SAND2012-10765C

Characterizing Android Application Plagiarism and its Impact on Developers

Clint Gibler
UC Davis
cdgibler@ucdavis.edu

Ryan Stevens
UC Davis
rcstevens@ucdavis.edu

Jonathan Crussell
UC Davis,
Sandia National Labs
jcrussell@ucdavis.edu,
jcrusse@sandia.gov

Hao Chen
UC Davis
hchen@cs.ucdavis.edu

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 304,275 free applications from 19 Android markets around the world and ran a tool to identify similar applications ("clones"). Next, we captured live HTTP advertising traffic generated by mobile applications at a tier-1 US cellular carrier for 12 days. To correlate each Android application with its network traces, we extracted a unique advertising identifier (called the client ID) from both the applications and their network traces. 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 estimate 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. 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.

1. INTRODUCTION

Criminals and other miscreants follow new computing technologies closely to take advantage of them, as exemplified by email spam, malicious web sites, botnets, etc. As mobile applications are leading a new wave of innovation, malicious activities targeting mobile application markets are on the rise. Many researchers studying computer-related crime observe that miscreants are primarily motivated by economic incentives. While technical solutions might mitigate the harm caused by these malicious activities to some extent, as long as the economic incentives remain, criminals will continue to operate. Therefore, to deter these criminals, we must limit their economic incentives, which requires us to first measure and understand these incentives.

TODO: Perhaps it would be better to talk about malware after plagarism in the intro, as we dont want to start with what we didnt do. Perhaps the most well known malicious activity on smartphones is malware. Malware usually manifests itself as seemingly benign applications which have additional malicious functionality, such as stealing private information or root exploits [22]. However, we choose not to include malware in this study for two reasons. First, different malware developers have vastly different incentives, such as stealing users' private information, making unauthorized actions that result in charges, and taking control of the phone. It would be difficult to quantify and compare economic incentives across these different malicious goals. Second, researchers have gained extensive knowledge in studying desktop malware and much of the experience applies to mobile malware too. Thus, our contributions in this area would be minimal.

Besides malware, another prominent malicious activity targeting mobile application markets is plagiarized (or cloned) applications [1]. 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 were many cloned applications on mobile markets [4, 21]. However, these studies leave many questions unan-

swered. 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 technical and policy implications. For example, even if there are 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 maintain legitimate developer interest in application development.

We take the first step toward answering the above questions. We use a technique that combines static application analysis and network analysis to bring a number of relevant insights into this problem. These techniques allow us to find cloned applications in the wild, analyze what properties are prominent among these clones, and 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 many markets (because often the plagiarized and original applications appear on different markets) and use a tool to detect clones among them. 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 number 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 correlate the plagiarized applications with their network traces.

1.1 Overview

Our study consists of the following steps (Figure 1).

- We crawled 304,275 free Android applications from 19 markets around the world. Then, we ran a tool [4] to find clones within these applications, resulting in 44,268 cloned apps. *TODO: Time frame*
- 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 origin developers. We estimate 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. 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 clone authors.
- We link applications crawled from major Android markets to their live traces in a tier-1 US cellular network carrier. This allows us to correlate the static properties of an application to its live network characteristics.
- 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.

2. BACKGROUND

2.1 Android Background

Android is a Linux-based smart phone operating system designed by Google which is designed to run Android applications (apps) that are downloaded from various Android app markets. An Android application is distributed as a `.apk` file, which is similar to Java's jar file format. The apk is an 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 [2]. 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

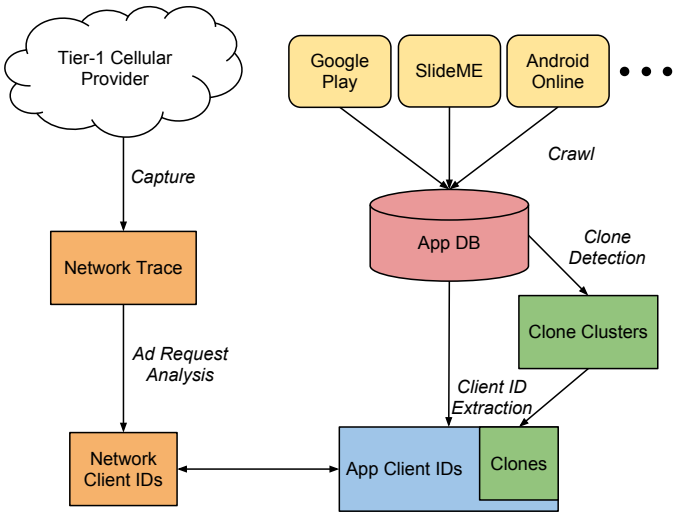

Figure 1: Overview of the methodology we used for our study.

