# MAdFraud: Investigating Ad Fraud in Android Applications

Jonathan Crussell
UC Davis
jcrussell@ucdavis.edu

Ryan Stevens
UC Davis
rcstevens@ucdavis.edu

Hao Chen
UC Davis
chen@ucdavis.edu

## ABSTRACT

Many Android applications are distributed for free but are supported by advertisements. Ad libraries embedded in the app fetch content from the *ad provider* and display it on the app's user interface. The ad provider pays the developer for the ads displayed to the user and ads clicked by the user. A major threat to this ecosystem is *ad fraud*, where a miscreant's code fetches ads without displaying them to the user or "clicks" on ads automatically. Ad fraud has been extensively studied in the context of web advertising but has gone largely unstudied in the context of mobile advertising.

We take the first step to study mobile ad fraud perpetrated by Android apps. We identify two fraudulent ad behaviors in apps: 1) requesting ads while the app is in the background, and 2) clicking on ads without user interaction. Based on these observations, we developed an analysis tool, MAdFraud, which automatically runs many apps simultaneously in emulators to trigger and expose ad fraud. Since the formats of ad impressions and clicks vary widely between different ad providers, we develop a novel approach for automatically identifying ad impressions and clicks in three steps: building HTTP request trees, identifying ad request pages using machine learning, and detecting clicks in HTTP request trees using heuristics. We apply our methodology and tool to two datasets: 1) 130,339 apps crawled from 19 Android markets including Play and many third-party markets, and 2) 35,087 apps that likely contain malware provided by a security company. From analyzing these datasets, we find that about 30% of apps with ads make ad requests while in running in the background. In addition, we find 27 apps which generate clicks without user interaction. We find that the click fraud apps attempt to remain stealthy when fabricating ad traffic by only periodically sending clicks and changing which ad provider is being targeted between installations.


*MobiSys'14,* June 16–19, 2014, Bretton Woods, New Hampshire, USA.

http://dx.doi.org/10.1145/2594368.2594391.

## Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection; K.4.1 [**Computers and Society**]: Public Policy Issues—*Abuse and crime involving computers*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

## General Terms

Security, Measurement

## Keywords

Android; online advertising; click fraud; app testing; data mining; network traffic classification

## 1. INTRODUCTION

Online advertising is a financial pillar that supports both free Web content and services, and free mobile apps. Both web and mobile advertising use a similar infrastructure: the *ad library* embedded in the web page or mobile app fetches content from *ad providers* and displays it on the web page or the mobile app's user interface. The ad provider pays the developer for the ads displayed (*impressions*) and the ads clicked (*clicks*) by the user. Because web and mobile advertising use a similar infrastructure, they are subject to the same security concerns, such as tracking and privacy infringements [9, 27, 29]. Perhaps the biggest threat to the sustainability of this ecosystem is *ad fraud*, where a miscreant's code fetches ads without displaying them to the user or "clicks" on ads programmatically. Ad fraud has been extensively studied in the context of web advertising but has gone largely unstudied in the context of mobile advertising.

On the web, ad fraud is often perpetrated by botnets, which are collections of compromised user machines called bots. Fraudsters issue fabricated impressions and clicks using bots so that the traffic they generate is varied (i.e., by IP address), making the fraud harder to detect [28]. The two properties of botnets that make them useful for ad fraud are:

- Botnets are comprised bots, which are real user devices.
- Bots can be instructed to make arbitrary network requests silently to the user.

Both of these properties also apply to malicious Android applications. Moreover, while attackers have to compromise desktops to acquire bots, they can simply distribute their malicious applications through Android markets. Based on

these observations, we hypothesize that Android (and mobile devices in general) are a lucrative target for those who commit ad fraud professionally.

We take the first step to study fraud and other undesirable behavior in mobile advertising. First, we identify unique characteristics of mobile ad fraud. On Android, at any time at most one app is running in the foreground, where the app has a UI. Our first observation is that when an app fetches ads while it is in the background, this is most likely fraudulent, because the app developer gets credit for this ad impression without displaying it to the user.[1] Our second observation is that when an app *clicks* an ad without user interaction, it is definitely fraudulent.

Based on our observations, we set out to measure the prevalence of ad fraud in the wild. We use two sets of apps: 1) 130,339 apps crawled from 19 Android markets including Play and many third-party markets, and 2) 35,087 apps that likely contain malware provided by a security company. We build a testing infrastructure, where we launch multiple instances of the Android emulator concurrently. In each emulator, we install an app from our datasets, run it for a fixed time, push it to the background, and continue running for a fixed time, while capturing all the network traffic from the emulator. Finally, we extract impressions, clicks, and other ad related activities from the network traffic.

We have to overcome a number of challenges, the biggest of which is how to identify ad impressions and clicks. The formats of impressions and clicks vary widely between different ad providers, so manually compiling a white list is laborious, unreliable, and prone to obsolescence. We propose a novel approach with three steps. First, we build *request trees* to link causally related HTTP requests (Section 4.2). Then, we use machine learning to discover new ad request pages based on known ones. Finally, based on the properties of ad clicks, we create rules to identify clicks from the HTTP request trees. From analyzing our datasets, we find that about 30% of apps with ads make ad requests while in running in the background, which may be malicious or indicate misconfiguration. In addition, 27 apps generate clicks without user interaction, which is definitely malicious. We find that the click fraud apps attempt to remain stealthy when fabricating ad traffic by only periodically sending clicks and changing which ad provider is being targeted between installations.

We call our system MAdFraud, which includes the testing infrastructure we used to run apps as well as the techniques used to identify impressions and clicks. The goal of MAdFraud is to automatically detect fraud behavior through black-box testing of apps alone, without the need to manually reverse engineer apps. In this way, our system can scale to large numbers of apps. Although the definition of ad fraud varies between ad providers as some ad providers only pay developers when ads are clicked, we define fraud as: 1) apps that show ads in the background, and 2) apps that click on ads without user interaction. Thus, MAdFraud exposes potentially fraudulent behaviors in apps but it is up to the ad providers to decide if the behavior is indeed fraudulent. From reviewing the terms and conditions of several ad providers [1, 19, 20], we find that all explicitly disallow (2) while only one explicitly disallows (1). It would be fairly trivial to enforce other definitions of fraud by analyzing the logs of impressions and clicks produced by MAdFraud. For example, the ad providers may restrict which other ad providers an app may contact. MAdFraud can be applied to other deployments, such as identifying ad behavior in network traffic captured from the wild, simply by re-training our machine learner on the new dataset (see Section 7). We make the following contributions:

- We take the first step to study ad fraud in mobile advertising by running a large number of apps and analyzing their network traffic.
- We propose a novel approach for automatically identifying ad impressions and clicks, which is the basis for detecting ad fraud.
- We discovered and analyzed various fraudulent ad behavior in mobile apps on popular markets.

