AdRob: Examining the Landscape and Impact of Android Application Plagiarism

Clint Gibler
UC Davis
cdgibler@ucdavis.edu

Ryan Stevens
UC Davis
rcstevens@ucdavis.edu

Jonathan Crussell
UC Davis
jcrussell@ucdavis.edu

Hao Chen
UC Davis
hchen@cs.ucdavis.edu

Hui Zang
Sprint Research
hui.zang@sprint.com

Heesook Choi
Sprint Research
heesook.choi@sprint.com

ABSTRACT

Malicious activities involving Android applications are rising rapidly. As prior work on cyber-crimes suggests, we need to understand the economic incentives of the criminals to design the most effective defenses. In this paper, we investigate application plagiarism on Android markets at a large scale. We take the first step to characterize plagiarized applications and estimate their impact on the original application developers. We first crawled 265,359 free applications from 17 Android markets around the world and ran a tool to identify similar applications ("clones"). Based on the data, we examined properties of the cloned applications, including their distribution across different markets, application categories, and ad libraries. Next, we examined how cloned applications affect the original developers. We captured HTTP advertising traffic generated by mobile applications at a tier-1 US cellular carrier for 12 days. To associate each Android application with its advertising traffic, we extracted a unique advertising identifier (called the client ID) from both the applications and the network traces. We estimate a lower bound on the advertising revenue that cloned applications siphon from the original developers, and the user base that cloned applications divert from the original applications. To the best of our knowledge, this is the first large scale study on the characteristics of cloned mobile applications and their impact on the original developers.

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*

General Terms

Security, Economics

MobiSys'13, June 2528, 2013, Taipei, Taiwan

Keywords

Android; underground economy; plagiarism; advertising; mobile; measurement

1. INTRODUCTION

As mobile applications become more popular and lucrative, they also become a more likely target for criminals and other miscreants. A malicious activity unique to mobile application markets is large-scale application plagiarism [2]. This is because mobile applications, especially Android applications, are straightforward to reverse engineer and copy. We define a *cloned* application as an app that is a modified copy of another app, and thus shares a significant portion of its application code with the original. A *plagarized* application is a cloned app that was fraudulently copied from one developer to another. Prior studies showed that indeed there are many cloned applications on mobile markets [6, 28]. However, these studies leave many questions unanswered. From the plagiarists' point of view, what are their incentives? From the users' point of view, how often do they run plagiarized applications? From the original application developers' point of view, to what extent does the practice of application plagarism impact them? The answers to these questions would have deep research, technical, and policy implications. For example, even if there were many plagiarized applications on the mobile markets, if users rarely download and run them, perhaps dealing with them is not a priority. On the other hand, if the plagiarized applications severely affect the economic interests of original application developers, we must deal with them swiftly and adequately to sustain interest in legitimate application development.

We take the first step toward answering the above questions. We use a technique that combines static application analysis with network analysis to bring a number of relevant insights into this problem. These techniques allow us to find cloned applications in the wild, to analyze the prominent properties among these clones, and to analyze the popularity and profitability of these clones. During the process, we face several challenges. First, we must download a large number of applications from various markets (because often the plagiarized and original applications appear on different markets) and detect clones among them using an automatic tool. Second, we must capture a large amount of live network data from mobile applications, allowing us to extract

relevant advertising information to bring insight into each app's advertising revenue. Note that these traces must be from large numbers of real users unaffiliated with and unaffected by our study. This precludes generating traces in our lab, because the users aren't "real," and capturing network traffic at our university, because the user population is not large or diverse enough. Finally, we must associate the original and plagiarized applications with their network traces to determine how much advertising revenue plagiarized applications receive relative to the original. For the purpose of this work, we attempt to distinguish *rebranded* apps (apps from the same developer with high code similarity) from truly plagiarized apps by merging developer accounts accross markets. We collectively refer to these techniques, along with the results presented, as *AdRob*.

1.1 Overview

The analysis used for AdRob consists of the following steps (Figure 1).

- We crawled 265,359 free Android applications from 17 markets around the world. Then, we ran a tool [6] to find clones within these applications, resulting in 44,268 cloned apps.
- We captured live HTTP advertising traffic generated by mobile applications at a tier-1 US cellular carrier for 12 days, resulting in 2.6 billion packets and 19,125,311 ad impressions.
- We link cloned applications detected in our lab to their network traffic by their *client IDs*. Most free Android applications include one or more advertising libraries. For each ad library, the application includes a client ID, which is sent along with the ad requests so that the ad provider can credit the application developer for ad impressions or clicks. We use static analysis to extract client IDs from the downloaded applications. Then, we extract client IDs for popular ad providers from captured HTTP traffic. Finally, we correlate these two sets of client IDs.

Using the data acquired in the above steps, we first examine properties of the cloned applications, including their distribution across different markets, application categories, and ad libraries. Next, we examine how cloned applications affect the original developers. We estimate a lower bound on the percent revenue that cloned applications siphon from the original developers, and the percent user base that cloned applications divert from the original applications. To the best of our knowledge, this is the first large scale study on the characteristics of cloned applications and their impact on the original developers.

We make the following contributions:

- We conduct a large scale study on the characteristics of cloned applications and their impact on the original developers. This serves as a first step toward understanding the incentives of application plagiarists.
- We combine static analysis with live network analysis to correlate the static properties of an application to its network characteristics. Using this methodology, we link applications crawled from major Android markets to their live traces in a tier-1 US cellular network carrier, which allows us to understand the impact of plagiarism broadly and deeply.
- We propose the use of client IDs in ad requests to correlate Android applications to their live network traces. We present approaches for extracting client IDs from both application code and network traces.
- We describe an approach to determine the *ownership* of apps using developer accounts, signing keys, and client IDs.

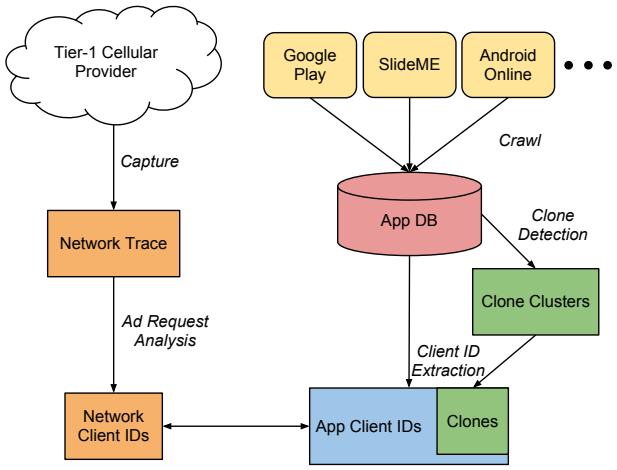


Figure 1: Overview of the AdRob methodology.

Non-Goal.

This work does *not* consider ad fraud. While app plagiarism detriments primarily the original developers of the plagiarized apps by siphoning their users and advertising revenue, ad fraud harms mainly the advertisers. Moreover, online ad fraud has been extensively studied [7, 11, 27]. Our work is the first to measure the impact of app plagiarism on the original developers.

2. BACKGROUND

2.1 Android Background

Android is a Linux-based smart phone operating system designed by Google that is designed to run Android applications (apps) that are downloaded from various Android app markets. An Android application is distributed as an `apk` file, which is similar to a Java `jar` file. The `apk` is a zip archive which contains all the code and data needed to install and run the app.

The largest and most popular app market is Google Play (also called the official Android market), which consists of over 550,000 applications by various developers [4]. In addition to Google Play, there are a number of third-party Android markets where users can go to download apps, for example the Amazon Appstore or SlideMe. There are various reasons developers would choose to release their apps on third-party markets: many do not charge to create a developer account and upload applications, they may have less stringent terms of service, or developers may simply want

to increase the exposure of their apps across these different markets.

