origin Policy, consists of two parts: 1) an app provenance mechanism that annotates outgoing HTTP(S) requests with information about which app generated the network traffic, and 2) a code isolation mechanism that separates code within an app that should have different app provenance signatures into *mobile origin*. As motivation for our work, we present two previously-unknown families of apps that perform click fraud, and examine how the lack of mobile origin information enables the attacks. Based on our observations, we propose *Trusted Cross-Origin Requests* to handle point (1), which automatically includes mobile origin information in outgoing HTTP requests. Servers may then decide, based on the mobile origin data, whether to process the request or not. We implement a prototype of our system for Android and evaluate its performance, security, and deployability. We find that our system can achieve our security and utility goals with negligible overhead.

1. INTRODUCTION

Many mobile services consist of two components: a server providing an API, and an application running on smartphones and communicating with the API. An unresolved problem in this design is that it is difficult for the server to authenticate which app is accessing the API. For example, Snapchat[1] wishes to allow only its official app to access its API. However, attackers reverse-engineered the API and produced malicious third-party apps to steal Snapchat users' credentials and photos [11]. Unfortunately, there is no robust way to protect against such an attack on Android or iOS. For example, common iOS HTTP libraries automatically include the app's name in the HTTP header of outgoing requests, but attackers can easily forge the header either by setting the value manually or by using sockets directly. As long as apps can use sockets directly, relying on HTTP libraries to provide app provenance in HTTP requests would be insufficient. An alternative is to embed a secret key in the app to create an authentication token for accessing the API, but attackers can reverse-engineer the app to uncover the secret, as in the case of Snapchat. Clearly we need a better solution.

To solve this problem, we propose a system that inserts unforgeable *app provenance* in outgoing requests, which the servers can then use to identify the app making the request. To do so, we add a network proxy on the device that observes all outgoing HTTP(S) communications and injects an app identifier into the HTTP header (this approach could be extended to work with other application layer protocols, but as we will show in Section 8.3, the majority of apps communicate over HTTP). This unforgeable app identifier in the header allows the server to identify the app and enforces access control accordingly. The benefit of using a proxy is that the proxy is agnostic to how the network requests were generated, including using HTTP libraries, sockets, WebView, or native code (in Android). In Section 6, we build such a system for Android and evaluate it in Section 7. We call this approach Trusted Cross-Origin Requests (TCOR for short), as we trust the operating system to add the unforgeable header. If widely deployed, it would enable private network APIs for apps, and even enable API providers to "sell" their API to developers without fear of API keys being stolen.

However, the proxy alone can identify provenance only at the app granularity, which cannot detect all forgery attacks because of the way how mobile apps are developed. For example, on Android, many apps include ad libraries to display ads to generate revenue. These ad libraries are provided by a third party called an ad provider, but are included as part of the app's code. Thus, our proxy presented above would

CODASPY'16, March 09-11, 2016, New Orleans, LA, USA
© 2016 ACM. ISBN 978-1-4503-3935-3/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2857705.2857712

[1]Snapchat is a smartphone messaging service that allows users to send ephemeral images that cannot be stored by the receiver.

not be able to distinguish between ad requests generated by the ad library and those forged by the app. When apps automatically generate ad requests without user interaction, they are committing ad fraud, as no real user is viewing or clicking ads. To motivate the need to protect against such an attack, we present two previously-undisclosed families of Android click fraud malware in Section 3. To defend against this attack, we need a mechanism to divide an app's code into different *mobile origins* — e.g., app code and ad library — so that the proxy can distinguish network requests by mobile origin and tag the correct mobile origin in the HTTP header. Since inner-app code isolation has been studied in Android, we use an existing system called LayerCake [21] for implementing code isolation in our system.

We have discussed the need for two security mechanisms in smartphone operating systems: app provenance in network requests and inner-app code isolation. We will call the combination of these mechanisms the Mobile Trusted-Origin Policy (MTOP for short), loosely motivated by browsers' Same-Origin Policy. Our contributions are as follows:

- We propose the Mobile Trusted-Origin Policy and Trusted Cross-Origin Requests to provide both app provenance in network requests and inner-app code isolation to enable private network API and defend against ad fraud.
- We report two previously-unknown families of click fraud apps by leveraging existing methodology for detecting ad fraud behavior in apps and clustering apps based on code similarity. For each family, we classify their fraudulent behavior and identify what aspects of mobile systems they use to perform fraud.
- We implement Mobile Trusted-Origin Policy on Android and evaluate its performance, security, and deployability. We find that our system can achieve our security and utility goals with reasonable performance overhead.

2. BACKGROUND

2.1 Android Advertising

Figure 1 shows an overview of Android app advertising:

- The *publisher* is the app owner that is paid to show ads in her app. Publishers register with *ad providers*, who maintain relationships with publishers and *marketers*. Marketers pay ad providers to have their ads shown.
- An ad provider gives a registered publisher an *ad library* to embed in her app and a *publisher ID* to identify her. The ad library has an API to display ads.
- When instructed to show an ad, the ad library sends an *ad request* to the ad provider's *ad server*. The ad request contains the publisher ID and any ad *targeting* information (e.g. the user's age or gender).
- The ad server responds with ad metadata that includes a URL for the ad's content and a *click URL* where to redirect the user if the ad is clicked.
- Publishers are paid for *impressions*, where an ad is successfully requested and shown, and for *clicks*, when the user clicks the ad. Clicks are tracked by redirecting the user through the ad server before sending them to the marketer's webpage.

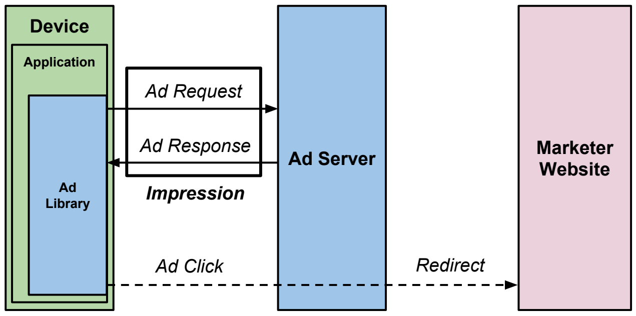

Figure 1: Overview of the Android in-app advertising model.

The primary difference between this ad serving model and the web ad serving model is how ad code is included with the publisher's content. Instead of code libraries, websites include external Javascript through `<iframe>` objects, called *ad tags*. This is necessary as the Javascript code must be hosted on the ad server to make network requests to the server due to the restrictions of the Same-Origin Policy. Once the ad request is made, the infrastructure to select and track ad views remains unchanged. In fact, many ad libraries choose to implement the requesting and display of ads by opening web ad tags in a `WebView` environment, which acts as an embedded browser in the app's UI.

3. MOTIVATION

In this section we motivate the need to mediate apps' network requests via MTOP. We first discuss private web APIs and then present two previously unpublished families of Android click fraud malware.