# 2. BACKGROUND

## 2.1 Web Advertising

Advertising on the Internet is pervasive, and allows for services such as websites, search, and email to be provided to customers for free by including advertisements (ads) as part of the content displayed to the user. Website owners and other service providers (called *publishers* in advertising jargon) typically include ads through a third party called an *ad provider*, which handles finding and selecting advertisements, as well as paying publishers for ads shown to their users. On the Web, this is typically implemented as an `<iframe>` or `<script>` HTML element embedded in the publisher's webpage, with a `src` attribute that points to the ad provider's ad server. When the web page is loaded by a browser, the ad is populated via an *ad request*, which contains the publisher's ID and information about the user that is used to select a relevant ad (known as targeting information). The ad server returns three pieces of content once an ad is selected: the ad content URL, a click URL, and a pixel URL. The ad content is typically hosted by the ad provider (usually through a CDN) instead of the digital marketer who owns the ad, ensuring the content will be available when the ad is loaded. Marketers who are paying for their ads to be distributed by the ad provider want to guarantee the ad provider is not fraudulently billing them, so they themselves host a *tracking pixel* (or web bug) that is loaded by browsers along with the ad so that the marketers can independently verify that ads are being requested. Finally, the click URL indicates which web page should be opened when a user clicks on an ad. The click URL typically points to the ad provider's ad server, which records the clicks and then redirects the user to the marketer's landing page. A complete ad request, response, and display of the ad and pixel to the user is called an *impression*, and opening the click URL is a *click*. Publishers are paid based on how many impressions and clicks their content generates.

## 2.2 Web Ad Fraud

Unscrupulous publishers may inflate their ad revenues by having automated bots visit their website and click on ads.

[1]One possibly legitimate exception is when the app tries to cache ad content while it is running in the background, but this is forbidden by the terms of service of many ad providers, as the app may never return to the foreground and therefore cannot guarantee that the cached content will ever be displayed on the UI.

This is referred to as ad fraud (or click fraud), and is a serious security issue as digital marketers who pay to have their ads shown online will not receive any business benefit for ads shown to bots. Although hard numbers on the amount of ad fraud is hard to determine, conservative estimates suggest 10% of Web ad traffic is due to fraud [11]. In order to receive revenue, fraudsters must remain undetected while issuing large numbers of ad requests and clicks. To do so, they employ a number of techniques. First, the ratio of click requests to requested ads is kept low (around 1% [28]) to avoid suspicion, as ads are rarely clicked on by real users. This means fraudsters issue far more ad requests than click requests. Second, fraudsters do not rely on a single publisher account, but rather have many accounts from many ad providers which they rotate through while issuing requests [28]. Not only does this mitigate the impact of any single account being detected, it also decreases the magnitude of fraudulent requests for each publisher ID and ad provider. Finally, fraudsters use botnets as the bots run code that consistently visits the fraudsters' webpages in the background and clicks on the ads located there, so that the fraudsters receive revenue. As mentioned in Section 1, botnets allow fraudsters to remain stealthy as the bots are real user devices which have been compromised.

### 2.3 Android App Advertising

Many Android applications are distributed for free on app markets, and use ads embedded in the user interface of the app to make money for the developer. The developer must register with an Android ad provider, which provides the developer with a publisher ID and an ad library to include in their app. The library is responsible for fetching and displaying ads when the app is being run. Requesting an ad for an app is analogous to doing so on the web: an ad request is made over HTTP to the ad server which includes the developer's publisher ID and user targeting information. The ad server returns the ad's content URL, click URL, and any tracking pixel URLs which must be fetched to display the ad. In fact, many ad libraries choose to implement making requests and displaying ads simply by loading a traditional HTML ad element in a web view. The primary difference between web and Android app advertising is that ad libraries are implemented in application code, and often contain special application-only logic, for example automatically collecting user targeting information or refreshing the ad.

## 3. DATASET

### 3.1 Crawled Apps

We evaluate our assumptions on 713,173 Android apps crawled from 19 markets including Google Play and numerous third-party markets. Since we are only interested in apps that can make network requests, we randomly selected 150,000 apps from the 669,591 apps that have the Internet permission. Due to incompatibilities with our analysis environment (e.g. mismatched API version and missing dependencies), we ran a total of 130,339 apps out of the 150,000 selected apps (see Section 5.1 for details on the failed apps). These apps were crawled between between October 30, 2012 and July 17, 2013. A breakdown of the 130,339 apps by market is shown in Table 1.

| Market | Apps |
|---|---|
| Google Play | 83,957 |
| Anzhi | 14,273 |
| Gfan | 8,423 |
| Brothersoft | 7,037 |
| Opera | 6,131 |
| SlideME | 4,627 |
| m360 | 3,096 |
| Android Online | 2,174 |
| 1Mobile | 1,650 |
| Eoemarket | 1,128 |
| Goapk | 628 |
| AndAppStore | 283 |
| ProAndroid | 278 |
| Freeware Lovers | 233 |
| SoftPortal | 216 |
| Androidsoft | 136 |
| AppChina | 74 |
| AndroidDownloadz | 39 |
| PocketGear | 36 |

Table 1: Market origins of the apps successfully run by MAdFraud. Since some apps appear on multiple markets, the total apps in the table is slightly more than the total 130,339 apps run.

### 3.2 Malware Apps

In addition to the dataset described above, we received 38,126 apps from a mobile security company between October 28, 2013 and November 16, 2013. According to the company, the dataset contains apps which are likely malware, but may contain some goodware as well. Regardless, the percentage of apps which are malware in this dataset is expected to be higher than in our crawled apps. We speculate the click fraud will be more pervasive in this dataset as ad fraud on the web is primarily conducted by bots.

### 3.3 Ad Provider Domains

In order to detect ad behavior, we must know which domains that ad providers use for their ad requests and clicks. We build our list using two domain blacklists [14, 25] which are curated to include only domains related to ad providers and analytics. We then manually removed domains related to the most popular analytics libraries. Unfortunately, even after pulling in two separate blacklists with one being specifically curated to include mobile ad providers, our list was not complete. Therefore, we supplement these lists with ad-related domains that we manually identified using AppBrain's list of popular ad libraries [2]. In total, we identified 3,118 ad-related domains, 3,062 from the blacklists and 56 manually.

## 4. METHODOLOGY

We wish to measure the prevalence of fraudulent and undesirable ad behavior (abbreviated simply as *ad fraud* henceforth) in Android applications. Based on common guidelines and terms of service in mobile advertising, we identify the following behavior as ad fraud:

- An Android app requesting ads while in the background, as it cannot display ads to the user (see a possible exception to this rule discussed in Section 1).

- An Android app generating clicks without user interaction.

To detect the above behavior, we design a dynamic analysis system to run apps in an emulator, capture their network traffic, and extract ad impressions and clicks from the traffic.

## 4.1 Running Android Apps

We run apps in our dataset in an Android emulator (API 17) and capture their network traffic. For each app, we create a new emulator image, install the app on the new emulator, run the app in the foreground for 60 seconds, put the app into the background, and run for another 60 seconds. To put the app in the background, we issue an intent to the emulator to open the browser on a static page hosted on our server. The HTTP request to this static page marks the boundary between the app's foreground and background activities in the captured network traffic. As we installed only one third-party app on the emulator, all the captured traffic not from the emulator (evident by a nonstandard TCP port) or browser (evident by our server's IP address in the IP header's source or destination) is attributed to this app. We choose not to interact with the app (i.e., touch events) even when it runs in the foreground to ensure that any ad clicks are generated without user interaction.