Before an app may be submitted to a market, it must be digitally signed by a certificate for which the developer owns the private key. This certificate is used by Android to determine authorship and trust relationships between apps [23]. Android phones and emulators will only install apps that are signed. Android developer certificates may be self-signed, so there is little barrier for a developer to create as many certificates as she wishes. However, for Android to allow an app to seamlessly update to the next version, the newer version must be signed with the same certificate. Otherwise, the user will be prompted to install the newer version as if it were a separate app.

An important file which is included with every app is the Android Manifest file. This XML document contains a number of parameters that the Android framework needs in order to run the app. This includes the names of the Activities, which are the different screens of the app, the permissions the app requires and the API version. Developers may also use this XML document to store any additional information the app may use; for example, advertising parameters are sometimes specified here.

2.2 Android Application Cloning

Android applications are written in Java and compiled into Dalvik bytecode in order to run on Android's Dalvik Virtual Machine. Dalvik bytecode is very well structured and clearly separates code from data which makes it much easier to reverse engineer than traditional machine code.

To protect their application from reverse engineering, prudent developers may use a tool called Proguard [16] to perform class and function name obfuscation. It can also be used to obfuscate the package name of included libraries. However, Proguard alone is not enough to protect apps from being reverse engineered or cloned. Open-source tools such as ApkTool [5] and smali [10] make it straightforward for unscrupulous individuals to reverse engineer, modify, and recompile the Dalvik bytecode of an Android application for distribution.

2.3 Advertising on Android

An Android developer who wants to make money by displaying advertisements as part of her application must sign up with an *ad provider* and download the provider's *advertising SDK*, which is a library (in the form of a `jar` file) that the developer includes in the app. The SDK provides an API for displaying an ad in the application, and abstracts away the complexity involved with fetching, displaying, and managing advertisements for the ad provider (an overview of which is shown in Figure 2).

In order to receive payment for ads shown in the application, the developer is given a *client ID* by the ad provider that uniquely identifies the app; this client ID is then specified in the application such that it is available to the ad library SDK. When it is time to display an ad, the SDK sends an *ad request* over HTTP to the ad provider's ad server. The ad request includes the client ID, a device identifier (for example, the IMEI), and other fields such as demographic information. The exact format of the ad request and the fields present differs between ad providers. Once the request is received, the ad server responds with the image URL of the ad to display and a click URL that opens in the browser

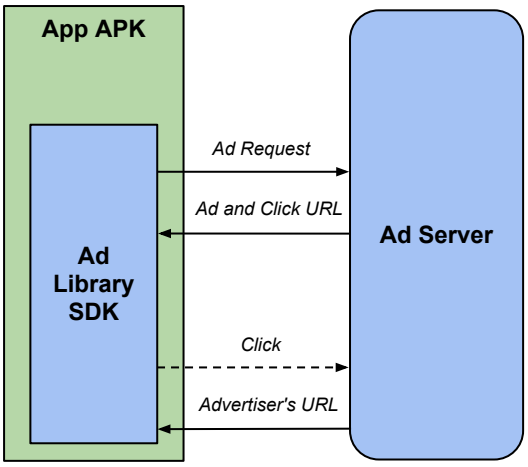


Figure 2: Overview of ad library SDK and ad server interaction.

app if the ad is clicked. A successful ad request and associated response is called an *impression* and represents one instance of an ad being displayed to the user. To track the clicks, the ad server generates a unique click URL for each impression which points to one of the provider's ad servers. When the browser opens this page the click is matched with its associated impression and recorded before the browser is redirected to the advertisement's landing page. Unlike ad requests, the click request does not contain the client ID of the application as the ad server can lookup the client ID once the click is matched with the impression that generated it. As we will explain in Section 4, this extra level of indirection prevented us from obtaining suitable click results.

3. DATASET

In this section we describe the data used in our study. The two main datasets are a large collection of downloaded Android applications, which we analyzed to find clones, and 3G network data from a tier-1 cellular carrier, which provided us insight into the advertising behavior of applications in our database.

3.1 Application Database

Our collection of Android applications consists of 265,359 Android apps from 17 markets around the world. Figure 4 shows these markets. The apps were collected via automated crawling and stored in a database along with any meta information provided by the market (such as developer name, number of downloads, and category). Only free applications are included in the collection. The breakdown across various markets is as follows: 73.7% of the apps are from the official Google Play market, 14.7% are from 9 third-party English markets, 13.8% are from 6 third-party Chinese markets, and 0.46% are from 2 Russian markets. Even though we may not have crawled all the markets or have downloaded all the applications from each market, our analysis in Section 4 indicates that our collection represents a significant portion of all the applications that include ads running on US cellular networks.

3.2 Plagiarized Applications

Using a more scalable tool based on DNADroid [6], we analyzed our application database to look for plagiarized applications. While DNADroid is highly precise because it is based on calculating subgraph isomorphism on PDGs, it is not scalable because it requires pair-wise comparison of apps. To avoid pairwise comparison, the tool used for this work converts each app into a set of semantic vectors, each of which represents a semantic feature of the program. Then, the tool clusters the semantic vectors of all the apps using LSH [1]. Finally, the tool considers a set of apps to be clones if they share a large number of semantic vector clusters and places these apps into the same *clone cluster*.

Each clone cluster contains applications published by different developers, as determined by their signing certificate. Each application appears in at most one cluster. In total, we investigate 5,431 clone clusters consisting of 44,268 unique applications. These clusters may or may not contain the "victim" of plagiarism, which we call the "original app" in this paper. Determining the original application is difficult and unreliable, as described in Section 6.2.1.

3.3 Network Data

We had access to live 3G network data from a tier-1 cellular network carrier, which consisted of a sample of all traffic that passed through a particular home agent in the southwest United States. Packets were tapped from the home agent and sent to the carrier's research lab for capture and analysis. The tap at the home agent was quarter-sampled at the flow level, so in theory all packets in a particular flow were available to be captured; however, packets were lost from limitations in bandwidth and capture speed of the tapping infrastructure. The network data included all 3G internet traffic sent from devices registered on the carrier's cellular network, including protocol headers and payload.

We captured HTTP traffic from the tap over the course of 12 days: from 28 June 2012 to 10 July 2012. Devices present on the network include both Android and iOS devices, as well as feature phones and wireless cards. The traffic characteristics are similar to those in [18]. To optimize the capture, only 32 bytes of the protocol header were captured along with the payload for each packet. Additionally, we performed IP filtering so that only packets with a source or destination IP address that corresponded with a known Android ad server were captured. In total, we captured 2.6 billions packets in our trace. We describe how we extract ad request data from the trace in Section 4.1.

4. METHODOLOGY

Figure 1 shows the procedure of our study. Our data came from two sources: (1) HTTP traffic generated by Android applications on a US tier-1 cellular network, and (2) Android applications downloaded from Android markets. We correlated the two datasets using advertising client IDs. In this section we explain how we extracted these IDs, both from the network and from applications.

4.1 Extracting Client IDs from Network Traces

We now explain how we extract client IDs from ad requests sent by Android applications. We extract advertising information from our cellular network trace (described in Section 3.3) by analyzing ad requests at the packet level as well as at the flow level. In order to identify ad requests, we manually ran a sample application for each Android ad provider and captured all traffic from the device so that we could characterize the format of each provider's requests. From each ad request, we record the ad provider, client ID, anonymized user identifier, and the application package name if available. We anonymize user identifiers to avoid recording any potentially sensitive device identifiers, such as the IMEI. We only record ad requests with an Android HTTP `User-Agent` field.