3.1 Private Web APIs

As previously mentioned, it is difficult for a network API provider to reliably ensure that only its mobile app can access its API. This difficulty contrasts with the case of such APIs for web apps, because web code is restricted by the Same-Origin Policy, preventing a web app on another website from completing a cross-domain network request. A recent study by Viennot, et. al. [26] found that thousands of apps embedded authentication tokens for web services, including API tokens for cloud computing services such as Amazon Web Services, as well as OAuth tokens for various social media sites. This is a high security risk because attackers can uncover these tokens by reverse-engineering, but developers have no other solutions for authenticating their apps with network APIs. Once an API token has been discovered, attackers can craft phishing apps or trojans that imitate the functionality of a legitimate one while secretly stealing user credentials or monitoring their behavior, as mentioned with Snapchat in the introduction. In addition, unscrupulous developers could use competitors APIs to bolster their own apps' functionality. All this can be done without exploiting the underlying operating system, motivating the need for a system-centric solution.

3.2 Case Study: Click Fraud

Ad fraud (or *click fraud*) is the practice of "viewing" and "interacting" with ads in an automated way to artificially inflate revenue. For example, an unscrupulous website publisher may inflate her ad revenue by having an automated script visit her website and click on ads. Ad fraud is a serious security issue as digital marketers who pay to have their ads shown will not receive any commercial benefit for ads shown to scripts.

In order to receive revenue, fraudsters must remain undetected while issuing large numbers of ad requests and clicks. To do so, fraudsters use botnets to instruct real devices to run code that periodically visits the fraudsters' webpages in the background and clicks on the ads located there, to generate revenue for the fraudsters. The use of bots allows fraudsters to use the characteristics of the compromised devices to generate varied ad traffic (for example, fraudulent requests from a botnet would use many different IP addresses). Unfortunately, Android apps distributed through a market can give fraudsters these same characteristics without the need to exploit users' machines.

3.2.1 Click Fraud Detection

To find apps that perform click fraud, we first run them through MAdFraud [3]. MAdFraud automatically identifies ad requests and clicks in mobile app traffic that is recorded from running the apps in an emulator. It uses simple rules to flag apps with suspicious ad behavior, which are candidates for further investigation. As a starting point, we use the initial MAdFraud dataset of 130,339 apps collected from 19 Android markets, which flagged 12,421 apps as potentially performing ad fraud. We then ran an additional 60,726 apps through MAdFraud, from the developers that uploaded these flagged apps. In total, MAdFraud identified 615 apps which send ad clicks without user interaction, which is too many to manually investigate. We chose to use AnDarwin [2] to cluster any detected click fraud apps into families based on code similarity. This significantly reduces the number of apps which need to be manually investigated.

Originally, AnDarwin excluded any feature appearing in at least N apps. This threshold-based approach is problematic for comparing our small set of apps that issue clicks as 1) we may set the threshold too low and exclude all click modules (code for committing click fraud that is shared across an app family), or, 2) we may set the threshold too high and fail to exclude libraries. Instead, we take an approach inspired by term-frequency inverse-document-frequency (TF-IDF). Rather than exclude features that are present in many apps, we instead give them a small weight based on the inverse of the number of apps they appear in. Now that features have weights, we cannot compare apps using the Jaccard similarity, as done in the original work. Instead, we compare apps using the Cosine similarity of their feature vectors which are weighted using the IDF weights. Ultimately, AnDarwin placed our 615 click fraud apps into 65 clusters. From our analysis of the app families, we found two clear examples of Android click fraud malware: Pixcel and AppsGeyser.

3.2.2 Pixcel

The first family of click fraud apps we present all contain a common package called `com.pixcel.core`, which contains code to perform fraud against a number of ad providers. The package implements an Android service which issues ad and click requests in the background when started. The ad fraud service first makes a request to `nucleardroid.com` to receive parameters used for issuing fraudulent ad traffic. An example response is shown in Figure 3 in the Appendix. The response contains instructions for the service, indicating which ad providers to perform fraud against (in this case Madvertise, Mobfox, and Tapjoy), which publisher IDs (sometimes called "siteid") to use, and how many fraudulent requests to make to each. A timing component uses parameters in the response (specifically, the `countperday` parameter) to start the service at regular intervals, or it defaults to running hourly if no parameters have been loaded. For each ad provider, the Pixcel package contains a custom module that is able to send ad requests to the ad provider's ad server, parse the response, and then issue click requests if instructed to do so. Interestingly, each custom module has very different code from the ad library of the associated ad provider. This implies that the authors of Pixcel spent the time to reverse engineer the ad server's API, instead of simply modifying an existing library. Figure 4 shows a code snippet (after decompilation) from the ad fraud service that issues fraudulent requests and clicks after receiving instructions. For each entry returned by `nucleardroid.com`, it dynamically instantiates the appropriate custom module to issue ad requests and clicks. The custom modules first issue `countfake` ad requests (from Figure 3), followed by `counttry` impressions and click requests.

The apps containing Pixcel are from various developer accounts and Android markets. Table 1 shows a breakdown of app packages which contain Pixcel and which markets these apps reside. Some markets have removed the Pixcel apps between the time we crawled the market and the time we discovered Pixcel. The markets may have discovered the fraud separately, or they may have removed them as many of the apps are "spam" apps that contain little content and are all essentially identical except for superficial cosmetic differences. To measure which ad providers Pixcel targets, we queried the Nucleardroid server hourly for three days and observed which ad providers and accounts are returned, using the package names in Table 1 as parameters. In total, Nucleardroid returned instructions to perform fraud using 43 publisher accounts from 5 different ad providers[2]. Since discovering Pixel, we have reported the fraud to each targeted ad provider and had the publisher accounts terminated.

The Pixcel module is an interesting case study. The practice of receiving instructions from a centralized server prior to performing fraud is reminiscent of botnets, where bots receive instructions from a command and control server. From looking at the responses, its clear the fraudsters wish to remain stealthy. Not only do they rotate which ad providers and publisher IDs they use, but they also balance the number of clicks with the number of impressions issued. In addition, because the publisher ID and behavior are chosen dynamically, the fraudsters can easily modify the app's behavior if one of their accounts is terminated. Given that Pixcel attempts to remain stealthy and can be easily reconfigured, it would be more effective to prevent ad fraud on the device than via detection of individual cases. By adopting the Mobile Trusted-Origin Policy presented in subsequent sections, such fraud techniques would not be possible.