To analyze the packets captured by MAdFraud, we use the Bro Network Security Monitor [22]. Using Bro, we can reconstruct TCP flows and extract application protocol entities for HTTP and DNS traffic. For HTTP, these entities include fields from the HTTP request and response. From the HTTP request, we extract header fields as well as the URL and request body. From the HTTP response, we record the status code, response type, and any URLs in the unzipped response body. For DNS, these entities include the hostname in the DNS query and the IP addresses in the DNS response. We store all of these entities in a database for further analysis.

## 4.2 Request Trees

We can group HTTP requests logically together, as a response from the server may trigger additional HTTP requests. For example, a browser loading a web page may fetch static resources, such as JavaScript or images, to embed in the HTML. In this case, we can group the HTTP requests using the HTTP `referrer` header to form a tree of requests where the request to the HTML page is the root and requests to the static resources are the children. Intelligently constructing this HTTP *Request Tree* is important because it enables a number of techniques for automatically analyzing ad traffic, such as automatically detecting clicks.

We represent each HTTP request in an app's network traffic by a node in a request tree, and connect two nodes if and only if they are related according to three rules. The first two rules are based on the HTTP protocol specifications [3]: 1) the client may set the request `referrer` field to indicate to the server the URL that contained the requested URL, so we consider the former URL as the parent of the latter URL, and 2) the server may set the `location` header along with a redirection status code to redirect the client to another URL, so we consider the original URL as the parent of the redirected URL. Finally, to account for when the referrer header is missing, we extract all the URLs in the HTTP response body of a node and consider the node as the parent of all the URLs (after URL normalization).

## 4.3 Ad Request Page Classifier

To identify ad fraud, we must first identify impressions in app network traffic, which begin with an ad request. Since there are many ad providers for Android (over 80 according to [2], and likely many more that are small or operate in other countries), it would be prohibitive to manually reverse engineer each ad provider's ad serving protocol to determine which URLs correspond with ad requests. Instead, we develop an approach for automatically identifying impressions using machine learning. From manually examining mobile ads in our previous work [27], we know ad requests have a common, characteristic format. They typically contain a large number of query parameters, such as the publisher ID and user and device information for selecting relevant ads, will receive at least three URLs in the response body, and are followed by HTTP requests to image content but not HTML or CSS. One naive approach would be to classify whether each HTTP request is an ad request directly; however, since this approach fails to consider any aggregate features, it would severely limit the features available to the classifier. Instead, we classify *request pages*, identified by the host and path names, which is the portion of URL before the '?' character that denotes the beginning of the query parameters. This allows us to extract features over the aggregate of all requests to each request page. We then classify whether each page is for requesting ads.

For each page we extract 33 features from three sources: 10 from query parameters, 16 from request trees, and 7 from HTTP headers. Since our classifier builds decision trees based on the predictiveness of features for the ground truth dataset, we include all the features that may be important based on our observations and then let the classifier determine which of these features are actually predictive. We measure the relative importance of these features in Section 5.2.2.

### *Features from query parameters.*

For each page, we aggregate all the requests to it to measure two properties on each query parameter: whether the parameter is an enumeration and how many bits of entropy it contains. To determine whether a parameter is an enumeration, we compute the ratio of distinct values found for that parameter over the total number of times the parameter appeared in a request. Some parameters will only have a small number of distinct values, and we expect this ratio to be tiny (for example, the IMEI of the emulator will always be the same). On the other hand, if the ratio is near one, it implies that most requests have different values for this parameter, indicating that the parameter may take arbitrary values (for example, a timestamp field). We also compute the ratio of distinct values found for that parameter over the number of distinct apps (to account for fields that remain unchanged between requests for the same app, such as the publisher ID). We then segment these two ratios into several intervals and count the numbers of query parameters whose ratios are in each interval. These counts contribute six features to our classifier. To estimate the entropy of the values of a key, we first attempt to categorize the values into classes (e.g. numeric, alphanumeric) which have different entropy levels per character. We then multiply the determined entropy level by the average length of the values associated with each query parameter to determine the entropy level for the query parameter. We consider an entropy to be high

if it has more than $2^{16}$ bits and low otherwise. We then count the number of query parameters that have high and low entropy, respectively, contributing two features to our classifier. The last two features describe the average and total number of query parameters, respectively.

*Features from request trees.*

These 16 features describe the structure of the request trees (Section 4.2). These features include the position in the request trees, such as the average height of trees containing the page, the average subtree height below the page, and the average depth of the page in the trees. We also extract features based on the number of children, the MIME types of the children, and the types of edges that connect the children to the parent.

*Features from HTTP headers.*

These 7 features include the status codes, the length of the requests, and the length of the replies, etc.

## 4.4 Finding Impressions and Clicks

With the classified ad request pages (Section 4.3) and HTTP request trees (Section 4.2), we can extract impressions and clicks for each app. It is tempting to simply consider each ad request as an impression. However, we found that many ad providers support *reselling* of ad impressions, where an ad request receives a new ad tag (i.e., an ad `<iframe>`) instead of ad content. This new ad tag then initiates another ad request, which may be resold again. Reselling of ad slots is done for business purposes: an ad provider may choose to buy ad slots from another ad provider so that the ad slots can be sold at a higher price to their own marketers, allowing for the provider to reach a larger target audience. For the purpose of measuring ad fraud, however, counting each resold ad request as an impression would be disingenuous, as the developer cannot control ad reselling and is paid only once regardless of how many times the ad is resold. To accurately measure the number of impressions, we consider an ad request as an impression only when none of its ancestors in the request tree are ad requests.

Besides impression fraud, another, perhaps more lucrative, revenue source for misbehaving apps is click fraud, where an app fabricates fraudulent clicks. There are two ways for apps to fabricate clicks without user interaction. First, the app may generate a touch event on the ad to trick the ad library to process it as a user click. Second, the app may parse the response body of the ad request and extract the click URL, and then make an HTTP request to the click URL. To detect either of these cases, we apply rules to the subtrees of ad request nodes in our request trees to determine if there is a path from an ad request node to a click node. We extracted the following rules by manually running apps in an emulator, clicking on their ads, analyzing their network traffic, and examining the properties of paths that led to ad clicks.

- Apps often handle clicks using HTTP redirection to send a request to the ad server before redirecting the user to the marketer's web page. Based on this observation, we look for an HTTP redirection in the subtree of an ad request node that ends on a node that received HTML as its response MIME type.
- Android apps have a variety of unique ways to handle clicks. For example, some clicks redirect to a `location` URL that has a `market://` schema, indicating the Google Play app should be launched. Similarly, we found an ad provider whose library downloads a `.apk` to the device after the user clicks an ad. To handle these cases as well as marketers that have HTTPS landing pages, we infer that an ad request that has a redirection in its subtree that redirects to a page with a schema other than `http://` contains a click.

Both of these rules require that the landing page be for a non-ad related hostname. We identify clicks in each app by building its request trees, finding all the impression nodes, and applying the above rules to each impression node. Figure 1 shows an example request tree with a click.