To get packet level data, we look at the HTTP header of each packet to determine if it is an ad request from a recognized Android ad provider and record relevant advertising information. To analyze ad requests at the flow level, we use the 32 bytes of protocol header to reconstruct each TCP flow, allowing us to observe both the ad request and the response from the server. The benefits of looking at ad requests at the flow level are twofold. First, we observed that ad requests for some ad providers are split across multiple packets, especially when the request uses HTTP POST, so it would be impossible to record the entire ad request in this case by only looking at packets individually. Second, the server response allows us to extract the click URL associated with each impression so that we can match the impression with any observed clicks in the trace. These clicks can be linked to an impression and thus a client ID by using the click URLs that we parse from each flow.

To extract client IDs, we first identify the ad provider by examining the host name in the HTTP request. For each ad provider, we create a pattern to identify its client IDs in the HTTP requests accurately.

4.2 Extracting Client IDs from Applications

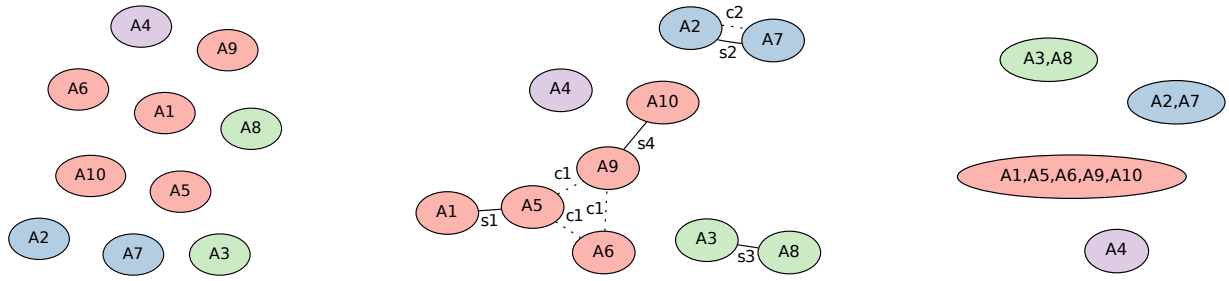
We extract client IDs from Android applications via static analysis. Commonly, ad providers require developers to provide their client ID in one of five ways: declaring it in the application's manifest file, declaring it in a layout XML file, passing it to an ad object in the application code, specifying it as a constant string, or receiving it dynamically over the network.

4.2.1 In Android Manifest

The most straightforward client IDs to extract are those declared in the Android manifest file. *Admob*, for example, recommends that developers store their client IDs in a meta-data field with the name ADMOB_PUBLISHER_ID. Usually the client ID is directly included in the Manifest XML; however, in some cases there can be extra levels of indirection: Android allows developers to abstract values into constant files such as `string.xml`. In cases where the developer has abstracted the client ID using notation such as `@string/admob_id`, we resolve these abstractions accordingly.

4.2.2 In Layout XML

Some ad providers recommend that client IDs be included in one of the auxiliary XML files that developers generate for their application (not the manifest). Conveniently for us, these ad providers usually provide an example for developers to copy and replace the client ID with their own, which provides a common naming convention for each ad library. We take advantage of this convention by using XPATH to search XML files for specific XML elements known to contain



(a) Initial owners based on developer accounts (Phase 1)

(b) Owners with edges based on shared certificates (s) and client IDs (c)

(c) Owners after merging (Phases 2 and 3)

Figure 3: Owner graph at various stages in owner merging.

client IDs. For example, *Jumptap* client IDs are usually in a field called `jumptaplib:publisherid`.

4.2.3 In Application Code

Client IDs that applications pass directly to ad library initialization methods are the most challenging to extract. The initialization may happen at any point in the code and the client ID may be instantiated in a number of ways, including being passed in as a constant, received over the network, generated dynamically, or read in from a file. We observed that the first two are the most common cases and focus on them accordingly.

Specified as Constants.

We dump values from the DEX string constants section in the application's `dex` file and search for strings that match the structure of client IDs for each ad library. This approach is effective for ad providers whose client IDs follow a distinctive pattern. To improve the precision of this analysis, we also implemented a static analysis tool that performs constant propagation to detect when constant values are passed to the client ID argument of ad library initialization methods. Unfortunately, we found it too slow to run on all of the apps in our dataset, as it took several minutes per app. We leave improving the precision of extracting constant-based client IDs in this manner to future work.

Received Over the Network.

In addition to ad libraries, many applications include Adwhirl's SDK. Adwhirl is not an ad provider, but rather offers a service that allows developers to dynamically decide which ad provider and client ID should be used in their ad requests. Before an application with Adwhirl makes an ad request, it queries Adwhirl with its Adwhirl client ID and receives a set of ad providers and client IDs, which the application then uses to send ad requests. This service allows developers to dynamically change the advertising behavior of their apps without having to push a new version of their app. We handle apps that use Adwhirl by first extracting Adwhirl client IDs from the application. Then, we query Adwhirl's server to obtain a mapping from an Adwhirl client ID to a set of client IDs and their ad providers. This allows us to obtain many client IDs not hardcoded in the applications.

From our application database, we extracted 1,386 unique Adwhirl client IDs, which mapped to 1,886 unique client IDs from one of the five ad providers we eventually used in this paper (described in Section 5.1). Of these, 1,539 (or 81.6%) were new client IDs that we had not extracted from our applications. Specifically, 764 were for *admob*, 523 for *millennial*, and 252 for *inmobi*.

4.3 Determining Ownership

As our goal is to measure the impact of plagiarism on developers, we must not regard as plagiarism similar apps from the same developer, known as *rebranded* apps. Rebranded apps may appear on the same or different markets. In this section we discuss how we determine app ownership by considering developer accounts, signing keys (represented by certificates), and client IDs. We make the following assumptions: first, each owner has her own unique set of certificates that are not shared with any other owner; second, each owner has her own unique set of client IDs that are not shared with any other owner. In Section 7.1.1 we discuss caveats associated with these assumptions. Note, however, that these caveats only cause us to merge owners more aggressively, resulting in lowering our measured cloning impact.

Merging owners consists of three phases. First, we build an owner graph where each node corresponds to a unique developer account that has uploaded an application that we crawled (Phase 1). Next, we merge owners based on shared certificates (Phase 2). Finally, we merge owners based on shared client IDs (Phase 3). An overview of the owner graph at various steps in the process is shown in Figure 3.

Phase 1: The first phase of merging owners is to create a graph with a node for each unique developer account. Each developer account is identified by the tuple of the market name and the developer's name and is associated with one or more apps in our database. Assuming that developer accounts are rarely compromised, then two apps from the same developer account must come from the same owner. Thus, developer accounts provide an initial mapping of apps to owners.

Phase 2: Next, we compute the set of certificates whose corresponding private keys were used to sign the apps uploaded by each developer account. As mentioned in Section 2, every Android app must be signed by the developer's private key before it may be uploaded to a market and then

Provider	Impressions	Clicks	Unique IDs
admob	9,288,333	920	348,275
airpush	3,212,878*	n/a	138,166
inmobi	220,982*	207	n/a
millennial	4,855,247	n/a	61,127
mobclix	1,547,871	1,080	32,751

Table 2: Number of observed impressions, clicks, and anonymized user identifiers in the network trace, broken down by ad provider. The report of impression counts uses packet level ad request analysis, unless otherwise noted (via *). Note that we were not able to measure clicks for all providers due to the format of their ad requests (for example, those with chunked HTTP encoding). Also note that inmobi does not include a user identifier in their ad requests.

installed on users' phones. Since private keys are supposed to be kept private (we discuss shared and stolen private keys in Section 7.1.1), two developer accounts that share a certificate must share the same owner so they are merged in the graph.