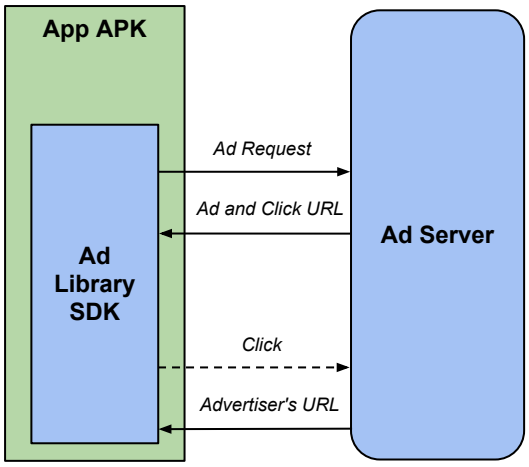

Figure 2: Overview of ad library sdk and ad server interaction.

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 the developer using her private key. This signature is used by Android to determine authorship and trust relationships between apps [16]. Android phones and emulators will not install apps that are not signed. Android developer certificates may be self-signed, so there is little barrier for a developer to create as many signing keys 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 key. 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. In addition to app code written in Java, Android allows developers to include native libraries with the application which interface with the Dalvik bytecode using JNI. This is commonly referred to as "native code" *TODO: Is native code relevant?*

To protect their application from reverse engineering, prudent developers may use a tool called Progaurd [12] 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 [3] and smali [8] make it straightforward for unscrupulous individuals to reverse engineer, modify, and recompile the Dalvik bytecode of an Android application for distribution *TODO: Mention legit uses of apktool, smali?.*

2.3 Advertising on Android

An Android developer that 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 `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* from the ad provider that uniquely identifies the app; this client ID is then hardcoded into the application such that 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 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. *TODO: Sample HTTP for the ad request?*

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 304,275 Android apps from 19 markets around the world. 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 of across various markets is as follows: 73.7% of the apps are from the official Google Play market, 14.7% are from 10 third-party English markets, 13.8% are from 7 third-party Chinese markets, and 0.46% are from 2 from 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. *TODO: List markets in footnote?*

3.2 Plagiarized Applications

Using a more scalable tool based on DNADroid [4], we analyzed our application database to look for plagiarized applications. This tool produced clusters of similar applications, which we verified using DNADroid. DNADroid compares applications using subgraph isomorphism on their Program Dependence Graphs (PDGs), which is robust against syntactical modifications and statement reordering, which plagiarists may use to decrease the likelihood of their clones being detected. In their evaluation of DNADroid, the authors found that it had a very low false positive rate.

Each clone cluster contains applications published by different developers, as determined by the public key signature. Each application appears in at most one cluster. In total, we investigate a total of 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 [13]. 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 application and from

the network.

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. *TODO: Explain why we dont have click results?*

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 three ways: declaring it in the application's manifest file, declaring it in a layout XML file, or passing it to an ad object in the application code.

4.2.1 Client IDs 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 Client IDs 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 searching XML files for specific XML elements known to contain client IDs. For example, *Jumptap* client IDs are usually in a field called `jumptaplib:publisherid`.

4.2.3 Client IDs 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.

Client IDs 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. For example, Inmobi client IDs consistently begin with `4028cb` or `ff8080`. To improve the precision of this analysis, we also implemented a static analysis tool that performs constant propagation on apps to detect when constant values are passed to the client ID argument of ad library initialization methods *TODO: This sentence seems irrelevant.*

Client IDs 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 dynami-

cally 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, 1539 (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*. *TODO: Shouldn't this be in the evaluation section?*

4.3 Determining Authorship

An important consideration when investigating application cloning is that not all cloned applications represent plagarism; we have observed that some developers release a number of very similar apps from the same developer account. We can extend this observation to consider cases where similar applications are released by the same author, but across different developer accounts (perhaps on different markets). We define an *author* to be the person responsible for uploading the final version of an app to a market, even though she may not have developed the majority of the code in the case of cloning. We would like to attribute each application in our clone clusters to its appropriate author (instead of developer account) in order to avoid misrepresenting the impact of cloning in the case that a legitimate developer publishes their apps over many developer accounts. In this section we discuss how we were able to do so, by merging developer accounts based on signatures and client IDs. We make the following assumptions: first that each author has their own unique set of signatures that is not shared by any other author, and second that each author has their own unique set of client IDs that is not shared with any other author. In Section 7.1 we discuss caveats associated with these assumptions. Note, however, that these caveats only cause us to merge more aggresively, lowering our measured cloning impact.