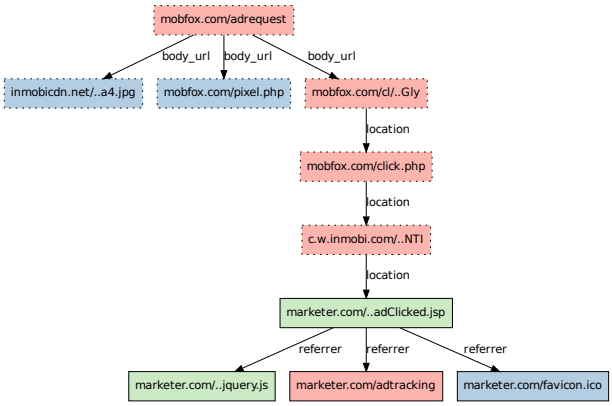


Figure 1: Example ad request tree with click. Nodes in blue are images and nodes in green are static web content. Nodes with a dotted outline are for requests with a known ad provider hostname. The ad request is the root, and its children are the ad image and a tracking pixel. The subtree on the right is the result of a click, with a redirect chain that ends on the marketer's homepage. All nodes have shortened URLs, for simplicity.

# 5. EVALUATION

We ran 130,339 of our crawled apps through MAdFraud. In total, these apps generated 1,807,379 HTTP requests, 508,005 of which we identify as being ad-related based on the hostname extracted from the request.

## 5.1 Apps without ad requests

77,461 apps made no HTTP request to a known ad provider hostname during the full duration of their runtime in MAdFraud. The possible reasons are: 1) the app crashes before it makes an ad request, 2) the app includes no ads, 3) the app requires user interaction to reach a UI state where ads are displayed, or 4) the ad library refuses to make ad requests when it detects the emulator environment.[2] To investigate why these apps make no ad requests, we ran a random sample of 7,500 (approx. 10%) apps through a separate tool called PyAndrazzi [12].

[2]We know of no ad libraries that do this.

| Status | Apps |
|---|---|
| Ran to completion, no ads | 3,807 |
| Ran to completion, with ads | 957 |
| Missing Native Library | 990 |
| Unknown Error | 733 |
| App crash | 573 |
| API Version Mismatch | 440 |

Table 2: Breakdown of PyAndrazzi's results for the 7,500 apps analyzed that made no ad requests when run through MAdFraud.

PyAndrazzi is a dynamic analysis framework designed to interact with apps and perform UI state exploration. Because it was originally designed to inspect how apps fail when their permissions are revoked, PyAndrazzi monitors apps extensively and can identify when and how the apps crash. This makes PyAndrazzi an ideal tool for determining whether apps are crashing, contain no ad libraries, or require interaction to reach a UI state where ads are displayed.

Table 2 shows that about 37% of the apps that did not generate ad traffic either crashed or failed to install properly on the emulator (due to missing x86 native libraries or an API mismatch). Both MAdFraud and PyAndrazzi use the x86 version of Android to speed up the Android emulator through hardware acceleration. However, some apps include native code for ARM devices but not x86 and thus cannot run correctly. In future work, we could address this limitation by running apps with ARM binaries on an ARM emulator. Among the apps that did run to completion, 957 made requests to a known ad provider. As these apps did not make ad requests during their analysis in MAdFraud, we conclude that these apps likely display ads on UI states that require user interaction.

## 5.2 Ad Request Page Classifier

### 5.2.1 Ground Truth

To build the ground truth dataset for identifying ad requests, we started with the most popular ad providers and manually investigated the pages that our apps requested. For each ad provider, if we were able to determine all the ad request pages for this ad provider by manually inspecting the HTTP traffic for each page, we labeled those pages as *ARQ* (ad requests) and all other pages for the ad provider as *NARQ* (not ad requests). Otherwise, we excluded this ad provider from our ground truth dataset and instead use our trained classifier to automatically identify the ad request pages for this ad provider. We also identified the top domains used by apps that are not ad providers and labeled all their pages as *NARQ*. In total, we identified 39 pages as *ARQ* from 25 ad providers and 11,484 pages as *NARQ*.

Since the dataset is heavily skewed towards the *NARQ* class, we use SMOTE [4] to over-sample the *ARQ* class. SMOTE is an algorithm for creating synthetic instances of a minority class to improve the classifier's sensitivity to the minority class. Once we have generated the new instances and added them to the ground truth, we then build a classifier using the `RandomForestClassifier` from scikit-learn [23]. The `RandomForestClassifier` creates a forest of decision trees by randomly selecting subsets of the feature space and training a decision tree using each subset.

| | | Prediction | | |
|---|---|---|---|---|
| | | *ARQ* | *NARQ* | Recall |
| Truth | *ARQ* | 28 | 11 | 71.8% |
| | *NARQ* | 9 | 11,475 | 99.9% |
| | Precision | 75.7% | 99.9% | |

Table 3: Confusion matrix of our ad request page classifier, computed using 3 fold cross-validation on our ground truth.

The most predictive decision trees are weighted appropriately, and in turn indicate which features are the most predictive. Decision trees are an appropriate choice for our feature space as we include categorical (i.e., non-Euclidean) features, which popular algorithms like SVM and K-neighbors do poorly on.

To evaluate the performance of the classifier, we apply cross validation to the ground truth. We split the ground truth into 3 folds, train a classifier on 2 of those folds and then evaluate its performance on the remaining fold. In the evaluation, we include no synthetic data points generated by SMOTE and used for training. Table 3 shows the overall confusion matrix for the classifier during cross validation. It shows that the classifier has a very low false positive rate (00.1%) and that it achieves a reasonable true positive rate (71.8%). Since we will apply the classifier to unknown pages from ad providers to report on the number of ad requests, we prefer the classifier to have erred towards being more precise. Overall, the classifier had a class-weighted accuracy of 85.9%.

### 5.2.2 Feature Importance

Once we have trained the classifier, we can evaluate the relative importance of its features to investigate which features are the most predictive. We compute feature importance using scikit-learn's built-in method, which measures feature importance by calculating the depth of nodes that use the feature in the decision trees. The higher a feature appears in the trees, the more important it tends to be. Using this methodology, each feature is assigned an importance between 0 and 100% with the sum of all feature importance values equaling 100%.

Figure 2 shows the importance values of features excluding features whose importance values are less than 1%. It shows that nine out of the top ten features are derived from the query parameters, such as the number of enumeration parameters, the number of low and high entropy parameters, and the total number of parameters. Note that many of these features could not be computed from single requests, confirming our assertion that classifying request pages, which are aggregate requests, is more predictive than classifying individual requests (Section 4.3). The most predictive request tree features focus on the height of the tree, as opposed to features measured by analyzing the page's child nodes. This makes sense, as ad requests tend to vary more in their tree heights and their depths in the tree (due to ad reselling) than other types of pages. The fact that features measured from child nodes are not very predictive is likely due to differences between how ad servers respond to ad requests.

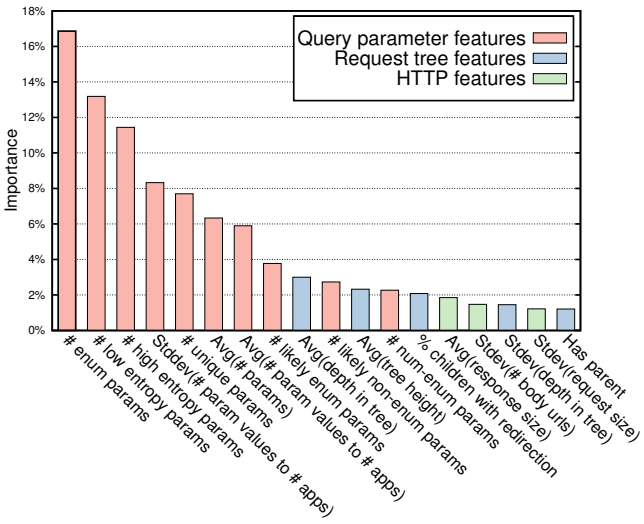


Figure 2: Feature importances ranking the predictability of each feature. Feature importances sum to 100%. Features with less than 1% importance are not shown.