Phase 3: Finally, we merge any two owners whose applications share a client ID. Client IDs are used by ad providers to uniquely identify who should receive revenue for each ad shown or clicked. Therefore, if two developer accounts share a client ID, they are likely to be from the same owner.

We separate the merging of owners by certificates and client IDs into two phases because we have lower confidence about merges based on client IDs. This is because client ID extraction may have false positives while certificate calculation cannot. Other situations, such as a plagiarist who leaves the original developer's client ID intact, are discussed in Section 7.1.1.

5. EVALUATION

In this section we summarize the raw results of our analysis and determine the accuracy of our application client ID extraction.

5.1 Network Client ID Extraction

One important consideration of our work is which Android ad providers to include in our analysis. We started with 16 ad providers based on their popularity in our application database. Among these ad providers, seven[1] had very small number of ad requests in our captured traffic (some of them are Chinese but our traffic was captured in the US), and four had very little overlap between the client IDs observed on the network and client IDs extracted from our application database (Table 1)[2]. Therefore, we focus on the remaining five ad providers, whose number of impressions, clicks, and unique user identifiers are summarized in Table 2. These five ad providers are also among the top six most prevalent Android ad libraries reported by AppBrain [3]. Because our network capture was lossy, we were not able to record significant click results from the trace, as we could only measure clicks when the ad request, server response, and then click request did not experience any loss.

[1]Buzzcity, Mojiva, Quattro, Vdopia, Wooboo, Youmi, ZestAdz

[2]These four providers only represented 7% of the total number of ad requests we collected.

5.2 Application Client ID Extraction

While extracting client IDs from HTTP traffic generated by Android applications is highly accurate, extracting them automatically from Android applications themselves may not be as reliable, because client IDs may be provided in several different ways, some of which are unfriendly to program analysis (Section 4). Therefore, we would like to evaluate the accuracy of our client IDs extracted from Android applications. However, it is difficult to determine the ground truth, because doing so would require us to manually review all the applications, which is prohibitive given our large number of applications. One might suggest that we run each application in an emulator to extract the client IDs from its HTTP traffic. However, an application may contain multiple ad libraries, so we may not observe the client IDs from all these libraries during the execution of the application. Moreover, some ad libraries would not send ad requests if they detect that they are running in emulators.

Instead, we take advantage of the client IDs extracted from HTTP traffic to estimate a lower bound of the accuracy of our client ID extraction from applications. For each ad provider AP, let D_{AP} be the set of client IDs extracted from our application databases, and N_{AP} be the set of client IDs extracted from our captured HTTP traffic. The *network coverage*, $|N_{AP} \cap D_{AP}|/|D_{AP}|$, estimates a lower bound of the true positive rate of our client ID extraction from applications. The *database coverage*, $|D_{AP} \cap N_{AP}|/|N_{AP}|$, estimates the coverage our application database on all the Android applications running on the network where we captured traces.

The results of this analysis are summarized in Table 1. The database coverages for most providers are fairly high, indicating that our application database represents a significant portion of all ad-supported Android applications that are used in the geographic area where we collected the trace.

6. FINDINGS

6.1 Properties of Apps in Clone Clusters

There are a number of questions that come to mind when investigating our clone clusters: Which markets do apps in the clusters belong to? Which markets have the highest proportion of apps involved in cloning? What categories are most represented in the clusters? What advertising libraries do cloners prefer? In this section we seek to answer these questions by investigating common features of apps in our clone clusters. Note, at this point we do not yet speculate as to which app in each clone cluster is the original and which are clones.

6.1.1 Market Characteristics

Figure 4 shows which Android markets are most prevalent in our clone clusters. A significant percentage of apps in our clone clusters are from Google Play; intuitively this makes sense as Play is the most popular Android market. To compensate for this observation, we computed what percentage of all applications in our database from the market are present in a clone cluster. In this context, Play does not stand out more than other markets, but a number of other markets such as AndroidSoft, and the Chinese markets AndroidOnline, GoApk, and AppChina, have more than a quarter of their apps in our clone clusters. To better understand the cloning relationship between markets, we calculated the number of similar apps between each pair of

Ad Provider	Unique client IDs		Network coverage	Database coverage
	from databases	from network		
admob	51,434	19,718	21.2%	55.4%
airpush	8,728	8,829	36.9%	36.5%
inmobi	514	487	28.6%	30.2%
millennial	786	2,030	32.6%	12.6%
mobclix	2,994	1,781	38.0%	63.9%
adfonic	59,170	318	0.0%	0.0%
greystripe	59,616	212	0.3%	68.4%
jumptap	8	78	0.0%	0.0%
smaato	0	144	n/a	0.0%

Table 1: Percentage of extracted client IDs we observed in network traffic and percentage of observed network client IDs we also extracted from the application database, broken down by provider. The lower group of providers are ones which we did not include in our study because they did not have significant overlap with our application database.

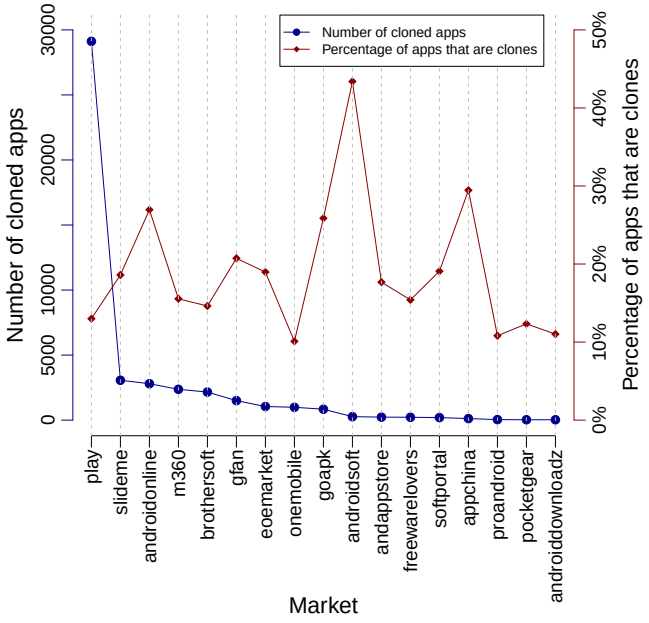


Figure 4: Plot showing the popularity of different app markets in our clone clusters. The absolute number of cloned apps from each market is represented by the axis labelled "Number of cloned apps", whereas the axis labelled "Percentage of apps that are clones" represents the popularity of each market in our clone clusters normalized over the total number of apps from that market in our database.

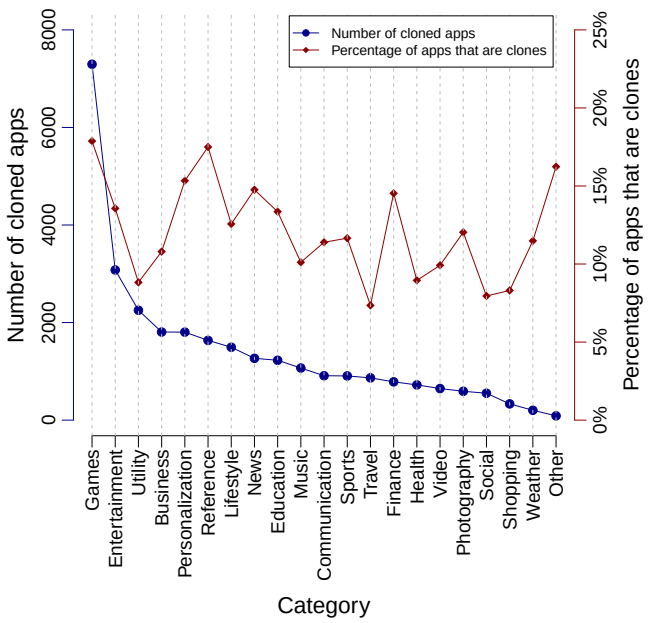


Figure 5: Plot showing the popularity of different app categories in our clone clusters. The absolute number of cloned apps in each category is represented by the axis labelled "Number of cloned apps", whereas axis labelled "Percentage of apps that are clones" represents the number of cloned apps in each category normalized over the total number of apps in that category in our database.

markets in a clone cluster. Specifically, for the apps in given clone cluster and a pair of markets (I, J):

$$MarketSim_{cluster}(I, J) = \min(\text{apps from } I, \text{apps from } J)$$

Then, to get a global view of the amount of similar applications between markets we calculate the total $MarketSim(I, J)$ as follows:

$$MarketSim(I, J) = \sum_{c \in clusters} MarketSim_c(I, J)$$

In Figure 7, we plot the results of this calculation in an undirected graph. To reduce the complexity of the graph, we only show an edge between markets whose $MarketSim$ value is over 300 and remove unconnected markets.