4.3.1 Building Initial Author Set

The first step in our process of determining app authorship is to create an author object for each unique developer key fingerprint on any app in our collection. 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 installed on users' phones. Thus, the maximum number of potential authors behind the apps in our collection is the total number of unique developer key fingerprints across every app (assuming no private keys are shared or stolen, an issue we discuss further in Section 7.1).

However, it is free and easy to create an arbitrary number of signing certificates, and as we expect cloners to attempt to hide their tracks, we take the following steps to connect authors across multiple key signatures.

4.3.2 Merging by Developer Account

Next, we merge two authors when at least one app owned by each was uploaded to the same developer account on the same market. When developers upload an app to a market, it is associated with their developer account on that market. Assuming that developer accounts are rarely hacked (and if so, the newly uploaded apps are quickly removed), then two apps from the same developer account on the same market must come from the same author. The two apps must share the same developer account *and* market because there is no guarantee that the same developer account on two different markets belong to the same author. That is, there is nothing preventing a malicious developer from registering a popular Play developer account name on a new third-party market.

4.3.3 Merging by Client ID

Lastly, we merge any authors whose apps share at least one client ID. As mentioned in Section 2, ad providers give a unique client ID to each developer who signs up for their service. Two independently developed apps have no reason to share a client ID, even if they use the same ad library. Thus we determine apps that share a client ID to be from the same author.

5. EVALUATION

TODO: Should this section be called Results? 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 was which Android ad providers to include in our analysis. We started with 16 ad providers based on their popularity in our application database; however, seven providers[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). Among the remaining 9 ad providers, four of them had very little overlap between the client IDs observed on the network and client IDs extracted from our application database (Table 1). These 5 providers only represented 7% of the total number of ad requests we collected. Of the remaining providers, the number of

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

Provider	Impressions	Clicks	Unique Device 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.

impressions, clicks, and unique user identifiers is summarized in Table 2. 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.

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 extraction 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 when they find that they are running in emulators. *TODO: it says that some ways of specifying client IDs are unfriendly to program analysis but after reading methodology it seems like we got them all – the challenges are commented out.*

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.

6.1.1 Market Characteristics

Figure 3 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 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 4, 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.

We define a market as a "cloning hub" if it shares an edge with many different markets in the graph. Play

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.

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.

6.1.2 Category Characteristics

We now investigate what 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 cluster 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 Section 6.2, 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 cloned apps on the original developers ad revenue and user install base. However, before these may be examined, the original app in the cluster must be determined, which is surprisingly nontrivial.