### 5.2.3 Results

Using the ground truth described in Section 5.2.1, we build a classifier to apply to all the pages requested by apps in our dataset. First, as a prefiltering step, we eliminate all pages whose extension indicate that they are likely a static resource such as an image, JavaScript file, or CSS file, as static resources cannot be used for ad requests. After prefiltering, there are 100,611 pages to classify. When we applied the classifier to these pages, 715 pages were classified as *ARQs*, among them 678 pages were not present in the ground truth and 229 pages were for hostnames that had been known to be ad-related domains. As there are far too many non-ad-related domains to manually investigate, we instead investigated 30 domains whose pages the classifier was most confident were *ARQs* (the classifier computes a probability that the page is in each class). From the 30 domains, we were able to determine that 13 of the domains were ad-related. A common trend we found when analyzing these non-ad-related domains is that the pages used by analytics libraries, which are used to record information about the user's actions, are often misclassified as ad requests. These analytics pages have a very similar format to ad requests as they both rely on many query parameters to pass identifiers for the app, device, and user back to the server. This corroborates our earlier findings in Section 5.2.2 that the classifier primarily uses features based on the query parameters. To avoid over-estimating the number of ad requests performed by apps, we choose to consider only ad requests to pages with known ad-related hostnames. This results in 229 pages for 77 ad providers.

## 5.3 Request Trees

We build request trees for finding clicks based on impression nodes. Using the rules described in Section 4.4, we automatically found 60 clicks, among which 59 clicks we manually verified. The single false positive was caused by an ad image redirecting to an HTML page. We suspect that this is due to a misconfiguration in either the ad provider or the CDN hosting the ad image.

| Apps... | Crawled Apps | Malware Apps |
|---|---|---|
| *in dataset* | 150,000 | 38,126 |
| *successfully run* | 130,339 | 35,087 |
| *with HTTP traffic* | 88,305 | 24,957 |
| *with ad traffic* | 52,878 | 10,960 |
| *with impressions* | 40,409 | 8,974 |
| *with bg impressions* | 12,421 | 1,835 |
| *with clicks* | 21 | 6 |

Table 4: The number of apps at each stage of our analysis in our two datasets. *in dataset*: the number of apps examined. *successfully run*: the number that actually ran in our dynamic analysis environment. *with HTTP traffic*: the number of apps that had any HTTP traffic (not attributed to the emulator or MAdFraud). *with ad traffic*: the number of apps that have traffic to known ad provider hostnames. *with impressions*: the number of apps with impressions is calculated based on the ARQ pages identified by our classifier. *with bg impressions*: the number of apps making ad requests in the background. *with clicks*: the number of apps issuing clicks.

Occasionally, we cannot link clicks to their impressions due to ad providers that construct click URLs dynamically in application or JavaScript code. To identify clicks for these ad providers, we would have to either reverse engineer this logic to add special cases for them when constructing request trees, or manually determine which pages are associated with clicks and look for requests to these pages in app traffic. This limitation may have caused us to underestimate the number of clicks. In the case that an impression is resold, we find that the subsequent ad requests are always in the response body of their predecessors, as this is intrinsic to the mechanism for reselling ads. One caveat, however, is that it would be impossible to manually determine that two ad requests are part of a reselling chain if there is no referrer, body URL, or redirection to indicate that an impression was resold, without reverse engineering ad library logic. We speculate that this case would occur very rarely, if at all.

# 6. FINDINGS

We describe the results of our analysis on the network data captured from the crawled app and malware app datasets. We start by focusing on the 130,339 crawled apps, then in Section 6.3 we compare these results with the results of the 35,087 malware apps. Table 4 shows an overview of our results.

## 6.1 Market Comparison

We investigate the popularity of various ad providers in our crawled app dataset compared with the markets from which we downloaded the apps. We group the providers Admob and Doubleclick together under 'Admob', as they are both owned by Google and often appear together in ad traffic. Overwhelmingly, Admob is the most popular provider among the apps we ran; 47.21% of our apps make a network request to an Admob domain. Figure 3 shows the popularity of Admob across the 9 markets from which we downloaded a majority of our apps. The popularity of the remaining 8 most popular ad providers across these markets is shown in Figure 4. We can see that the Chinese providers WAPS and YouMi have much higher popularity among apps

from Chinese markets (Anzhi, Gfan, m360, and Android Online) compared with their popularity on English (language) markets. Similarly, English ad providers are less popular on the Chinese markets, although Admob is still the most popular provider on m360 and Android Online. Finally, the ad provider Adlantis is the only popular ad provider which is neither English or Chinese; Adlantis is a Japanese ad provider. Its popularity in our dataset is surprising, as we do not crawl any Japanese markets.

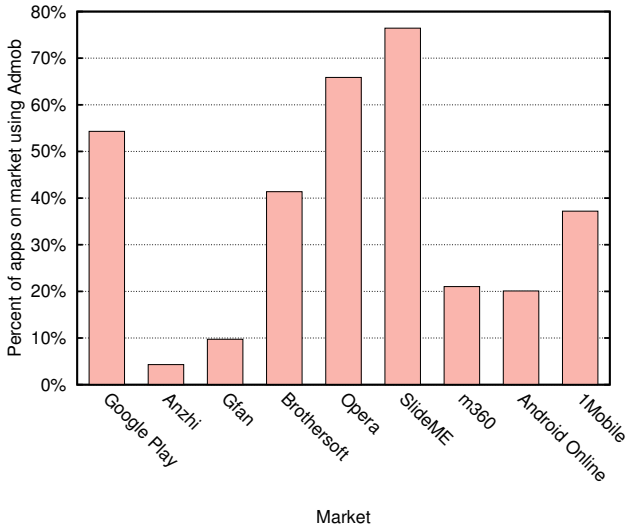


Figure 3: Popularity of Admob for apps from the top 9 Android markets, ordered by number of apps tested from that market. The y-axis is percentage of our crawled app dataset that appear on that market that make a request to an Admob domain.

## 6.2 Ad Fraud

Here we report on the ad behavior of our crawled app dataset. As previously mentioned, we identify two behaviors that are consistent with ad fraud: 1) apps which make ad requests in the background, and 2) apps that click on ads without user interaction.

