Clone or Relative?: Understanding the Origins of Similar Android Apps

Yuta Ishii
Waseda University
Tokyo, Japan
yuta@nsl.cs.waseda.ac.jp

Takuya Watanabe
Waseda University
Tokyo, Japan
watanabe@nsl.cs.waseda.ac.jp

Mitsuaki Akiyama
NTT Secure Platform Laboratories
Tokyo, Japan
akiyama.mitsuaki@lab.ntt.co.jp

Tatsuya Mori
Waseda University
Tokyo, Japan
mori@nsl.cs.waseda.ac.jp

ABSTRACT

Since it is not hard to repackage an Android app, there are many cloned apps, which we call "*clones*" in this work. As previous studies have reported, *clones* are generated for bad purposes by malicious parties, e.g., adding malicious functions, injecting/replacing advertising modules, and piracy. Besides such *clones*, there are legitimate, similar apps, which we call "*relatives*" in this work. These *relatives* are not *clones* but are similar in nature; i.e., they are generated by the same app-building service or by the same developer using a same template. Given these observations, this paper aims to answer the following two research questions: (RQ1) *How can we distinguish between clones and relatives*? (RQ2) *What is the breakdown of* clones *and* relatives *in the official and third-party marketplaces*? To answer the first research question, we developed a scalable framework called APPraiser that systematically extracts similar apps and classifies them into *clones* and *relatives*. We note that our key algorithms, which leverage sparseness of the data, have the time complexity of $O(n)$ in practice. To answer the second research question, we applied the APPraiser framework to the over 1.3 millions of apps collected from official and third-party marketplaces. Our analysis revealed the following findings: In the official marketplace, 79% of similar apps were attributed to *relatives* while, in the third-party marketplace, 50% of similar apps were attributed to *clones*. The majority of *relatives* are apps developed by prolific developers in both marketplaces. We also found that in the third-party market, of the *clones* that were originally published in the official market, 76% of them are malware. To the best of our knowledge, this is the first work that clarified the breakdown of "similar" Android apps, and quantified their origins using a huge dataset equivalent to the size of official market.

Keywords

mobile security, Android, repackaging, large-scale data

IWSPA'16, March 11 2016, New Orleans, LA, USA

DOI: http://dx.doi.org/10.1145/2875475.2875480

1. INTRODUCTION

Android is an open-source operating system used for mobile devices such as smartphones. Android is one of the most popular mobile device platforms widely used in the world. Worldwide shipments of Android smartphones exceeded 1 billion units in 2014 [3]. The number of Android apps available on Google play has exceeded 1.8 million as of November 2015 [7]. Of the millions of Android apps that can work on a billion smartphones, it is known that a non-negligible number of apps were *replicated* from the original apps. For instance, through the analysis of 23K apps collected from six different third-party marketplaces, Zhou et al. [22] reported that 5 to 13 % of apps hosted on third-party marketplaces were repackaged. They also reported in Ref. [21] that "piggybacked apps", which added malicious payloads to legitimate apps, accounted for 0.97 to 2.7% of 85K apps they collected. In this work, we generally call those repackaged apps "*clones*". The high number of *clones* stems from the fact that repackaging an Android is not a hard task. In fact, there are several tools that can systematically repackage apps [1].

As previous studies have revealed [22, 21], many of the *clones* are created for malicious purposes, e.g., inserting advertising modules that were *not* present in the original version, replacing accounts used for ad libraries, and/or inserting a malicious code that steals privacy-sensitive information. While these *clones* add malicious payloads to the original apps, there is another class of *clones* — *pirated apps* — that illegally repackage/crack paid apps. The existence of these *clones* are harmful not only for end-users but also for many other stakeholders such as app developers, copyright holders, and marketplace providers. Besides *clones*, there are apps that are not *clones* but are *unintentionally* similar to each other, i.e., they have mostly similar appearances and behaviors. As we shall present in this paper, such similar apps originate from two categories: apps generated with app building frameworks/services and apps developed by the same developer, possibly with a fixed template. In this work, we generally call those unintentionally similar apps "*relatives*".

Both *clones* and *relatives* are the apps that are similar to other apps in nature. However, we need to distinguish between *clones* and *relatives* because the former apps are harmful and should be removed from marketplaces. Given these backgrounds in mind, this paper aims to answer the following two research questions through the analysis of Android apps in the wild:

RQ1: *How can we distinguish between* clones *and* relatives?

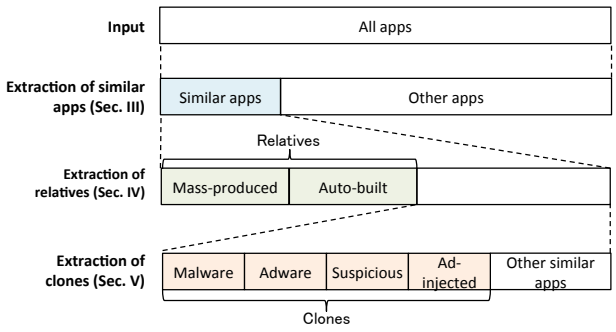


Figure 1: High-level overview of the APPraiser framework.