6.2.1 Determining Original App

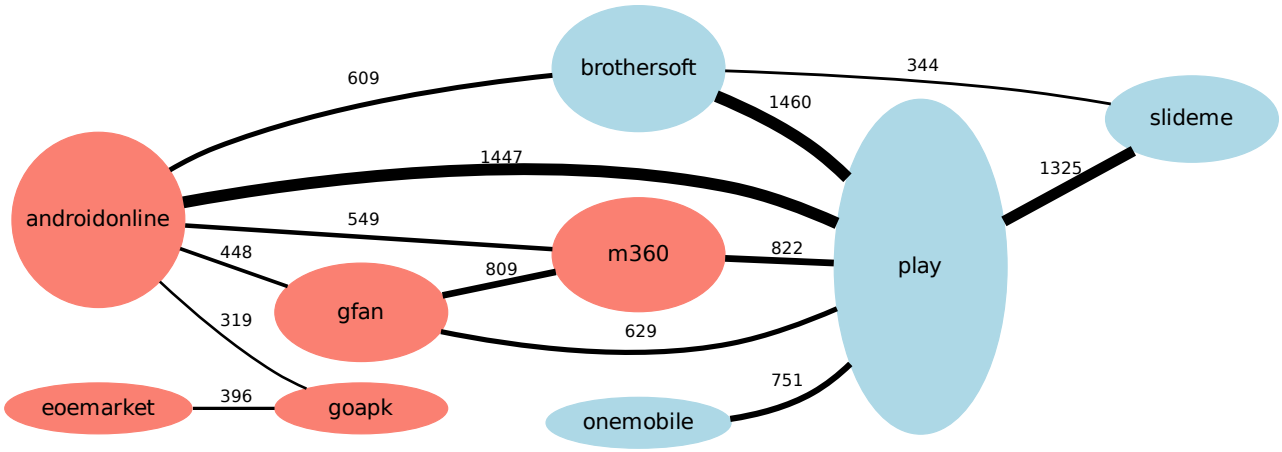

Figure 4: Markets which have a significant cloning relationship. The thickness of an edge represents the magnitude of the cloning relationship between the markets, and the height of a node is proportional to the sum of the edge weights for a given market. Markets with nodes colored blue are US-based markets whereas markets with nodes colored red are Chinese-based markets.

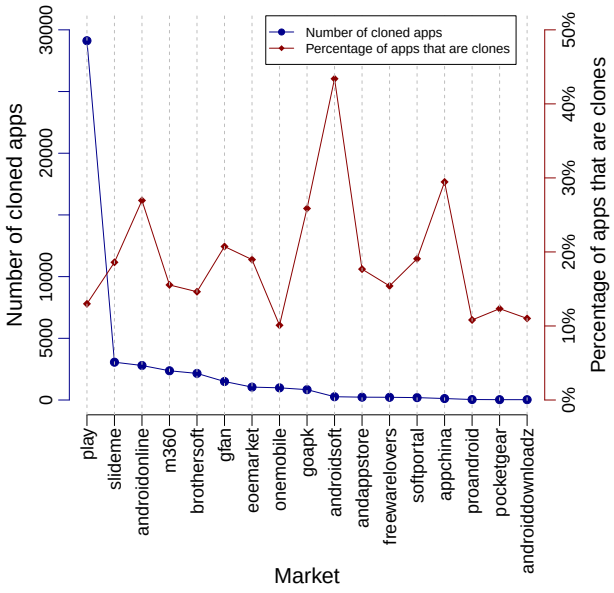

Figure 3: Plot showing the popularity of different app markets in our clone clusters. The absolute number of cloned apps from each market is represented by the axis labelled "Number of cloned apps", whereas the axis labelled "Percentage of apps that are clones" represents the popularity of each market in our clone clusters normalized over the total number of apps from that market in our database.

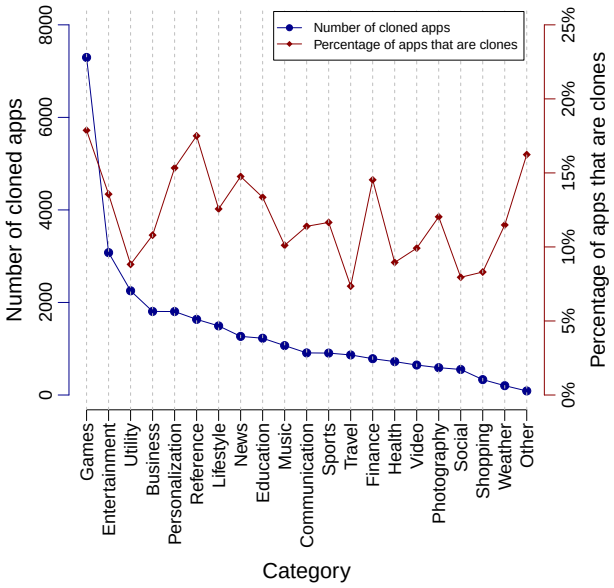

Figure 5: Plot showing the popularity of different app categories in our clone clusters. The absolute number of cloned apps in each category is represented by the axis labelled "Number of cloned apps", whereas axis labelled "Percentage of apps that are clones" represents the number of cloned apps in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