3.2.3 AppsGeyser

Here we discuss a second family of apps which issue click requests when run. These apps are all made with the Apps-

[2]Tapjoy, Madvertise, Vserv, MobFox, and Mojiva

Package Name	Markets	Distinct Apps	Install Count
com.pixcel.DroidSaver	Play*, Opera, SlideME*	3	10,619
com.pixcel.FoxSaver	Play*, Opera, SlideME*	3	5,547
com.pixcel.OlympicWallpapers2012	Play*, Opera, SlideME*	2	2,077
com.pixcel.PRDroid	Play*, Opera, SlideME	3	4,963
com.pixcel.MorzeDroid	Play*, Opera, SlideME	2	3,874
com.pixcel.NuclearFlashlight	Opera, SlideME	2	4,072
com.wallpapersdroid.bestcarwallpapers	Opera, SlideME*	2	2,596
com.wallpapersdroid.housemdwallpapers	Opera, SlideME*	2	997
com.wallpapersdroid.offroadcarwallpapers	Opera, SlideME*	2	1,139
com.wallpapersdroid.porschewallpapers	Opera, SlideME*	2	2,936
com.wallpapersdroid.luxurycarwallpapers	Opera	1	238
com.wallpapersdroid.lamborghiniwallpapers	Opera	1	337
com.wallpapersdroid.bmwwallpapers	Opera	1	490
com.wallpapersdroid.peugeotwallpapers	Opera	1	148
com.wallpapersdroid.animalworldwallpapers	Opera	1	225
com.wallpapersdroid.supersedanswallpapers	Opera	1	84
com.wallpapersdroid.mobilewallpapers	Opera	1	335
com.wallpapersdroid.coolwallpapers	Opera	1	221

Table 1: Information about the apps containing the Pixcel click fraud module. An astericks next to a market denotes that the app is no longer available through the market. Distinct apps are counted based on the SHA-1 hash of the APK file. For markets that provide a range for install counts, we use the lower bound to ensure that our total install count is a lower bound on the actual number of installs.

Geyser framework[3], and perform fraud by loading malicious Javascript into a `WebView` that displays ads.

The AppsGeyser framework allows developers to develop web applications with HTML and Javascript and then wrap them in an Android app that can be distributed through Android Markets. We found 211 click fraud apps that were built using AppsGeyser, based on our clustering. Apps built through AppsGeyser will show advertisements when run. As part of the AppsGeyser business model, AppsGeyser receives a portion of revenue for ads shown in the apps in exchange for providing their app building service for free.

The AppsGeyser framework loads ads using a `WebView`. Upon starting the app or refreshing an ad, the framework makes a request to `ads.appsgeyser.com/`, which returns Javascript that bootstraps the `WebView` for showing ads. Under normal circumstances, benign Javascript is loaded and ads are displayed as usual. This Javascript sets up timers for refreshing the ad after some amount of time, configures the "close" button in the top right corner of the ad space, and sets up an `onClick` listener for handling when the user clicks an ad. Finally, a request is made to `postupdate.info/delivery/ajs. php` with some parameters, which returns an ad tag from another ad provider that eventually loads and displays the ad. In our experiments, we observed that these ad tags used a variety of ad providers, including InMobi, InnerActive, Jumptap, Madgic, MassiveImpact, Mocean, and Vserv. The practice of reselling ads from one ad provider to another is common practice, so fetching third party ad tags is not fraud.

However, rarely, we find apps making a request for additional Javascript content located at `postupdate.info/carousel/ad_track.js`. This additional Javascript contains obfuscated code that performs click fraud. The de-obfuscated Javascript content can be found in the Appendix. When run, it scrapes the HTML DOM for any anchor tags, as well as the DOM of any included `<iframe>` objects. For each URL, it creates a 1 pixel by 1 pixel invisible image, sets the `src` attribute of this image to be the click URL, and then inserts this image into the DOM. The result of this is that all click URLs in the populated ad tags of the third-party ad providers will be followed, which results in fraudulent click requests. There is a 1-to-1 mapping between requests to `postupdate.info/carousel/ad_track.js` and clicks detected by MAdFraud.

Normally, this kind of attack would not be feasible due to the Same-Origin Policy of browsers and `WebView`. However, we found that some ad requests to third-party ad providers contain the `Access-Control-Allow-Origin: *` header in the servers' response, which explicitly disables the Same-Origin Policy on delivered content [16]. Due to this vulnerability, the Javascript is able to extract click URLs and follow them by creating image objects which point to the click URLs. A properly configured Same-Origin Policy could have prevented these attacks, and the case study illustrates how trivially ad fraud can be performed without origin protections. This is especially disconcerting given that AppsGeyser app code runs in the same WebView as the ad code. If AppsGeyser were configured to have the app code and ad code run in separate mobile origins, then apps would be protected from misconfigurations in ad code. Additionally, with Trusted Cross-Origin Requests, WebViews with SOP misconfigurations or that have the SOP disabled (using the `setAllowUniversalAccessFromFileURLs()` API) would at least have app provenance information appended to outgoing requests in the case that malicious Javascript was loaded into the insecure WebView.

4. GOALS

Our goal is to provide app provenance for network requests so that servers can differentiate authorized and unauthorized app traffic. In order to do so, we must also isolate code within an app into separate protection domains, called *mobile origins*, when the code should have different app prove-

[3]`http://www.appsgeyser.com/`

nance signatures. Section 5.1 describes how our framework unambiguously identifies and reports mobile origins. The Mobile Trusted-Origin Policy provides the following security benefits:

- Code in one mobile origin should not be able to access or manipulate the code and/or data from other mobile origins. However, code from separate origins can communicate using message passing. On Android, this communication is done through Intents. (On the web, cross-origin communications can be done using *cross-document messaging.*)

- Code should be able to communicate with only authorized web domains, where the definition of "authorized" is left up to the receiver of the communication. This is because we wish to protect against network forgery attacks without restricting the rich functionality of legitimate apps. Many legitimate apps communicate with several different domains. If we were to require each app communicate with only one domain, as the web's Same-Origin Policy does, we would break many benign apps.

We discuss the first point in Section 5.2 and the second in Section 5.3. The former requirement isolates app code and third-party libraries. The latter requirement ensures that malicious apps cannot use the device as a network bot to send unwanted or fraudulent traffic to arbitrary websites, as well as restricts private API to only authorized apps.

4.1 Assumptions

We make the following assumptions in designing our system:

- The user, mobile operating system, and system apps are trusted, but user-installed apps are untrusted and potentially malicious.

- There exist mechanisms to split and sandbox app and library code on the user's device. In our implementation, we chose LayerCake.

- Malicious code cannot exploit the operating system or sandbox to escalate privilege.

- The user has not rooted the device (a rooted device has no restrictions regarding which apps can run code as the superuser).

4.2 Non-Goals

Our goal is to prevent malicious apps running on a benign user's device from attacking the other apps on the device or using the device as a bot to attack network sites. Thus, we do not consider the case where attackers run their code on their own devices or operating systems (just as the web's Same-Origin Policy does not consider the case where attackers launch attacks on their own machines or modified browsers), because such attacks would not have the properties that make the attacks presented in Section 1 useful, such as access to user data or a diverse range of IP addresses. Additionally, we do not aim at preventing other forms of attacks. For example, our framework does not defend against apps that exfiltrate sensitive user data to the attacker's server. Other systems have already been proposed to detect these kinds of attacks [8, 13]. Just as the SOP does not protect web apps against malicious code injected via cross-site scripting (XSS), the MTOP does not protect vulnerable apps that are tricked to run untrusted code.

5. DESIGN

Here we discuss the design of MTOP in order to achieve the goals presented in the previous section. In Section 6, we describe how we implemented the following design.