RQ2: *What is the breakdown of* clones *and* relatives *in the official and third-party marketplaces*?

As a solution to the first research question, we developed a lightweight framework called *APPraiser* that automatically extracts similar apps and classifies them into *clones*, *relatives*, and other subcategories. The key idea of the *APPraiser* framework is to adopt a three-stage strategy; it first extracts similar apps using the appearance analysis. It then extracts *relatives*, using several intrinsic fingerprints such as developer identities and application package names. Finally, it classifies *clones* using the code difference analysis and anti-virus checkers. To address the second research questions, we use the *APPraiser* framework to study over 1.3 million of apps collected from both official and third-party marketplaces. Analyzing apps published on two different types of markets enabled us to perform intra- and inter-market analysis of *clones* and *relatives*. We stress that although our approach has some limitations, which will be discussed in Sec. 7, good scalability of the *APPraiser* framework enabled us to perform the analysis to a million apps; thus, we can understand the entire picture of *clones* disseminated in the wild.

Our extensive analysis revealed the following findings: In the official marketplace, 79% of similar apps were attributed to *relatives* while, in the third-party marketplace, 50% of similar apps were attributed to *clones*. The majority of *relatives* are apps developed by prolific developers in both marketplaces. We also found that in the third-party market, of the *clones* that were originally published in the official market, 76% of them are malware.

The rest of this paper is organized as follows. Section 2 describes an overview of the *APPraiser* framework. Section 3, 4, and 5 describe the methodologies to extract similar apps, *relatives*, and *clones*, respectively. Section 6 presents key findings we obtained through the analysis of our dataset with the *APPraiser* framework. Section 7 discusses the limitations of the *APPraiser* framework and future research directions. We also discussed the possible countermeasures against malicious *clones*. Section 8 summarizes the related work. We conclude our work in section 9.

2. OVERVIEW OF THE APPRAISER FRAMEWORK

In this section, we describe the goal and overview of the *APPraiser* framework. The goal of the *APPraiser* is to extract *clones* and *relatives* from a given set of apps. The key challenge here is to cope with huge number of apps in a scalable manner. To meet this, the *APPraiser* adopts the three-stage strategy we describe in the followings.

Figure 1 depicts the high-level overview of the *APPraiser* framework. In the first stage, the APPraiser framework extracts similar apps, using the appearance analysis, which will be described in Sec. 3. The extracted similar apps are clustered according to the similarity measure. For each cluster, the APPraiser framework identifies the origin app by checking the meta data of the apps; e.g., ID number in the market, number of downloads, or published date, etc. In the second stage, the APPraiser framework extracts *relatives*, which compose of two categories; mass-production and auto-built, which are the apps generated by a prolific developer and the apps generated with app-building frameworks/services, respectively. The details of extracting *relatives* and the two categories will be given in Sec. 4. In the third stage, the APPraiser framework extracts and classifies *clones*, which compose of four categories; malware, adware, suspicious apps, and ad-injected apps. To this end, we adopt anti-virus checkers and code difference analysis. The details of extracting *relatives* will be given in Sec. 5. We also discuss the breakdown of remaining apps; i.e., "other similar apps" in Fig. 1.

3. EXTRACTION OF SIMILAR APPS

In this section, we describe how the *APPraiser* framework extracts similar apps. The key idea is to measure the differences between two apps by examining their appearances. The reason why we adopt appearance as a measure to extract similar comes from the following observation. When an app is intentionally cloned, its appearance is likely unchanged. For instance, because the objective of creating malicious *clones* is to attract end users by pretending to be an authentic one, there is no reason to change its appearance. For *relatives*, we empirically found that majority of apps have similar resources except the superficial appearance such as name of apps or app icons. Thus, we can assume that most of similar apps have the similar appearances with the original ones. In fact, several studies such as Ref. [20] and Ref. [16] adopted resource files in detecting similar apps. We note that while these studies used the same approach in detecting similar apps, they did not consider the difference between clones and relatives.

In the followings, we first describe the methodologies we used to extract information from Android app files. Next, we present the appearance analysis that extracts similar apps from a large number of apps. We also present how we aggregate the extracted similar apps into clusters.

3.1 Processing APK files

An Android app is packaged with a format called APK. An APK file is an archive that consists of developer certificate, manifest file, DEX file, and resource/asset files. Developer certificate can be used to extract information about the developer of an app. Manifest file consists of essential information about an app. For instance, it declares permissions to access resources. By carefully the Manifest file, we can check which permissions are added/removed from an original app. DEX file consists of Dalvik bytecode where Dalvik is a virtual machine that executes applications on Android OS. DEX format file can be disassembled by using a tool such as smali [15]. Again, by carefully analyzing smali code, we can check which API functions are added/removed from an original app. Resource file and asset file are used to control the appearance of an app. It consists of XML files that define the layout of screen, image files, sound files, and etc. The way how we use these information will be described below.