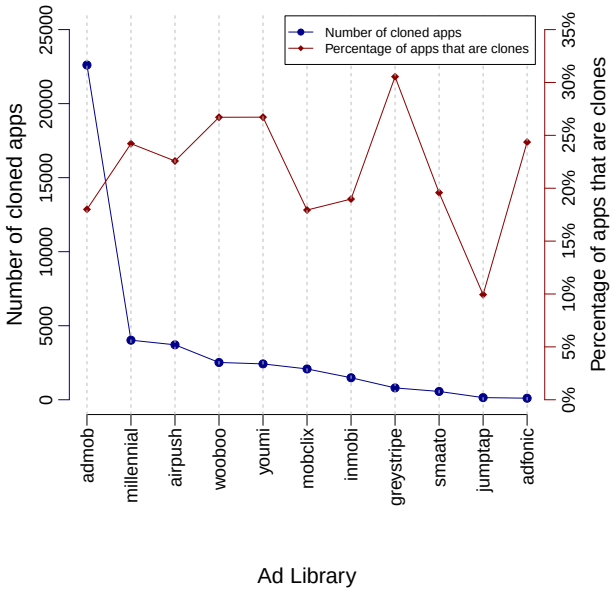

Figure 6: Plot showing the popularity of different ad libraries among apps in our clone clusters. The absolute number of cloned apps with each library is represented by the axis labelled "Number of cloned apps", whereas axis labelled "Percentage of apps that are clones" represents the number of cloned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

There are a number of approaches one could use 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 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.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 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 guarantees a lower bound on our findings. Specifically, for each cluster we deem the developer with the most observed impressions to be the original developer and all other developers are treated as clones. In some cases, we will mislabel the original app as a clone but this is acceptable since we are trying to determine a lower bound. Clearly, if we count the number of impressions received by the original app as siphoned impressions and the clone app received more than the original app, we still ensure a lower bound on the number of impressions siphoned by the *real* clones. This key difference between this approach and the flawed application popularity approach described above is that one cannot guarantee a lower bound using the above approach.

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

In order to determine how much advertising revenue cloned applications siphon from 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

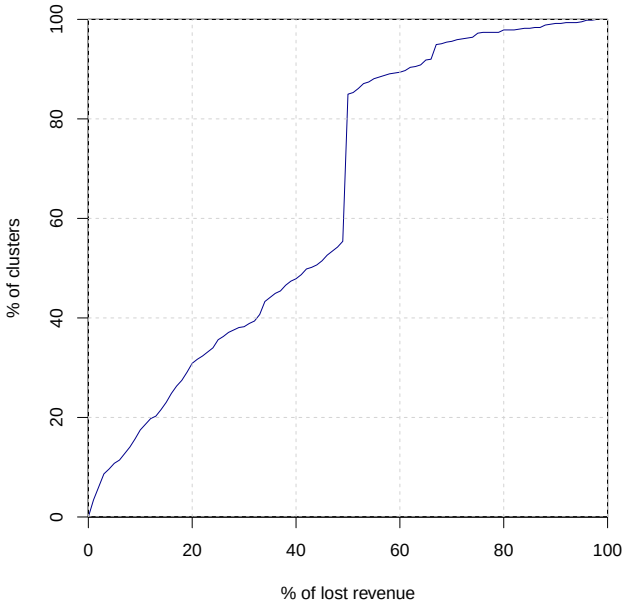

Figure 7: CDF of lost revenue per clone cluster.

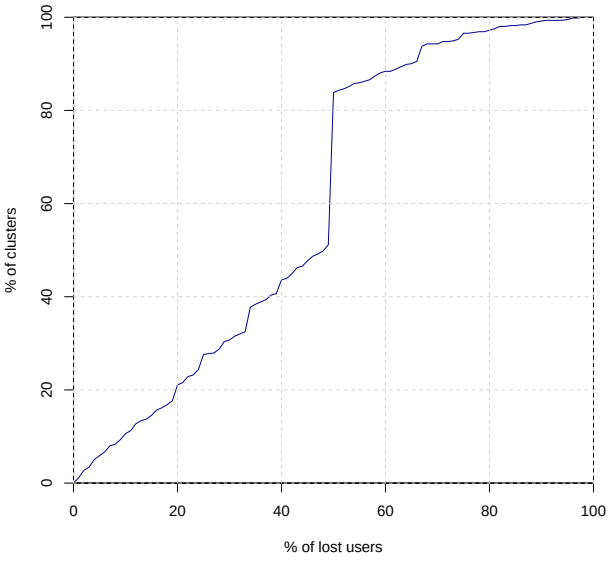

Figure 8: CDF of lost users per clone cluster.