5.1 Identifying Mobile Origins

Our first challenge is how to determine the mobile origin of code and data contained in an app. To identify mobile code origins, we need a globally unique identifier for each app that an attacker cannot forge. For Android, one might consider an app's package name as its mobile origin, as the package name identifies the app uniquely on the device (e.g. *com.example.game*). Unfortunately, package names are not globally unique, as different apps on different markets may have the same package name. Instead, we take advantage of digital signatures. Code signing allows content to be unambiguously and unforgeably assigned an origin: libraries are signed by the library developer's key before being distributed, and apps are signed by the app developer's key before being uploaded to markets. The public key used to verify the app's signature is sufficient for identifying the app's mobile origin; given a large key size, app public keys should be globally unique.

This is analagous to how Android uses signatures to allow apps to share a UID: two apps which are signed by the same key may share a UID (effectively running both apps in the same mobile origin). This differs from how certificates are used for authentication on the web (i.e., in SSL/TLS). Authenticating a website requires that the certificate be signed by a certificate authority. This is because users need to make security decisions based on the identity information stored in the certificate, so they need the certificate authority to bind the identify to the certificate. By contrast, our system requires a trust relationship between the server and developer to be set up beforehand. Once established, the server is expected to remember the binding between its partners and their certificates, allowing the server to easily determine if a certificate is trusted (e.g., is the certificate its own or its partner's); it does not rely on the identity information in the certificate and therefore requires no certificate authority. Without reliance on a certificate authority, developers are free to self-sign their apps, which is already common practice in Android app publishing.

In the case of ad servers which receive ad requests from reselling, any resold ad requests (i.e., the publisher ID is from a trusted affiliate) would simply be trusted; the ad server would not check the TCOR header and assume their affiliate did the check. This is reasonable since affiliate ad networks require a certain amount of trust (often times the context of the original request is lost in resold ad requests, making ad fraud detection difficult on affiliate traffic [24]). Ad spammers would not have incentive to forge affiliate requests, as they would not receive revenue for the requests.

5.2 Origin Sandboxing

In the current version of Android, apps are sandboxed from each other, however, all code included within an app runs in the same sandbox. For example, a developer who wishes to monetize her apps may include ads, but this requires including an ad provider's library with her app's code. The library code runs with all the privileges that her app has been granted, which leads to privacy concerns; for example ad libraries automatically use these privileges to collect user data [23]. To reap the benefits of the Mobile Trusted-Origin Policy, app and library code should be split into separate origins and run in separate sandboxes.

There are several ways that app and library code can be split. Because of the aforementioned privacy infringements from ad libraries, recent research has primarily focused on splitting app and ad code. This previous work provides an excellent foundation upon which to enforce our policy. Some examples include:

- AdSplit [22]: Adds provenance to data and actions, such that the operating system can enforce policies dictating how data and actions can pass between app and ad code.
- AFrame [27]: Separates ad libraries into their own apps, leveraging Android's existing access control policies. Modifies the framework to allow app UIs to be embedded within each other.
- LayerCake [21]: Like AFrame, runs library code in a separate process. Unlike AFrame, generalizes to any scenario that requires UIs to be embedded within each other, ensuring the code that controls each UI will not run in the same sandbox.

Any of these approaches could be used to sandbox code and data from separate origins on Android. Leveraging AFrame and LayerCake requires libraries to be distributed as separate Android apps. On the other hand, AdSplit would allow an app to contain multiple origins and sandboxing could be achieved within a single app. We build our system on top of LayerCake and discuss our reasoning for doing so in Section 6.1.

5.3 Trusted Cross-Origin Requests

In Section 3.2, we saw an example of a family of apps that can issue fraudulent ad requests and clicks because there is no mechanism in Android to mediate apps' network communications. To mitigate this, our framework should be able to restrict which domains code can contact, so that untrusted code cannot silently forge arbitrary network requests from the user's device. On the web, code can only talk to the origin (i.e., domain) from which it was fetched. However, mobile apps are expected to be able to communicate with many domains (an email client app, for example), and thus a policy that only allows communication to one domain would be overly restrictive in the mobile space. To alleviate this, we propose *Trusted Cross-Origin Requests*, which shifts the decision of whether a mobile origin is authorized to contact a domain from the OS to the server the app is contacting. The mobile origin of the app is included in outgoing HTTP requests by the operating system, such that it cannot be forged by apps. If the server wishes to accept the origin, it responds with the requested content as usual. However, requests from unauthorized origins should receive a `Client Error 4xx` HTTP response status code, such as `403 Forbidden`, to indicate that the request was unauthorized.

An alternative to this design would be to let the app developer annotate each app with a list of authorized domains, e.g., by adding elements to the app's *Manifest* file, which is already used to grant permissions. This approach is problematic for two reasons. First, since we do not trust apps in our threat model, we cannot just let the app developer declare which domains her app can visit. Therefore, the market or the user must determine whether to authorize the app to visit those declared domains; however, neither party has enough information to make an informed decision. Second, this would break the existing advertising model as some content is hosted on the digital marketer's domain, which the developer cannot know ahead of time. For these reasons, we choose to let the server authorize.

Our approach has a similar effect to *cross-origin resource sharing* (CORS) on the web [16], where the server indicates in its response which origins in the web page may read the content in the response. CORS makes sense for browsers, as the browser always receives the content in the response; the CORS header only enables cross-origin reads for the specified domains. However, for the Mobile Trusted-Origin Policy, apps that make unauthorized network requests will not receive any content. Therefore, it is more economical for the server to refuse to deliver content to unauthorized apps up front.

To add Trusted Cross-Origin Requests to Android, we propose adding a custom HTTP header, `X-Mobile-Origin`, that contains information about the app that is making the request. This header will be added by the device and the server will be free to use it as needed. Some HTTP servers are designed to communicate with all clients and may ignore the header completely. On the other hand, servers that host a private API may restrict communications to only a few origins. Finally, some servers may wish to maintain a whitelist of authorized apps. Server whitelists are commonplace among services already — e.g., publisher IDs for ad providers, username/password credentials for web APIs, or symmetric keys as in Section 3.1. However, these credentials are all subject to replay attacks once the attacker has identified or reverse-engineered them from the apps. By contrast, the signatures used by Mobile Trusted-Origin Policy cannot be replayed as long as the developer's signing key has not been compromised.

To mitigate the privacy concern that a network eavesdropper may determine what apps a user is running by examining the `X-Mobile-Origin` header, we require apps that need this header to opt in by requesting a special permission, which our system uses when modifying the proxied traffic. Apps that do not opt in will have an empty value for `X-Mobile-Origin` in outbound requests (see Section 8.2).

6. IMPLEMENTATION

Using the design presented in the previous section, we develop an implementation of the Mobile Trusted-Origin Policy for Android.

6.1 Origin Sandboxing

As previously mentioned in Section 5.2, we use LayerCake to sandbox code and data from different mobile origins. We chose to build our framework on LayerCake because it is simpler than AdSplit, general enough to handle more than just ad libraries, and is implemented for a more recent version of Android than the alternatives. LayerCake adds the concept of *embedded activities* to Android, allowing for one app's UI to be embedded in another app without the parent or child UI being able to forge touch events on the other.