We define a market as a "cloning hub" if it shares an edge with many different markets in the graph. Play is the largest cloning hub among all the markets, as it has a significant cloning relationship with almost every other market. Similarly, AndroidOnline is a cloning hub among the Chinese markets. The existence of these cloning hubs implies one of two things: either plagiarists prefer cloning from apps on these markets, or that plagiarists prefer these markets to publish their cloned apps. Since we do not speculate on which app in a cluster is the original, we leave the determining which of these cases is true to future work. Note that the

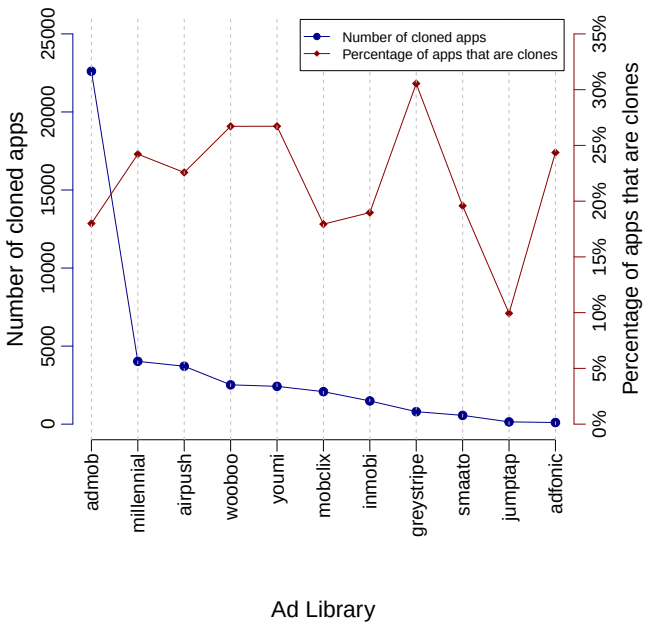


Figure 6: Plot showing the popularity of different ad libraries among apps in our clone clusters. The absolute number of cloned apps with each library is represented by the axis labelled "Number of cloned apps", whereas axis labelled "Percentage of apps that are clones" represents the number of cloned apps with each library normalized over the total number of apps with that library in our database.

prevalence of Play as a cloning hub is likely influenced by the fact Play represents 73.7% of the apps in our database.

6.1.2 Category Characteristics

We now investigate which app categories are popular among apps in our clone clusters so that we can better understand what types of apps are involved in cloning. One difficulty with comparing categories among apps from different markets is that different markets use different category names to refer to the same type of application. To avoid this problem, we chose 21 meta-categories that represent the spectrum of different categories observed across all our markets (our mapping from category strings to meta-categories is presented in the Appendix). Figure 5 presents the number of applications in our clone clusters that belong to each meta-category. As we did for markets, we normalize the number of cloned apps in each category with the number of all apps in our database for that category to determine what percentage of apps in the category are involved in cloning. Interestingly, "Games" is the most popular category among apps in the clone clusters, but also has the highest prevalence of apps involved in cloning. Thus, markets that care about application cloning should focus on apps categorized as "Games." Additionally, assuming that the original app and the clone belong to the same category, this implies plagiarists prefer applications categorized as "Games." We hypothesize that this is because games are relatively complex, popular among Android users, and are often run for long periods of time, allowing more advertising revenue to be generated.

6.1.3 Ad Library Characteristics

Figure 6 gives a breakdown by ad provider of the applications in our clone clusters, as well as normalized across our entire application database. Admob is the most popular provider among cloned applications, but is also the most popular among all applications, and thus does not have a higher percentage of cloned apps compared with other providers. On the other hand, for our Chinese ad providers, Wooboo and Youmi, cloned applications represent a larger subset of the total applications that use these providers. Nonetheless, an alarmingly large percentage of applications for each provider are in our clone clusters, meaning that they either are clones of another app, or are a legitimate app that has been cloned. Note that we do not consider what percentage of ad traffic was generated by our clone clusters for each ad provider, as we do not speculate which app is the original and want to avoid any assumptions regarding how much traffic for a provider is a result of app plagiarism. In the next section, we estimate a lower bound for how much advertising revenue cloning siphons from legitimate applications.

6.2 Comparing Clones and Non-clones

In previous sections we examined a number of properties of apps in our cloned clusters without distinguishing the "original" apps from the clones. In this section, we wish to gain insight into the impact of cloning on developers. Specifically, we investigate the effects of cloning on the original authors' ad revenue and user install base. To do so, the original author in the cluster must be determined, which is surprisingly nontrivial.

6.2.1 Determining Original App

There are a number of intuitive approaches one could use to determine which application among a cluster of similar applications is the original. Unfortunately, most of these initial approaches are flawed. For example, one could use:

- Date first uploaded to the market
- Application popularity by number of installs or rating
- Code size by number of methods, instructions, or other metric

The date an application was first uploaded to the market is difficult to know as an external observer. Each market knows when the application was first uploaded, but unless an external observer has been crawling markets since their creation in both the free and paid sections, she cannot know for sure which app came first. Additionally, there have been cases where beta releases have been taken and uploaded to markets before the original developer's app. Application popularity may sometimes differentiate the clone from the original, for example in the case of Angry Birds, however, for less popular applications, users may be just as likely to download the clone as the original. Further, application popularity by both number of installs and ratings is vulnerable to sybil attacks which would be relatively easy to perform as most market accounts require only a valid email address. Lastly, the code size of applications can easily be distorted

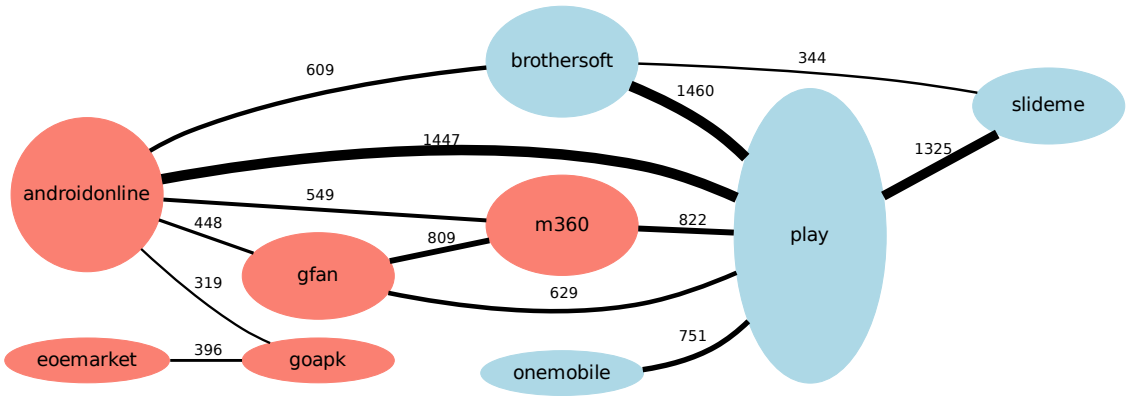


Figure 7: Markets which have a significant cloning relationship. The thickness of an edge is proportional to the $MarketSim$ value between the markets, and the height of a node is proportional to the sum of the edge weights for a given market. Markets with nodes colored blue are US-based markets whereas markets with nodes colored red are Chinese-based markets.