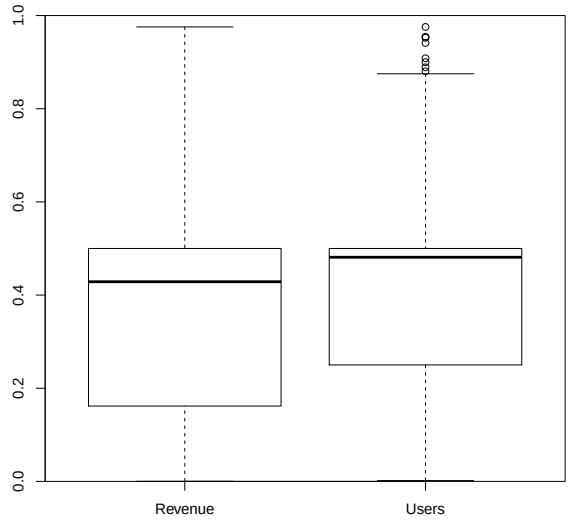

Figure 9: Box plot of lost revenue and lost users for our clone clusters.

the ad provider does not want to disclose how much it or its affiliated developers are making. Instead, we will show what percentage loss developers experience by comparing the ratio of impressions that belong to cloned applications compared with the total number of impressions we observed belonging to applications in our clone clusters. The percent revenue lost for a single cluster is calculated as:

$$PercImpsLost_{cluster} = \frac{\text{\# imps for clone apps}}{\text{sum of all impressions}} \quad (1)$$

Figure 7 shows a CDF of the percentage of lost revenue across our clone clusters. Notably, 28% of the clusters have exactly 50% lost revenue, which we determined resulted from clusters of two applications which had applications that shared a client ID, but not developer account or signature. Regardless, we can see that many of the clusters had a significant percentage of lost revenue. Figure 9 gives a boxplot of the lost revenue per cluster. Alarmingly, the mean percentage of lost revenue averaged across the $PercImpsLost_{cluster}$ values is 52%.

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

To show the total number of users of cloned apps, we will sum the number of unique users for every cloned app in every cluster. Note that this is the total of unique users for given apps, not a total number of unique people. If a user has n cloned apps on their phone, this method will count her as n users. We choose this counting method because different ad libraries create user

identifiers differently, making merging users across apps and ad libraries impossible. See [17] for a more thorough treatment of how ad libraries create user identifiers.

To demonstrate the average percent diverted user base we present a similar CDF as the one described in the previous section. For each cluster:

$$PercUsersLost_{cluster} = \frac{\#\text{ unique users for clone apps}}{\text{sum of all unique users}} \quad (2)$$

The CDF of these values is presented in Figure 8. Like lost impressions, the estimated percentage of lost users is alarmingly high, the mean of the $PercUsersLost_{cluster}$ values is 49%.

7. DISCUSSION

7.1 Challenges Determining Authorship

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 [4] 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

Previous work on illict behavior in Android applications includes privacy concerns from information leak within an application [7] and between applications [6], as well as privacy issues regarding third party libraries, especially advertising libraries [10, 15, 14, 17]. Additionally, research has been done investigating and characterizing Android malware [22]. Tools to detect cloned Android applications include DroidMOSS [21] and DNADroid [4].

Previous underground economy work primarily focuses on fraud which is detrimental to the user, such as fake anti-virus software [19], keyloggers [11], and spam [18]. By contrast, application cloning detriments the developers who originally created the application by taking away potential users of their application. Online advertising fraud has been heavily studied in the literature [5, 9, 20], however as long as users are still viewing the advertisements, application cloning does not necessarily imply advertising fraud.

9. CONCLUSION

As the first step towards understanding the economic incentives of application plagiarism on Android markets, we characterized application plagiarism and its impact on developers. Towards this goal, we crawled 304,275 free applications from 19 Android markets around the world and detected clones among them, 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 them. 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. 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

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

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

app-marketplaces-applanet-appbucket-piracy-sting/2012-08-22.

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

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

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] P. Pearce, A.P. Felt, G. Nunez, and D. Wagner. "AdDroid: Privilege separation for applications and advertisers in Android". In: *Proceedings of AsiaCCS*. 2012.

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

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

[17] 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).

[18] 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.

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

[20] 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.

[21] 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.

[22] 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 3: Mapping between meta-categories and market categories

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

.