### 6.2.1 Background Requests

We start by investigating which apps made ad requests while in the background. As previously mentioned, we run apps for 60 seconds in the foreground and 60 seconds in the background. We build request trees from the network captures of the apps and find impressions by looking for classified ad request pages in the request trees. Using the list of ARQ pages identified by our classifier, we find that 40,409 of our crawled apps generated a total of 274,128 impressions. Figure 5 shows a distribution of the number of impressions over time, relative to the start time of each app. From this figure, we can see a number of interesting patterns. First, most apps request an ad immediately upon startup without user interaction. 16,755 of the apps only request one ad during their run. Second, there is a clear periodicity to the distribution of impressions. This is because many Android ad libraries are configured to automatically refresh ads at constant intervals, the most common intervals being 30 seconds and 60 seconds. There appears to be a large spike at 60 seconds (the time when we put apps in the background) due to apps which are configured to request ads at these intervals. This is because these apps had initiated their ad request while in the foreground, but did not complete it until being put into the background. We do not wish to consider these impressions as fraudulent, as they do not indicate that the apps would have had the same behavior if they had been put into the background at a different time. Thus, we allow a 5-second grace period after apps are put into the background where we do not consider impressions as fraudulent. Based on this, we found 91,784 background impressions from 12,421 apps.

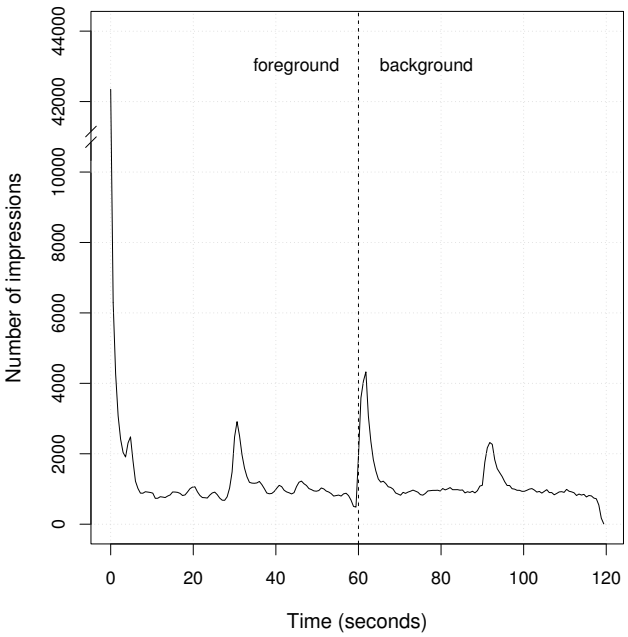


Figure 5: Distribution of the number of impressions over time, relative to the start time of each app.

Finally, the periodicity of ad requests continues after apps have been put into the background, implying some apps continue to run the ad library after losing focus. We expect that some of these cases are due to misconfiguration, however we do not attempt to determine intent. Regardless, requesting ads while the user is not using the app is undesirable behavior, as it is wasteful of device bandwidth and battery life, and may affect the ad provider's bookkeeping.

### 6.2.2 Click Fraud

We can now use the 274,128 request trees containing impressions that were found in the previous section to find apps that are fabricating clicks. We consider all clicks in our apps as fabricated, regardless of whether they were performed while the app was in the foreground or background, as we do not interact with the UI of our apps while running them. Using the click rules described in Section 4.4, we find that 21 apps fabricate a total of 59 clicks. We find that 24 of the clicks were performed in the foreground and 35 were performed in the background, indicating that the apps fabricating clicks continue doing so regardless of whether they are on the screen. We manually investigate the HTTP traffic for these 21 apps to confirm that they were all indeed

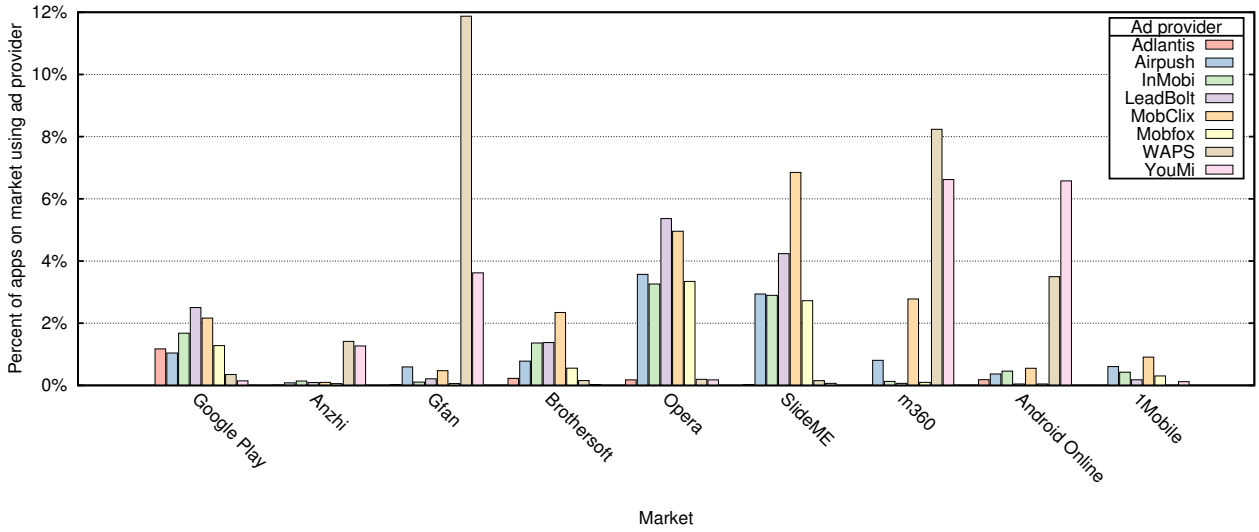


Figure 4: Popularity of the top 8 ad providers on the top 9 Android markets by percentage of apps. We omit Admob from the list of ad providers (but plot it in a separate figure, Figure 3) because of its overwhelming popularity.

click requests. This means that this result is a lower bound for the actual number of clicks across all ad providers. To further investigate the behaviors of these apps, we ran an additional experiment where we left the apps running for a total of twelve hours — six hours in the foreground and six hours in the background. The primary goal of this experiment was to determine whether the apps fabricate clicks for some period of time and then stop, or, if they continue to fabricate clicks on a regular interval indefinitely.

The results from this experiment are shown in Table 5 (along with six apps found to fabricate clicks in the malware dataset). Interestingly, only 16 of the 21 apps fabricated clicks when run a second time, perhaps to stay hidden. For each app, we show its source (either a market, the malware dataset, or '?' if the market information was missing), the number of installs the app had at the time it was crawled along with the number of impressions and clicks during the experiment. Lastly, we report the average interval of time between clicks, calculated as time between the first and last clicks divided by the total number of clicks, and list the ad providers whose impressions were clicked on.

There are a number of surprising results in this table. First, three apps on Google Play, collectively with thousands of installs, fabricated clicks. When we looked up the apps on the market, we found that only one of the three apps were still available. It is unclear whether these apps were removed due to the fraudulent ad behavior or for other reasons. Second, a number of apps have a very similar number of impressions and clicks. We speculate that the same miscreant may have uploaded separate apps with the same click fraud code. Third, only three ad providers appear in the table. This could be for a variety of reasons including, 1) these ad providers are easy to sign-up for, or 2) these ad providers have less sophisticated click fraud detection. Fourth, apps fabricate clicks to at most one ad provider during the experiment. To reduce their risk of being detected, we expect miscreants to rotate between different ad providers but this does not seem to be the case. Interestingly, some apps fabricated clicks to different ad providers during this experiment than in the previous experiment. We speculate that the apps may select an ad providers on start-up rather than rotating while they are running.