	P1	**P2**	**P3**
Num Clusters	610	370	128
$LostImps_{total}$	1,764,609	1,076,614	56,884
$PercLostImps_{total}$	52%	51%	14%
$LostUsers_{total}$	166,534	122,718	3,808
$PercLostUsers_{total}$	49%	48%	10%

Table 3: Total amount of lost impressions and users across all of our clone clusters between each phase of developer merging. The value of Num Clusters represents how many clone clusters had more than one owner after each phase. P1, P2, and P3 represent Phase 1, Phase 2, and Phase 3, respectively.

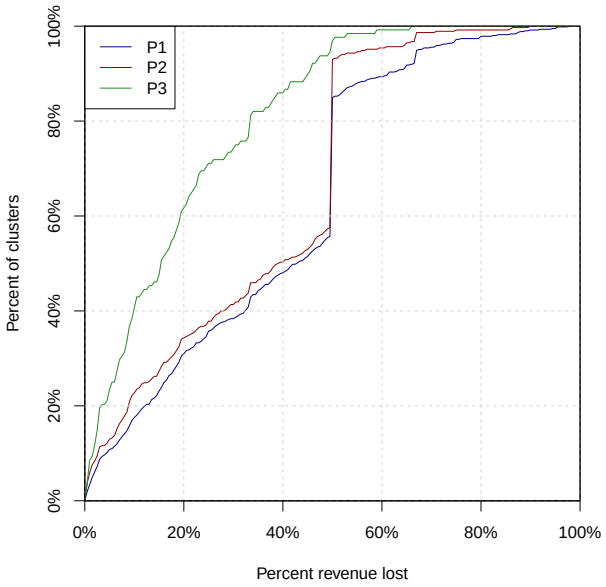
by plagiarists to make the plagiarized app appear larger or smaller.

Rather than rely on one of these flawed approaches, we instead use an approach that estimates a lower bound on our findings. For each cluster we deem the owner with the most observed impressions to be the original author and all other owners are treated as plagiarists. In the case where clones are more popular than the original, we will mislabel the original author as a plagiarist, but this is acceptable as we will only underestimate the percentage of lost advertising traffic. Indeed, this approach guarantees we will not overestimate the number of lost impressions from mislabeling the original as the above approaches might, as we only consider the least popular owners in a clone cluster as plagiarists (other caveats presented in Section 7.1.2). As we will see, merging developer accounts has significant impact over our results when app owners within a clone cluster are merged by certificate or client ID, and in some cases may attribute all apps within a cluster to the same merged owner. We do not consider such cases in our computations of lost revenue, as they do not represent fraudulent cloning.

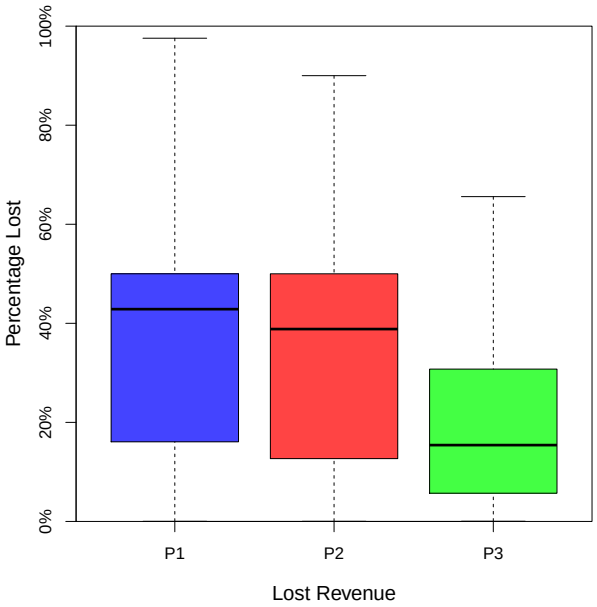
6.2.2 *How much revenue do clones siphon from the original authors?*

In order to determine the impact of plagiarism on the advertising revenue of the original application, we first observe that hard dollar amounts are difficult to determine when looking at advertising network traffic alone. This is because an impression does not explicitly indicate how much it is worth in an ad request, as the ad provider does not want to disclose how much it or its affiliated developers are making. Instead, we show what percentage of lost ad traffic developers experience by comparing the ratio of impressions that belong to plagiarized applications compared with the total number of impressions we observed belonging to applications in our clone clusters. The number of lost impressions for a cluster is represented by $LostImps_{cluster}$, and the percentage of lost impressions for a cluster is this value divided by all impressions attributed to apps in the cluster, times 100 (represented by $PercLostImps_{cluster}$). Additionally, $LostImps_{total}$ represents the total number of plagiarized app impressions over all the clusters, and $PercLostImps_{total}$ represents what percentage of all impressions from all clusters that are a results of $LostImps_{total}$.

As previously stated, we consider the original author in a cluster as the app owner with the greatest amount of advertising traffic, thus the numerator is the number of impressions for all apps in the cluster not owned by this author. Additionally, we do not consider clusters with only a single owner. Figure 8a shows a CDF of the $PercImpsLost_{cluster}$ values across our clone clusters for each stage of application owner merging. When only considering developer accounts on their own (Phase 1 of the owner graph, see Section 4.3) nearly 25% of the clusters have exactly 50% lost revenue. This is a result of clusters having exactly two applications which belong to different developer accounts but which share a client ID. Intuitively, we would want to remove these anomalies by merging developer accounts where it makes sense. However, merging by certificates alone, which is very accurate, does not significantly change the distribution of lost revenue across our clusters (Phase 2 of owner graph). Thus, we use the results of Phase 3 of the owner graph where owners are merged by client IDs to ensure that our results do not overestimate the percentage of lost revenue. Figure 8b shows the impact of merging at each stage on the $PercImpsLost_{cluster}$ values, and Table 3 sum-