In this way, libraries can be separated into a different apps, and sandboxing is achieved through Android's current app isolation mechanisms. The decision to use LayerCake means that code from different origins will need to be distributed as separate apps. Regular apps and libraries may communicate using `Intents`, which allow apps to send messages between each other in Android. This may seem heavy-handed, but it is reasonable given that not all libraries will need to be run in a separate origin, only libraries which require a unique TCOR header. Additionally, various network-enabled libraries on Android already require another app to be installed to work such as the Facebook SDK [4] or Adobe Air [5].

6.2 Trusted Cross-Origin Requests

6.2.1 Failed Attempts

Our first idea was to use app rewriting [5] to modify apps so that all calls to make HTTP requests are wrapped in code that appends the origin header. This works in the simplest of cases but has many drawbacks. First, it requires rewriting every app to add the header logic. Second, and more severe, it also requires rewriting every app to remove existing logic that would forge the header. The second drawback would be very difficult to solve as it would require extensive program analysis. Third, it is prone to omission as existing rewriting tools require a list of method signatures to rewrite. Finally, this approach does not cover native code and would have difficulties when apps use non-library methods for making HTTP requests (such as writing to a TCP socket directly).

Next, we considered modifying the popular Android HTTP libraries to automatically append the origin header. However, this solution has many of the same limitations as the rewriting solution: it cannot handle apps that include their own HTTP libraries, native code, or apps that use TCP sockets directly.

6.2.2 Framework Modifications

Our solution solves all of the above issues by using a transparent HTTP proxy to capture and set the origin header for all HTTP requests made by apps installed on the device. We handle HTTPS traffic by installing a new certificate authority (CA) on the device and allowing our proxy to man-in-the-middle apps' HTTPS connections using the private key of the new CA. Figure 2 shows an overview of our implementation.

We implement the Mobile Trusted-Origin Policy for Android using a new system app. Our system app has two components: a Service that manages the proxies and a `BroadcastReceiver` that listens for apps being installed or removed. When an app is installed, the `BroadcastReceiver` instructs the Service to create a new proxy and then generates a user-ID-specific `iptables` rule that redirects all outbound HTTP and HTTPS traffic from the app to the newly created proxy. Since each app in Android is installed with a unqiue Linux user ID, UID granularity is sufficient[6] When an app is removed, the `BroadcastReceiver` removes the `iptables` rules and terminates the app's proxy.

[4]`https://developers.facebook.com/docs/android/`
[5]`https://play.google.com/store/apps/details?id=com.adobe.air`
[6]There is an exception to this rule: apps signed by the same key may share a UID. Given that we define mobile origin by signing key, this is not a problem.

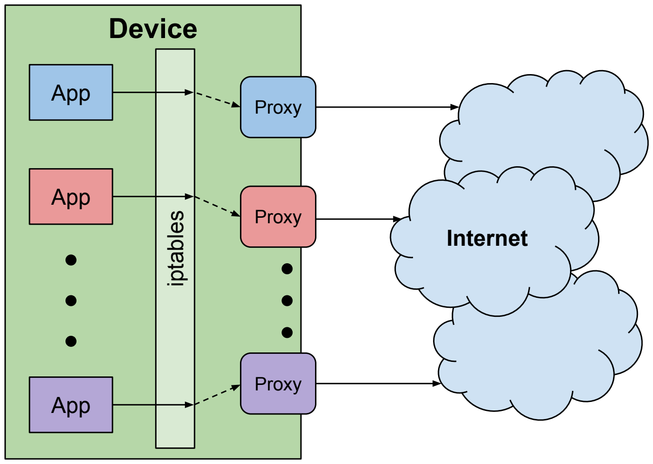

Figure 2: Overview of our implementation of the Mobile Trusted-Origin Policy for Android.

Once the `iptables` rule is in place and the proxy is running, the app can begin to make HTTP requests. These requested will be transparently redirected through our proxy which acts as forwarding proxy. For every request, the proxy modifies the HTTP request header to include the origin of the app that made the request before forwarding the request to the server. The origin header is populated with the SHA1 of the public key used to sign the app, which the proxy retrieves from Android's `PackageManager` when the app was installed. This signature will be unique to apps from the same developer, assuming that private keys are not disclosed. The proxy then forwards the response from the server back to the app which made the request.

For HTTPS traffic, there are a few additional steps. On first boot, the system app creates a new CA and installs it as a trusted authority on the device, keeping the new CA's private key (we should not do this at system compilation time as we do not want different devices to share the same CA, in case the private key becomes known). When an app makes an HTTPS connection, it will be redirected to the proxy. The proxy first fetches the destination server's certificate, extracts identity information from the sever certificate, creates a new certificate using the extracted server's identify information, and signs the new certificate using the new CA's private key. Then, the proxy returns the new certificate to the app during the SSL/TLS handshake. This allows the proxy to perform a man-in-the-middle modification on the HTTPS requests from the app to the server where it modifies the HTTP headers to include the app's origin. In the special case that the server's certificate is not signed by a trusted CA, our system proxy will sign the new certificate with an ephemeral key, so that we do not elevate the trust level of the destination server.

7. EVALUATION

Here we evaluate to what extent MTOP achieves the goals presented in Section 4. We evaluate our security goals and make a special case for deployment concerns, as any real-world adoption of our system would have to deal with partial deployment on devices, at least initially. Finally, we investigate the performance overhead of proxying HTTP traffic and adding the TCOR header.

7.1 Security

In Section 4, we defined two goals of our framework: 1) sandbox code and content from separate mobile origins, and 2) allow servers to detect unauthorized communications via mobile origins. We achieve the first goal by separating libraries and regular apps into separate Android apps and using LayerCake to allow them to share the screen. We achieve the second using the TCOR which allows servers to decide which mobile origins can communicate with it. Because our proxy determines mobile origin per UID, even HTTP requests generated by the app in a `WebView` (a UI object that acts as an embedded browser window), will have the correct mobile origin header. However, this means that any `<iframes>` in loaded websites will also be Trusted Cross-Origin Requests as they will also contain the app's origin header. For this reason, web APIs that allow for writes should contain *cross-site request forgery* tokens to protect from these "cross-mobile-origin request forgery" attacks.

Malicious code authors may attempt to circumvent TCOR by proxying their traffic through their own server to remove the header. As mentioned in Section 4, we do not consider the case where attackers run their code on their own devices or operating systems. In this case, proxying traffic would have the side effect of limiting all malicious traffic to a small set of IP addresses, which could be detected by private web APIs or ad networks.

7.2 Deployment

We consider the difficulty of deploying the Mobile Trusted-Origin Policy, and what security concerns arise if there is partial adoption from devices, servers, and apps.

7.2.1 Complete Deployment

When all mobile devices support MTOP, no app can fabricate its mobile origin, so receiving servers will be able to differentiate legitimate and unauthorized traffic from apps, effectively prohibiting unauthorized cross-domain communications. Note that just because cross-domain communications are forbidden doesn't mean that servers can rely on this property for client authentication. For example, a web server should not assume that all incoming XMLHttpRequests are generated by its pages, and neither should a mobile server assume that all incoming requests are generated by its apps, because an attacker can generate both these requests from his computer without using a web browser or mobile OS.

7.2.2 Partial Deployment

When Mobile Trusted-Origin Policy is partially deployed on mobile devices, it has the following implications.

Servers not supporting Mobile Trusted-Origin Policy.