## 6.3 Dataset Comparison

We analyze the network traffic generated from our malware apps and compare the result with our measurements from the crawled app dataset. Figure 6 shows the popularity of various ad providers in our crawled apps, malware apps, and from App Brain, an organization that presents statistics about Android apps on Google Play. App Brain publishes ad provider statistics [2] based on the *presence* of ad libraries in apps. It provides an interesting comparison to our results, which measure *usage* of ad libraries by apps. There are clear discrepancies between these three datasets. Notably, apps in the malware dataset prefer certain ad providers (WAPS, AdsMogo, AppLovin) while avoiding other (Admob, Mobclix). This could be due to certain providers vetting developers more aggressively, or because malware apps may target non-English markets. The popularity of Airpush in the malware dataset is explained by the numerous security and privacy infringements found in the Airpush ad library [26], meaning some of the apps in the malware dataset may only be there because they include the Airpush library. Finally, we note that App Brain's methodology for measuring ad provider popularity does not indicate which providers are actually used. Unused libraries may be dead code, or used only rarely by apps. In addition, determining which libraries are included in an app using only static analysis is complicated by programs like Proguard [24], which obfuscate package and class names for Android apps.

### 6.3.1 Ad Fraud Differences between Datasets

We now investigate the prevalence of fraudulent ad behavior in the malware dataset. We find a total of 8,974 apps made 57,619 impressions overall, and that 1,835 apps made 16,565 impressions while in the background. Of the clicks we measured, 27 clicks were generated from 6 apps in the malware dataset. Interestingly, the rates of fraudulent behavior

| App | Source | # Installs | # Impressions | # Clicks | Click Interval (s) | Ad providers |
|---|---|---|---|---|---|---|
| 79b85a | Opera | 236 | 63 | 18 | 1,935.6 | MobFox |
| 9e5b41 | SlideME | 915 | 63 | 18 | 1,925.7 | MobFox |
| f56bda | Google Play | 1,000 | 63 | 17 | 2,049.3 | MobFox |
| 5a6fc0 | Opera | 117 | 54 | 6 | 5,795.6 | MobFox |
| 63fd85 | Opera | 255 | 54 | 6 | 5,798.7 | MobFox |
| 76a7dc | Opera | 125 | 54 | 6 | 5,797.3 | MobFox |
| 86cd17 | SlideME | 915 | 54 | 5 | 6,956.4 | MobFox |
| 94bfa8 | Google Play* | 500 | 4,366 | 5 | 8,353.1 | Migital |
| 7bb12f | 1Mobile | N/A | 4,392 | 4 | 6,717.8 | Migital |
| 807a0a | BrotherSoft | N/A | 98 | 2 | 0 | AppsGeyser |
| d9162a | BrotherSoft | N/A | 4,385 | 2 | 16,919.2 | Migital |
| 57b67c | Google Play* | 10,000 | 56 | 1 | N/A | AppsGeyser |
| a3d816 | ? | N/A | 4,381 | 3 | 10,000 | Migital |
| b611ea | ? | N/A | 4,374 | 1 | N/A | Migital |
| c7681c | ? | N/A | 4,416 | 1 | N/A | Migital |
| d55ece | ? | N/A | 4,384 | 1 | N/A | Migital |
| c31310 | Malware | N/A | 63 | 15 | 2,323.1 | MobFox |
| 1fcc39 | Malware | N/A | 54 | 6 | 5,796.5 | MobFox |
| 247e2e | Malware | N/A | 414 | 6 | 5,797.6 | MobFox |
| 721797 | Malware | N/A | 414 | 6 | 5,796.0 | MobFox |
| 35e164 | Malware | N/A | 54 | 5 | 6,956.1 | MobFox |
| 9863d9 | Malware | N/A | 414 | 5 | 6,955.2 | MobFox |
| Total | | 14,063 | 32,670 | 139 | | |

Table 5: Results of running apps that fabricate clicks for six hours. During the experiment, only 22 (16 crawled from various markets and 6 from our malware dataset) of the 27 apps tested fabricated clicks after being found doing so in previous experiments. An asterisk next to an app from Google Play indicates that the app is no longer available. Apps whose sources are labeled '?' were downloaded by our crawlers but their markets were not recorded.

in the malware dataset tend to be lower than those in the crawled dataset. For example, 9.5% of crawled apps have an impression in the background, whereas 5.2% of malware apps have an impression in the background. This could be due to a number of factors. First, this could mean that malware apps do not commonly perform ad fraud (remember the malware dataset contains apps that are likely malware, but also includes some goodware). On the other hand, the security company that collected the malware samples may not check for ad fraud. Lastly, malware may be more likely to perform emulator detection and thus would avoid performing malicious activities if the malware apps detect that they are run in an emulator. The relationship between malware and ad fraud on Android is an interesting direction for future work.

# 7. LIMITATIONS

Here we discuss limitations of our study. We first discuss three limitations that may have led us to underestimate the prevalence of fraud behavior in apps. First, we ran apps in emulators instead of on real devices. It is possible, though we have not observed it, that some ad libraries may refuse to display ads while in an emulator, and some fraud apps may not send fraudulent traffic to avoid being analyzed. Second, we do not interact with the apps and thus we may not reach a UI state where apps would perform fraud. Given that our work was inspired by botnet ad fraud on the web, we would like to find apps which issue fabricated impressions and clicks silently to the user. Fraud apps which only perform fraud on certain UI states risk not being able to perform the fraud when users are not interacting with the app. Regardless, if exploring the UI of apps is desirable, MAdFraud could be run using a more sophisticated dynamic analysis tool which can explore the UI state of apps without accidentally clicking on ads (in Section 5.1 we ran MAdFraud using the output of the PyAndrazzi dynamic analysis tool, for example). Designing such a system is non-trivial, however, as different ad libraries implement ads differently. Thus, heuristic approaches would be prone to false positives, where ads are clicked accidentally, causing MAdFraud to flag the apps as having click fraud. This may cause us to overestimate our results. Third, we ran all our emulators on a single static IP address. It is possible that some ad providers may have blocked our IP address during our experiments. However, given the prevalence of NAT in our network and that we run no more than 70 apps concurrently, we believe that this is unlikely.

Lastly, we discuss a type of fraud our system cannot detect, *display fraud* where apps obscure or hide ads in the UI of the app. Liu, et al. [13] investigate display fraud on Windows Mobile via a system called DECAF which analyzes the UI structure of apps. Their approach is complementary to our system, as the two systems detect different types of ad fraud. The display fraud detected by DECAF requires the user to run the app (so that ads can be obscured), and to interact with it (so that the user can be coerced into clicking on an ad). On the other hand, MAdFraud detects fraud that is performed silently to the user, either ads requested in the background or ad clicks without user interaction. If we combine the methodologies of MAdFraud and DECAF, we could detect new types of fraud that neither system can detect currently; for example, apps that request ads while running in the foreground but do not put the ad in the UI of the app.