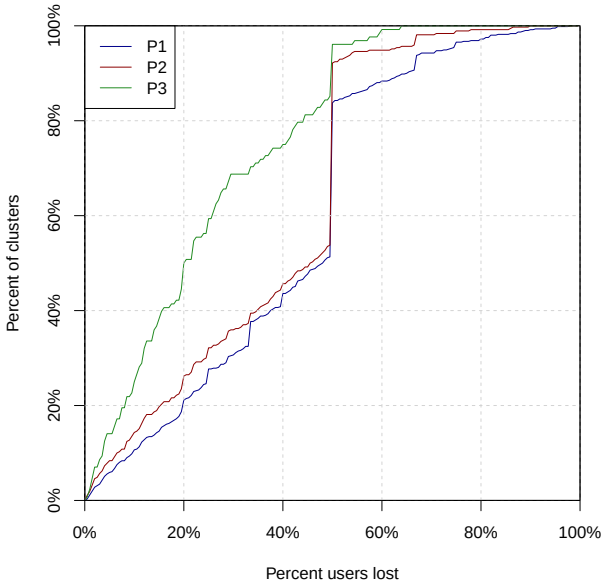


(a) $PercLostImps_{cluster}$

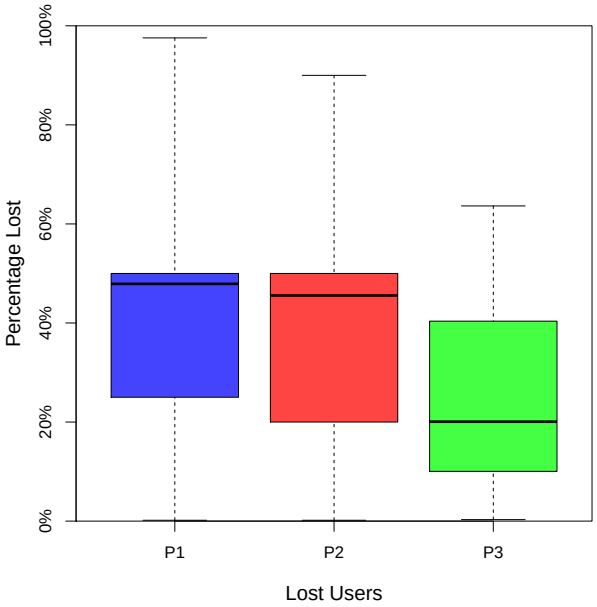


(b) $PercLostImps_{total}$

Figure 8: CDF and boxplot of percent impact to developers in terms of impressions for each phase of merging on the owner graph. P1, P2, and P3 represent Phase 1, Phase 2, and Phase 3, respectively.



(a) $PercLostUsers_{cluster}$



(b) $PercLostUsers_{total}$

Figure 9: CDF and boxplot of percent impact to developers in terms of users for each phase of merging on the owner graph. P1, P2, and P3 represent Phase 1, Phase 2, and Phase 3, respectively.

marizes the total impact over all the clusters. Alarmingly, the value of $LostImps_{total}$ after all developer account merging is complete is 14%, meaning that authors who are the target of cloning on average lost 14% of their advertising revenue to application plagiarism, assuming the users who downloaded the plagiarized versions would have used the original app instead.

6.2.3 How much of the user base do clones divert from the original app?

To determine how much user base application plagiarism siphons from the original authors, we used the anonymized device identifiers present in the ad traffic from our network trace to estimate the number of unique users per application. Similar to lost impressions, we compute the $LostUsers_{cluster}$ and $PercLostUsers_{cluster}$ values for each cluster and plot the distribution in Figure 9a. We also compute the total number of lost users across all our clusters as $LostUsers_{total}$ and the percentage lost across all the clusters as $PercLostUsers_{total}$. The value of these are summarized in Figure 9b and Table 3. We see that on average, 10% of the user base for an author in our clone clusters is siphoned due to plagiarism if we assume the users who downloaded the plagiarized applications would have used the original app if no plagiarism were present.

7. DISCUSSION

7.1 Limitations

7.1.1 Authorship Determination

During our exploration of potential methods for merging owners we encountered several challenges. All of these challenges could cause our analysis to merge developer accounts more aggresively, potentially causing us to underestimate lost impressions and users due to plagiarism.

Certificates Shared By Many Apps.

We found that some certificates are present on many apps that are associated with a large number of developer accounts on several markets. Eight certificates were present on apps from over one hundred developer accounts, the largest having 413. After manually reviewing the developer accounts associated with several of the most prevalent certificates, we decided that either one or several parties made a significant attempt to distribute their apps across many developer accounts or that the private keys were leaked or stolen. In fact, we discovered one of the certificates corresponds to a private key used in the Android Open Source Project[3] (AOSP). These keys are often used to sign custom ROMs. Apps signed with these keys can run with greater privileges when running on a custom ROM and some malware has been found to take advantage of this [21]. Regardless of why many developer accounts share a certificate, we were surprised to find that two apps sharing a certificate may not necessarily have the same owner.

Apps With Multiple Certificates.

Though it has not been discussed in the literature to the best of our knowledge and is only briefly mentioned in the Android documentation page on signing apps [23], Android apps may be signed with multiple developer keys. We found 667 apps in our dataset that have more than one certificate, with the largest having four. Android developers may do this as a means to collaborate with another developer or to ease the process of selling an app to a company [15, 19].

We hoped to use apps with multiple keys to merge owners in possession of at least one of the keys. Intuitively, if one app is signed by developer keys A and B, one would like to merge the owners of each key, as the developer of an app should have possession of each key that signed her app. We manually reviewed a number of cases where we tried to merge owners based on apps with multiple keys. While in several cases the developer accounts of each key seemed related, this was not always the case. Thus, in this work we ignore any app with multiple certificates as they may taint our results. Furthermore, these apps constitute a very small portion of our dataset (667 out of 265,359 apps).

Lazy Cloning.

We define "lazy cloning" to be cases when a cloner copies an app but does not change all of the client IDs in ad libraries contained in the original app. This would cause our analysis to merge the owners representing the original developer and the cloner, as they are responsible for apps that share a client ID. We believe this occurs infrequently in practice, as one of the major incentives behind cloning is financial gains.

7.1.2 Estimation on Lost Revenue and User Base

Since we cannot identify the original app in a clone cluster reliably, we choose the most popular app, measured by the number of ad impressions, as the original app. While this decision provides a lower bound estimation on the percent lost revenue and user base in most cases, it would fail in the following unusual cases.

Failure to merge owners.

When we fail to merge two different developer accounts of the same owner, we could inflate the number of unique owners in a clone cluster. This would happen when an owner publishes similar apps from different developer accounts using different certificates and different client IDs. Additional information, such as payment accounts and addresses, from app markets or ad providers could improve our app ownership determination.

Sharing of client IDs between different apps.

We assume that one client ID is specific to one family of similar apps. However, if a developer includes the same client ID in different apps but only some of them are in a clone cluster, we could inflate the number of impressions generated by the apps in the cluster. To verify this occurs rarely in practice, we calculated the total number of client IDs of which only a subset were in a cluster. Only 0.65% of our client IDs appear in apps both in and out of a cluster; thus, client ID sharing does not significantly impact our results.

Missing original apps.

If our database fails to include the original app for a cluster, we may overestimate the losses in cases where the missing original app is very popular. For example, we could find a large impression or user base differential between two Angry Birds clones, leading us to calculate a strong developer

[3]http://source.android.com/

loss. However, by missing the original Angry Birds app we wouldn't see that it had many impressions, so the real loss ratio is quite low. We believe this is unlikely to occur as our crawling focused on both newly published and popular apps.

Paid apps.

The estimated lost revenue and users is based on the assumption that if a clone app were unavailable, its users would have installed the original app and spent the same amount of time on it. Since we focus exclusively on free apps, it is reasonable to assume there is little difference between finding, installing, and running the original app and its clones. However, this assumption may not hold when considering paid apps.

7.2 Potential Steps to Reduce Cloning

A primary goal of our effort to gain insight into the current state of Android app cloning is to protect Android developers and users. Technical solutions such as the automatic plagiarism detection methods used for this work [6] should be employed by markets to improve the speed with which clones are caught and removed from markets. Reducing the lifetime of clones on markets limits their downloads and thus their impact.

However, as Android apps are straightforward to decompile, modify, and resubmit to markets, we do not believe technical solutions alone are sufficient. Detection tools can always continue to be improved, but it is unlikely that a tool can catch *every* clone, whether it's due to lack of access to every app across every market or significant code obfuscations. Instead, we believe reducing economic incentives is a more effective way to limit app cloning. For example, if all markets began charging at least a nominal registration fee then cloners would have to make back the registration cost from their clones before the account is banned or they will lose money. This also disincentives a plagiarist from creating many developer accounts, possibly making plagiarism detection easier. Similarly, ad providers could also charge a registration fee or delay ad revenue payout for some period of time to allow the developer's apps to be vetted. By increasing the time and/or cost to create developer accounts or sign up with ad providers, legitimate developers may be slightly affected but cloners wishing to create many developer or ad provider accounts can be significantly impeded.