If a server does not wish to support MTOP, meaning it accepts communications from all apps, it needs no modification to handle requests from devices supporting Mobile Trusted-Origin Policy. These servers can simply ignore the TCOR HTTP headers (provided that the HTTP header would not conflict with any headers added by the client itself).

Servers supporting Mobile Trusted-Origin Policy.

When a server supports MTOP but not all mobile devices do, the server would encounter two problems when handling requests from these non-compliant devices:

1. The requests from the server's authorized apps would not have the required HTTP header.

2. A malicious app would be able to fabricate the required HTTP header.

To solve the first problem, the server's authorized apps could add the required HTTP header by itself as a temporary compatibility measure. The second case, on the other hand, essentially reduces non-compliant devices to the current security model. As discussed above, the mobile server should not rely on Mobile Trusted-Origin Policy for client authentication (just as a web server should not rely on the Same-Origin Policy for client authentication), instead our system allows the server to identify and reject unwanted requests from compliant devices. As more devices deploy Mobile Trusted-Origin Policy, such unwanted traffic will decrease.

When a server starts to support Mobile Trusted-Origin Policy, it adds the mobile origins of its authorized apps to its whitelist. However, a special case arises when the server provides third-party libraries for apps. When these apps embed the libraries, requests from the libraries would be rejected by the server because the mobile origin of these libraries would be those third-party apps rather than the library itself. To handle this case, the server should release its code in a standalone app instead.

Fortunately, even low levels of adoption can be useful for reducing attacks like click fraud. Fraudsters will have two choices when devices start to append the *X-Mobile-Origin* headers): 1) they can detect when this occurs and not perform their fraud, or 2) they can perform their fraud regardless. In the former case, fraudsters will lose out on the revenue they would have gained from installs on MTOP-enabled devices. In the latter case, MTOP would append the mobile origin to the fraudulent app's HTTP requests, which may help servers identify fraudulent traffic (and which apps are generating it). This hurts the fraudsters regardless of the choice that they make.

7.2.3 Legacy Apps

Apps which do not contain network-enabled libraries do not need to be modified to work with our framework. Apps which do contain network-enabled libraries, on the other hand, would likely be incompatible with our framework, as traffic generated from the libraries would have the mobile origin of the parent app. Library developers would be incentivized to modify their library to be a standalone app in order to protect against unauthorized access to their web APIs, and app developers would be incentivized to upgrade their apps in order to be able to use the functionality of these libraries.

7.3 Performance

We evaluate the additional delay to network requests introduced by our proxy implementation. As our implementation is built on LayerCake, we inherit its performance characteristics. Roesner and Kohno [20] determine that LayerCake introduces minor overhead when UI embedding is used, and negligible overhead to input event dispatch delay.

Webpage Size	**Load time (ms)**					
	HTTP			**HTTPS**		
	w/o Proxy	**with Proxy**	**Overhead**	**w/o Proxy**	**with Proxy**	**Overhead**
Small (431B)	7	17	10	27	85	58
Large (580KB)	238	250	12	308	521	213

Table 2: Comparison of the average time required for an app to load a small webpage (averaged over loading 10,000 times) and a large webpage (averaged over loading 1,000 times) while using our proxy vs. not using our proxy.

Our proxy-based approach introduces overhead on HTTP traffic from apps as the traffic must pass through our proxy. To determine whether the overhead is manageable, we perform two experiments: loading a small webpage (431 B) 10,000 times and loading a large webpage (580KB) 1,000 times. For both experiments, the website is hosted on a nearby server and the phone (a Nexus 4) is connected to the Internet via USB. Table 2 summarizes the results. It took 168 seconds and 68 seconds to load the small website 10,000 times with and without the proxy, respectively, so the proxy introduced only 10 millisecond overhead in each page load. For loading the large webpage 1,000 times, it took 250 seconds and 238 seconds with and without the proxy, respectively, so the proxy introduces only 12 milliseconds overhead in each page load. Such overhead caused by our proxy is negligible and therefore should be unnoticeable to users.

We perform the same benchmark for our HTTPS proxy. Our HTTPS proxy introduces a significant inital latency (about 1.5 seconds) the first time when it makes a request to a host, as the proxy must fetch the server's certificate, extract its identify information, and create a new one. (By comparison, without our proxy, the app still needs to fetch the server's certificate but need not create a new one.) However, subsequent requests to the same host are much faster since they reuse the newly created certificate. It took 854 seconds and 273 seconds to load the small website 10,000 times with and without the proxy, respectively, resulting in an overhead per page load of 58.1 milliseconds. The proxy must establish two SSL connections each time an app connects using HTTPS, meaning the cost of fetching a small webpage is dominated by the cost of the SSL handshakes. For the large webpage, it took 521 seconds and 308 seconds to make 1,000 requests with and without the proxy, respectively. Fetching the larger webpage amortizes the cost of the handshakes over a longer connection, meaning that the larger webpage had a significantly lower overhead per byte compared to the smaller one.

8. DISCUSSION

8.1 Other Operating Systems

Mobile Trusted-Origin Policy could apply to other smartphone operating systems besides Android. An interesting case is Firefox OS. Firefox OS is a mobile phone operating system which distributes apps as HTML and Javascript, and enforces the web's Same-Origin Policy to prevent externally loaded Javascript from accessing device functionality or reading app data. Additionally, app code is restricted from communicating with arbitrary websites according to the restrictions of the web's Same-Origin Policy. Despite this, Firefox mobile apps may request privileges to circumvent this restriction (the `network-http` permission or `systemXHR` permission). In fact, this permission is the most commonly requested among apps on the Firefox mobile app market [6]. Given this, Firefox OS could still benefit from using the Trusted Cross-Origin Requests HTTP header to prevent apps from making unauthorized network requests. Firefox OS would be free to implement the remainder of the Mobile Trusted-Origin Policy using the existing Same-Origin Policy built into the OS.

8.2 Privacy

Our framework may create privacy concerns as it enables a network snooper to determine which apps a user has installed by observing the `X-Mobile-Origin` values in outbound HTTP requests. We note there are a number of ways network eavesdroppers may already infer what apps a user is running, for example by network profiling [4], observing publisher IDs in ad requests [23], or by simply observing HTTP user-agent strings of apps in iOS. Regardless, our system relies on a partial mitigation to reduce the privacy impact of the TCOR headers. Apps which require a TCOR header must opt-in to having them added by requesting a special permission which our system uses when modifying the proxied traffic. Apps which opt-in will have the `X-Mobile-Origin` header added as usual. Apps which do not will have an empty value for `X-Mobile-Origin` in outbound requests. What is important to note is that all apps are subject to having their traffic proxied, regardless of whether they opt-in, meaning the origin headers cannot be forged by apps which choose to opt-out.

8.3 Limitations

We use the app's verification key to create its mobile-origin signature. However, in Android, developers may sign many of their apps with the same key, meaning our system defines mobile origin at the developer level. An alternative design would be to use a combination of the verification key and the app's package name, but since two apps on different markets may have the same package name (as discussed in 5.1), this only provides per-developer, per-market granularity. Finally, the code signature itself could be used, but since apps may be updated, this would be too granular, causing servers to have to update their signature each time an app updates.