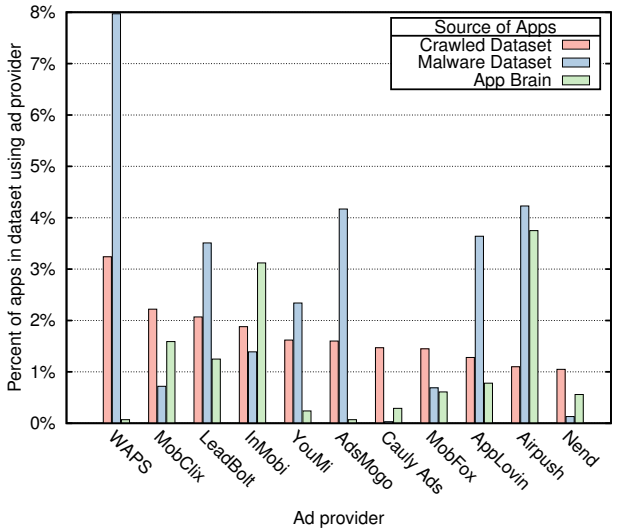


Figure 6: Ad provider popularity by dataset. The list of ad providers differs slightly from those in Figures 4 and 3 because those figures were limited to apps on the top markets. The results from our crawled dataset may overestimate the popularity of ad providers as we only select apps that have the `INTERNET` permission. Again, we omit Admob from the list of ad providers because of its overwhelming popularity. In comparison, Admob is in 47.21%, 28.3%, and 36.64% of crawled apps, malware apps, and App Brain apps, respectively.

# 8. RELATED WORK

## 8.1 Web Advertising

Ad fraud has been studied extensively in the context of online Web advertising. Daswani [6] provides an overview of terms and techniques used for Web ad fraud. Many detection and defense techniques have been proposed to combat web ad fraud. Some detection techniques focus on detecting *duplicate clicks* [30, 17], where a publisher inflates its clicks by clicking the same ad many times. Other techniques detect fabricated impressions and clicks by aggregating ad traffic across client IP addresses and cookie IDs and looking for clients whose ad traffic deviates from expected behavior [16, 18] However, both of these techniques are thwarted by sophisticated botnet ad fraud [28], which uses many compromised machines to vary the IP addresses and cookie ID of fraudulent requests. Fraud performed on many mobile devices would be similarly resistant to these detection techniques. Defenses against ad fraud focus on defeating economic incentives of fraud [21, 15] or using false ads to find malicious publishers [10].

## 8.2 Mobile Advertising

There is very little security research on ad fraud on mobile devices. Notably, Liu [13] investigates *display fraud* on Windows Mobile, which uses techniques that analyze the UI of apps to determine if developers are hiding, obfuscating, or stacking ads to increase their ad revenue or coercing users to click ads. These techniques would not be able to detect when apps fabricate ad traffic in the background. Symantec [5] and Lookout [7], two security companies, provide case studies of malware which uses ad fraud to make money. These include a type of click fraud called *search engine poisoning*, where search engine results can be modified by clicking on links in search engine results to increase their page rank. We note that despite similar nomenclature, this type of click fraud is independent of ad click fraud. Finally, plagiarized applications have been found to use advertising to make money by replacing the ad libraries of victim applications with libraries configured to use the plagiarist's account details [8]. Despite the damage that plagiarism causes to developers, this attack is not ad fraud, as plagiarized apps will still show ads to users.

# 9. CONCLUSION

We have taken the first step to study mobile ad fraud on a large scale. We developed a system and approach, MAdFraud, for running mobile apps, capturing their network traffic, and identifying ad impressions and clicks. To deal with the wide variety of formats of ad impressions and clicks, we proposed a novel approach for automatically identifying ad impressions and clicks in three steps. We discovered and analyzed various fraudulent behavior in mobile ads.

**Acknowledgements** This paper is based upon work supported by the National Science Foundation under Grant No. 1018964 and by the Intel Science and Technology Center for Secure Computing. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Intel Science and Technology Center for Secure Computing.

We would like to thank the anonymous reviewers for their insightful feedback. For his help obtaining Android apps, we would like to thank Clint Gibler. For helping us evaluate our apps that did not make ad requests, we would like to thank Eric Gustafson and Kristen Kennedy. Finally, for helping us manually investigate click URLs, we would like to thank Sam Dawson.

# References

[1] *AdSense Terms and Conditions*. URL: `https://www.google.com/adsense/localized-terms`.

[2] AppBrain. *Android Ad Networks*. 2013. URL: `http://www.appbrain.com/stats/libraries/ad`.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. 1996. URL: `http://www.isi.edu/in-notes/rfc1945.txt`.

[4] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. "SMOTE: Synthetic Minority Over-sampling Technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

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

[6] Neil Daswani et al. "Online advertising fraud". In: *Crimeware: understanding new attacks and defenses* (2008).

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

[8] Clint Gibler et al. "AdRob: Examining the Landscape and Impact of Android Application Plagiarism". In: *Proceedings of 11th International Conference on Mobile Systems, Applications and Services*. 2013.

[9] 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.

[10] Hamed Haddadi. "Fighting online click-fraud using bluff ads". In: *ACM SIGCOMM Computer Communication Review* 40.2 (2010), pp. 21–25.

[11] *How Facebook Beats Ad Fraud*. URL: `http://www.businessweek.com/articles/2013-11-26/how-facebook-beats-ad-fraud`.

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

[13] 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.

[14] Peter Lowe. *Ad blocking with ad server hostnames and IP addresses*. 2013. URL: `http://pgl.yoyo.org/as/`.

[15] Mohammad Mahdian and Kerem Tomak. "Pay-per-action model for online advertising". In: *Proceedings of the 1st international workshop on Data mining and audience intelligence for advertising*. ACM. 2007, pp. 1–6.

[16] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Detectives: detecting coalition hit inflation attacks in advertising networks streams". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 241–250.

[17] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Duplicate detection in click streams". In: *Proceedings of the 14th international conference on World Wide Web*. ACM. 2005, pp. 12–21.

[18] Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi, and Qi Zheng. "On hit inflation techniques and detection in streams of web advertising networks". In: *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*. IEEE. 2007, pp. 52–52.

[19] *Millennial Media Terms and Conditions*. URL: `https://tools.mmedia.com/login/termsAndConditions/index`.

[20] *MobFox Terms of Service*. URL: `http://www.mobfox.com/terms-of-service/`.

[21] Hamid Nazerzadeh, Amin Saberi, and Rakesh Vohra. "Dynamic cost-per-action mechanisms and applications to online advertising". In: *Proceedings of the 17th international conference on World Wide Web*. ACM. 2008, pp. 179–188.

[22] Vern Paxson. "Bro: a system for detecting network intruders in real-time". In: *Computer networks* 31.23 (1999), pp. 2435–2463.

[23] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[24] *ProGuard*. URL: `http://proguard.sourceforge.net/`.

[25] Dominik Schürmann. *Adaway*. 2013. URL: `http://sufficientlysecure.org/index.php/adaway/`.

[26] T. Spring. *Sneaky Mobile Ads Invade Android Phones*. 2013. URL: `http://www.pcworld.com/article/245305/sneaky%5C_mobile%5C_ads%5C_invade%5C_android%5C_%20phones.html`.

[27] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. "Investigating user privacy in android ad libraries". In: *Workshop on Mobile Security Technologies (MoST)*. 2012.

[28] 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.

[29] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. "Adnostic: Privacy Preserving Targeted Advertising." In: *NDSS*. 2010.

[30] Linfeng Zhang and Yong Guan. "Detecting click fraud in pay-per-click streams of online advertising networks". In: *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE. 2008, pp. 77–84.