We offer these potential solutions with caution, as one of the great advantages of the Android ecosystem is its openness and low barrier to entry. The associated costs must be carefully weighed to reduce cloning while not discouraging legitimate developers.

8. RELATED WORK

Malicious activities on Android.

Prior work examined information leaks within an application [9] and between applications [8]. Researchers discussed privacy violations in third-party libraries, especially advertising libraries [12, 22, 20, 24]. [29] characterized Android malware. DroidMOSS [28] and DNADroid [6] detected repackaged and cloned Android applications.

Examining the impact of malicious activities.

Kanich et al. investigated the underground economy of spam [14, 17]. Prior work examining the impact of malicious activities primarily focused on fraud that harmed the user, such as fake anti-virus software [26], keyloggers [13], and spam [25]. By contrast, application plagiarism detriments mainly the original developers of the plagiarized applications by taking away their users and advertising revenue. Online advertising fraud has been extensively studied in the literature [7, 11, 27]; however, application cloning does not necessarily imply advertising fraud because the advertiser still receives impressions and clicks from the users of cloned applications.

9. CONCLUSION

In this paper we presented AdRob, which is designed to be the first step towards understanding the economic incentives of application plagiarism on Android markets. We characterized application plagiarism and its impact on developers by crawling 265,359 free applications from 17 Android markets around the world and detecting clones among them. We captured live HTTP traffic generated by mobile applications at a tier-1 US cellular carrier for 12 days, and extracted client IDs from both applications and network traces to correlate the two datasets. Based on the data, we first examined properties of the cloned applications, including their distribution across different markets, application categories, and ad libraries. Next, we examined how cloned applications affect the origin developers. We estimated a lower bound on the revenue that cloned applications siphon from the original developers, and the user base that cloned applications divert from the original applications, and find an alarmingly high percentage of impressions is siphoned from developers who are victims of plagiarism. To the best of our knowledge, this is the first large scale study on the characteristics of cloned applications and their impact on the original developers.

10. ACKNOWLEDGEMENTS

We would like to thank Sharad Agarwal and the anonymous reviewers for their insightful feedback, as well as Liang Cai, Dennis Xu, Ben Sanders, Justin Horton, and Jon Vronsky for their assistance in obtaining Android applications. This paper is based upon work supported by the National Science Foundation under Grant No. 1018964.

References

[1] A. Andoni and P. Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In: *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. Ieee. 2006, pp. 459–468.

[2] Jason Ankeny. *Feds seize Android app marketplaces Applanet, AppBucket in piracy sting*. Aug. 22, 2012. URL: `http://www.fiercemobilecontent.com/story/feds-seize-android-app-marketplaces-applanet-appbucket-piracy-sting/2012-08-22`.

[3] AppBrain. *Android Ad networks*. Mar. 2013. URL: `http://www.appbrain.com/stats/libraries/ad`.

[4] AppBrain. *Number of available android applications*. Nov. 2012. URL: `http://www.appbrain.com/stats/number-of-android-apps`.

[5] Brut.alll. *Android-Apktool*. URL: http://code.google.com/p/android-apktool.

[6] J. Crussell, C. Gibler, and H. Chen. "Attack of the Clones: Detecting Cloned Applications on Android Markets". In: *Computer Security–ESORICS 2012* (2012), pp. 37–54.

[7] N. Daswani et al. "Online advertising fraud". In: *Crimeware: Understanding New Attacks and Defenses* (2008).

[8] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach. "Quire: lightweight provenance for smart phone operating systems". In: *USENIX Security*. 2011.

[9] William Enck, Landon P. Cox, and Jaeyeon Jung. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: (2010).

[10] Jesus Freke. *Smali/Baksmali*. URL: http://code.google.com/p/smali.

[11] Mona Gandhi, Markus Jakobsson, and Jacob Ratkiewicz. "Badvertisements: Stealthy click-fraud with unwitting accessories". In: *Online Fraud, Part I Journal of Digital Forensic Practice, Volume 1, Special Issue 2*. 2006.

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

[13] T. Holz, M. Engelberth, and F. Freiling. "Learning more about the underground economy: A case-study of keyloggers and dropzones". In: *Computer Security–ESORICS 2009* (2009), pp. 1–18.

[14] Chris Kanich et al. "Show Me the Money: Characterizing Spam-advertised Revenue". In: *USENIX Security Symposium*. San Francisco, CA, Aug. 2011.

[15] *Keystore and Aliases - is there a use to multiple aliases?* Dec. 2012. URL: http://stackoverflow.com/questions/2667399/keystore-and-aliases-is-there-a-use-to-multiple-aliases.

[16] Eric Lafortune. *Proguard*. URL: http://proguard.sourceforge.net.

[17] Kirill Levchenko et al. "Click Trajectories: End-to-End Analysis of the Spam Value Chain". In: *IEEE Symposium and Security and Privacy*. Oakland, CA, May 2011.

[18] H. Liu, C.N. Chuah, H. Zang, and S. Gatmir-motahari. "Evolving Landscape of Cellular Network Traffic". In: *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*. IEEE. 2012, pp. 1–7.

[19] *Multiple Signatures and Shared UIDs*. Dec. 2012. URL: https://groups.google.com/forum/?fromgroups=#!topic/android-security-discuss/LyyEWyFg5xc.

[20] P. Pearce, A.P. Felt, G. Nunez, and D. Wagner. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM. 2012.

[21] *Security Alert: Malware Found Targeting Custom ROMs (jSMSHider)*. Dec. 2012. URL: https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/.

[22] S. Shekhar, M. Dietz, and D.S. Wallach. "Adsplit: Separating smartphone advertising from applications". In: *CoRR, abs/1202.4030* (2012).

[23] *Signing Your Applications*. Dec. 2012. URL: http://developer.android.com/tools/publishing/app-signing.html.

[24] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. "Investigating User Privacy in Android Ad Libraries". In: *IEEE Mobile Security Technologies (MoST), San Francisco, CA* (2012).

[25] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. "The underground economy of spam: A botmasters perspective of coordinating large-scale spam campaigns". In: *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. 2011.

[26] B. Stone-Gross et al. "The underground economy of fake antivirus software". In: *Economics of Information Security and Privacy III* (2011), pp. 55–78.

[27] B. 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.

[28] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. "Detecting repackaged smartphone applications in third-party android marketplaces". In: *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM. 2012, pp. 317–326.

[29] Y. Zhou and X. Jiang. "Dissecting android malware: Characterization and evolution". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 95–109.

APPENDIX

Table 4: Mapping from market category strings to meta-categories. Meta-categories with just one market category are excluded.

Meta-category	Market category
Business	Business
	Enterprise
Entertainment	Comics
	Entertainment
Games	Arcade & Action
	Brain & Puzzle
	Cards & Casino
	Casual
	Fun & Games
	Games
	Racing
	Sports Games
Health	Health
	Health & Fitness
	Medical
Music	Music
	Music & Audio
News	News
	News & Magazines
Other	Developer / Programmer
	Home & Hobby
	Other
	Religion
Personalization	Personalization
	Wallpapers
Reference	Books & Reference
	E-books
	Ebooks & Reference
	Reference
Social	Collaboration
	Social
	Social Responsibility
Travel	Transportation
	Travel
	Travel & Local
Utility	Email & SMS
	Libraries & Demo
	Location & Maps
	Productivity
	System
	Tools
	Utilities
Video	Media & Video
	Multimedia