Our implementation only adds origin information for HTTP traffic on port 80 and HTTPS traffic on port 443. Thus, servers which wish to enforce origin-based policies should listen on one of these ports. Apps which communicate via other protocols will not have their origin included with outbound requests. To estimate how many apps connect with a protocol other than HTTP(S), we ran a random sample of 1,000 Android apps through PyAndrazzi [15], a dynamic analysis tool which performs UI state exploration. From the resulting network logs, we found that of the 793 apps with network connections to known ports, 622 apps (78%) only connect over HTTP, while 786 apps (99%) only use HTTP or HTTPS (based on destination port number)[7].

A limitation of our proxy for HTTPS connections is certificate pinning, where the client app specifies the exact certificate it expects the server to respond with, instead of relying on certificate authorities. Apps which use certificate pinning will get certificate errors from our proxy, which resigns the remote server's certificate so it can perform its man-in-the-middle. According to [9], approximately 10% of the top apps on Google play use certificate pinning. Breaking certificate pinning is a limitation of the prototype implementation rather than of MTOP. This necessitates further research into how to send the provenance information. For example, it could be added at a lower level protocol (e.g., as an IP option).

9. RELATED WORK

One of the goals of the Mobile Trusted-Origin Policy is to prevent a user's device from being turned into a "bot" without exploiting the device's operating system, simply because a malicious app was downloaded. Botnets are collections of compromised user machines (called bots) which can be instructed to generate fraudulent network traffic. The damage caused by botnets is well documented in previous literature. For example, botnets can be used to perform ad fraud [24], send spam emails [25], and perform distributed denial-of-service attacks [10].

9.1 Mobile Ad Fraud

There is limited prior work investigating mobile app ad fraud. Liu, et al. [18] investigate *display fraud* on Windows Phone, which uses techniques that analyze the UI of apps to determine if developers are obfuscating ads or coercing users to click ads by placing them near buttons and other actionable UI elements. Our proposed framework prevents this attack as the app cannot interfere with the ad app's UI. Crussell et al. [3] built a system, MAdFraud, to automatically extract ad requests and clicks from network traces to investigate apps performing *ad fraud*: apps making ad requests when in the background and issuing clicks without user interaction. Our work performs a detailed investigation of two click fraud app families and proposes a framework that prevents the ad fraud found by MAdFraud. In addition, Symantec [1] and Lookout [12], two security companies, provide case studies of malware which issues fraudulent clicks to search engines. These include a type of click fraud called *search engine poisoning*, where search engine results can be influenced by clicking on search engine links to increase their page rank. We note that despite similar nomenclature, this type of click fraud is independent of ad click fraud.

9.2 Inner-App Code Isolation

Grace, et. al. [14] and Stevens, et. al. [23] investigate the practice of Android ad libraries leveraging permissions of the parent app to exfiltrate sensitive user data. As a result of these infringements, there have been proposals for different ways to separate ad libraries and apps into different security domains on Android. LayerCake [21] and AFrame [27] achieve this by separating the app and ad library into separate processes, relying on the built-in Linux process isolation mechanisms to put them into separate security domains. This requires distributing the app and library as separate Android apps which share a UI, while the systems ensure that touch events cannot be forged between security domains. Moshchuk, et. al. [19] develop a content-based isolation mechanism for Windows called ServiceOS, in which the operating system is able to sandbox code based on where the code came from, like the browser's SOP. This allows for isolation beyond application granularity (for smartphones) or user granularity (for desktops). These systems can isolate library and app code, but do not provide a way to mediate apps' network access to prevent the attacks in Section 3.

9.3 App Provenance for Network Requests

AdSplit [22], on the other hand, is built on Quire [7], which adds call stack provenance to IPC and RPC calls. Quire's RPC attestations, while able to to protect against the attacks in Section 3, rely on the presence of a manufacturer-embedded private key on the device to sign the RPC call stacks. Another work, AdAttester [17], provides impression and click attestations for ad requests by leveraging ARM's TrustZone hardware, preventing apps from sending forged impressions and clicks. Like AdSplit, it requires a private key stored on the device to sign its attestations. Not only would compromise of this key allow for forged attestations to be made, but it is unclear whether servers would be able to enumerate all the certificates associated with these deployed keys, as each device and manufacturer would need to have its own. Our MTOP, on the other hand, does not require a secret to be embedded on the device, and servers would need to enumerate only signatures for authorized apps, instead of for all devices which may contact the server. We note that it is already common for servers to whitelist apps, e.g., in the form of publisher IDs for ad providers and username/-password credentials for web APIs.

10. CONCLUSION

We proposed the Mobile Trusted-Origin Policy, which consists of two parts: 1) an app provenance mechanism which annotates outgoing HTTP(S) requests with information about which app generated the network traffic, and 2) a code isolation mechanism that separates code within an app which should have different provenance signatures into what we call a "mobile origin". To handle point (1), we propose Trusted Cross-Origin Requests, which allows servers to decide if an app's communication is authorized by including app provenance in outgoing HTTP requests. To motivate such a system, we investigated two families of click fraud malware in Section 3.2 and observe how the lack of origin protections enables the fraud to occur. Finally, we implement MTOP on Android, using a network proxy to add the TCOR header to outgoing HTTP requests and HTTPS requests, evaluating its deployability and performance.

Acknowledgements This paper is based upon work supported by the Intel Science and Technology Center for Secure Computing and UC Davis RISE award.

[7]PyAndrazzi cannot log in to services, and thus may not expose all the network functionality of an app. For apps that perform log in over HTTPS, however, PyAndrazzi will still attempt to submit the form with dummy values, and thus we expect to see the outgoing HTTPS connection regardless, even if the log in fails.

References

[1] Eric Chien. *Motivations of Recent Android Malware.* Tech. rep. Technical Report, Symantec Security, 2013.

[2] Jonathan Crussell, Clint Gibler, and Hao Chen. "AnDarwin: Scalable Detection of Semantically Similar Android Applications". In: *Computer Security–ESORICS 2013.* Springer, 2013, pp. 182–199.

[3] Jonathan Crussell, Ryan Stevens, and Hao Chen. "MAdFraud: Investigating Ad Fraud in Android Applications". In: *Proceedings of 12th International Conference on Mobile Systems, Applications and Services.* 2014.

[4] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. "Networkprofiler: Towards automatic fingerprinting of android apps". In: *INFOCOM, 2013 Proceedings IEEE.* IEEE. 2013, pp. 809–817.

[5] Benjamin Davis and Hao Chen. "RetroSkeleton: Retrofitting Android Apps". In: *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.* ACM. 2013, pp. 181–192.

[6] Daniel DeFreez, Bhargava Shastry, Hao Chen, and Jean-Pierre Seifert. "A First Look at Firefox OS Security". In: *Workshop on Mobile Security Technologies.* 2014.

[7] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. "QUIRE: Lightweight Provenance for Smart Phone Operating Systems." In: *USENIX Security Symposium.* 2011.

[8] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.* Vol. 10. 2010, pp. 1–6.

[9] Sascha Fahl et al. "Why Eve and Mallory love Android: An analysis of Android SSL (in) security". In: *Proceedings of the 2012 ACM conference on Computer and communications security.* ACM. 2012, pp. 50–61.

[10] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. "Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks". In: *Computer Security–ESORICS 2005.* Vol. 3679. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 319–335.

[11] Sean Gallagher. "Snapchat images stolen from third-party Web app using hacked API". In: *Ars Technica* (Oct. 2014). URL: http://arstechnica.com/security/2014/10/snapchat-images-stolen-from-third-party-web-app-using-hacked-api/.

[12] John Gamble. *MaClickFraud: Counterfeit Clicks and Search Queries.* 2013. URL: https://blog.lookout.com/blog/2013/11/01/maclickfraud-counterfeit-clicks-and-search-queries/.

[13] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale". In: *Trust and Trustworthy Computing.* Springer, 2012, pp. 291–307.

[14] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza 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.

[15] Kristen Kennedy, Eric Gustafson, and Hao Chen. "Quantifying the Effects of Removing Permissions from Android Applications". In: *Workshop on Mobile Security Technologies.* 2013.

[16] Anne van Kesteren. "Cross-Origin Resoucrce Sharing". In: *World Wide Web Consortium W3C* (Jan. 2014). URL: http://www.w3.org/TR/cors/.

[17] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. "AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone". In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* ACM. 2015, pp. 75–88.

[18] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. "DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps". In: *Presented as part of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* Seattle, WA: USENIX, 2014.

[19] Alexander Moshchuk, Helen J Wang, and Yunxin Liu. "Content-based isolation: rethinking isolation policy design on client systems". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM. 2013, pp. 1167–1180.

[20] Franziska Roesner, James Fogarty, and Tadayoshi Kohno. "User Interface Toolkit Mechanisms for Securing Interface Elements". In: *Symposium on User Interface Software and Technology* (2012).

[21] Franziska Roesner and Tadayoshi Kohno. "Securing Embedded User Interfaces: Android and Beyond". In: *Proceedings of the 22nd USENIX Security Symposium* (2013).

[22] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating smartphone advertising from applications". In: *Proceedings of the 21st USENIX Security Symposium.* 2012.

[23] Ryan Stevens, Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "Investigating User Privacy in Android Ad Libraries". In: *Workshop on Mobile Security Technologies.* 2012.

[24] Brett Stone-Gross et al. "Understanding fraudulent activities in online ad exchanges". In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference.* ACM. 2011, pp. 279–294.

[25] Gianluca Stringhini, Oliver Hohlfeld, Christopher Kruegel, and Giovanni Vigna. "The Harvester, the Botmaster, and the Spammer: On the Relations Between the Different Actors in the Spam Landscape". In: *Proceedings of the 9th ACM Symposium on Information, Computer, and Communication Security.* ACM. 2014.

[26] Nicolas Viennot, Edward Garcia, and Jason 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.

[27] Xiao Zhang, Amit Ahlawat, and Wenliang Du. "AFrame: isolating advertisements from mobile applications in Android". In: *Proceedings of the 29th Annual Computer Security Applications Conference.* ACM. 2013, pp. 9–18.

APPENDIX

```
POST //NuclearCoreWeb//NuclearCoreServlet HTTP/1.1
Content-Type: text/plain; charset=ISO-8859-1
Host: core.nucleardroid.com:8080
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

{"namespace":"com.pixcel.MorzeDroid","request":"getAd"}
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Powered-By: Servlet/3.0; JBossAS-6
Date: Thu, 13 Feb 2014 23:04:34 GMT

{"countperday":5,"items":[{"countfake":5,"counttry":2,"name":"Madvertise-1","enable":true,"namespacelist":["com.
pixcel.MorzeDroid"],"siteid":"QCPWyzAF","publisherid":"","countsuccess":3,"adname":"Madvertise","namespace":"com
.pixcel.MorzeDroid"},{"countfake":5,"counttry":2,"name":"MobFox-1","enable":true,"namespacelist":["com.pixcel.Mo
rzeDroid"],"siteid":"84baac85e1f8a3092bd7cd4847dd0483","publisherid":"","countsuccess":1,"adname":"MobFox","name
space":"com.pixcel.MorzeDroid"},{"countfake":5,"counttry":2,"name":"Tapjoy-1","enable":true,"namespacelist":["co
m.pixcel.MorzeDroid"],"siteid":"c9e2a2f6-fb99-4131-bba0-944be954493d","publisherid":"IQAnj1JzUVHSFvRqja1R","coun
tsuccess":3,"adname":"Tapjoy","namespace":"com.pixcel.MorzeDroid"}],"enable":true,"namespace":"com.pixcel.MorzeD
roid"}
```

Figure 3: Example Nucleardroid query.

```java
Iterator localIterator = localJSONAd.getAdItems().iterator();
while (localIterator.hasNext()) {
  JSONAdItem localJSONAdItem = (JSONAdItem)localIterator.next();
  if (localJSONAdItem.isEnable()) {
    LogHelper.log("processingAd: ad item: " + localJSONAdItem.getName() + " processing");
    BaseGenerator localBaseGenerator = GeneratorFactory.getGenerator(getApplicationContext(), localJSONAdItem);
    if (localBaseGenerator != null)
      try {
        localBaseGenerator.fakeRequest();
        waiteOne();
        localBaseGenerator.clickRequest();
    } catch (Exception localException2) { }
  } else {
    LogHelper.log("processingAd: ad item: " + localJSONAdItem.getName() + " disabled");
  }
}
```

Figure 4: De-compiled Java for Pixcel's ad fraud service.

```javascript
setTimeout(function() {
  var i=1;
  var frames=document["body"]["getElementsByTagName"]("iframe");
  for(var j=0; j<frames["length"]; j++) {
    var content =
      frames[j]["contentWindow"] ? frames[j]["contentWindow"]["document"] : frames[j]["contentDocument"];
    if(content) {
      var atags= content["body"]["getElementsByTagName"]("a");
      if(atags["length"]) {
        _0x432cx7(atags[0]["href"], clickUrls[Math["min"](i, clickUrls["length"]-1)]);
        i++;
  } } }
  function _0x432cx7(href, clickUrl) {
    var div=document["createElement"]("div");
    div["innerHTML"]="<iframe src=\"javascript:‘<!doctype html><html><head><meta charset=\‘utf-8\’></head><body
        >’ + decodeURIComponent(‘" + encodeURIComponent("<img src=\""+href+"\"/>")+"’) + ‘</body></html>’\"></
        iframe>";
    div["style"]["width"]="1px"; div["style"]["height"]="1px";
    div["style"]["visibility"]="hidden";
    document["body"]["appendChild"](div["firstChild"]);
    if(clickUrl) {
      var img=document["createElement"]("img");
      img["setAttribute"]("src", clickUrl);
      img["style"]["width"]="1px"; img["style"]["height"]="1px";
      img["style"]["visibility"]="hidden";
      document["body"]["appendChild"](img)
} } }, 3000);
```

Figure 5: De-obfuscated Javascript from postupdate.info/carousel/ad_track.js**.**