3.2 Appearance analysis

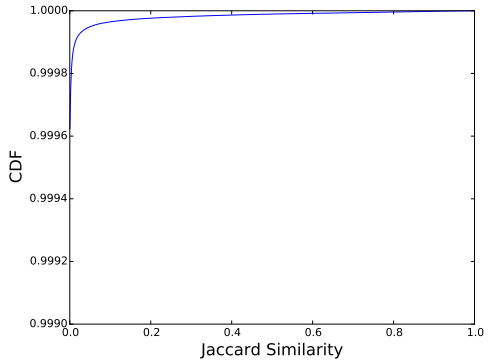


Figure 2: Jaccard index vs. cumulative fractions of app pairs.

While previous studies [20, 16] adopted resource files in detecting similar apps, we extend the approach by incorporating files in `assets` and `lib` folders, which also form the appearances of apps. We also develop a scalable algorithm that can extract similar objects from a large population.

In the followings, we present the procedure of extracting similar apps using the resource files. We first compute MD5 digest for each resource file. We then apply the DF-thresholding technique, which is widely used for text classification tasks [19]. By applying the DF-thresholding, we eliminate very popular resources that appear in a majority of the apps. These resources are too generic to measure the similarity between apps. Specifically, we introduced a threshold, K, and eliminated the top-K resources. We empirically derived the threshold as $K = 100,000$, which accounted for roughly 0.1% of all resources.

We now compute the appearance similarity between two apps, using the Jaccard index, which is a metrics used for computing the similarity of given two sets. Let a set of hash digests of an app x be $\mathbf{R}(x)$. Jaccard index is defined as follows. For apps a and b, the Jaccard index of the two apps is computed as $J(a,b) = \frac{|\mathbf{R}(a) \cap \mathbf{R}(b)|}{|\mathbf{R}(a) \cup \mathbf{R}(b)|}$. The Jaccard index takes a range between 0 and 1. If there are no common resource files between two apps, the Jaccard index becomes zero. If entire resource files are common between two apps, their Jaccard index becomes one. In extracting resource files, we made use of a tool called Androguard [5].

Figure 2 shows the relationship between the computed Jaccard index for all pairs and cumulative fraction of pairs. Note that number of all pairs is $N(N-1)/2$, which is much larger than the number of actually similar apps. We can see that majority of pairs have Jaccard index close to zero. In fact, more than 99.98% of pairs had Jaccard index of zero. We will leverage the sparseness in computing similarities between app pairs efficiently.

As a threshold to determine the similarity of two apps, we empirically adopt 0.8; i.e., if $J(a,b)$ for a given pair of apps a and b, we extract these two apps as similar apps. We note that the threshold is not so sensitive to our findings; i.e., other thresholds such as 0.7 and 0.9 did not affect our findings.

3.3 Fast algorithm to compute Jaccard index for all app pairs

A naive approach to extract apps that have high Jaccard index is to compute Jaccard index for pairwise combinations of all apps. Clearly, such approach is not scalable because its time complexity is $O(N^2)$, where $N \approx 1.3 \times 10^6$ for our dataset. We leverage the fact that data has sparseness; i.e., many of pairs do not have common resources and the Jaccard index is zero for such pairs. We denote a set of all applications $\mathbf{A}$. Let $\mathbf{I}(r)$ denote a set of applications that have a resource, r.

Algorithm 1: An algorithm to compute Jaccard index for all pairs in a set of applications, $\mathbf{A}$.

```
1  c(x,y) = 0                    /* a counter of a tuple (x,y) */
2  S = ∅                         /* a set to check entrance */
3  T = ∅                         /* will be used in Algorithm 2 */
4  U = ∅                         /* will be used in Algorithm 2 */
5  for ∀a ∈ A do
6      for ∀r ∈ R(a) do
7          for ∀b ∈ I(r) do
8              c(a,b) ← c(a,b) + 1
9              if (a,b) ∉ S then
10                 add (a,b) into S
11 for (a,b) ∈ S do
12     J(a,b) = c(a,b)/ (|R(a)| + |R(b)| − c(a,b))
13     if J(a,b) ≥ 0.8 then
14         if (a,b) ∉ T then
15             add (a,b) into T
16         if a ∉ U then
17             add a into U
```

Algorithm 2: A greedy clustering algorithm.

```
1  G(x) = ∅                      /* a set of items in cluster x */
2  while U ≠ ∅ do
3      x = random(U)
4      for ∀y such that (x,y) ∈ T do
5          add y into G(x)
6          remove y from U
7      remove x from U
```

An algorithm that computes Jaccard Index for all pairs is shown in Algorithm 1. Note that if $(a,b) \notin S$, the Jaccard index is $J(a,b) = 0$.

Now, we turn our attention to the time complexity of the algorithm. Because $\mathbf{R}(\cdot)$ is independent of $n = |\mathbf{A}|$, the algorithm has the time complexity of $O(|\mathbf{A}|\langle\mathbf{I}\rangle) = O(n\langle\mathbf{I}\rangle)$ where $\langle\mathbf{I}\rangle$ is the expected value of $|\mathbf{I}(r)|$, i.e., $\langle\mathbf{I}\rangle = \frac{1}{|\mathbf{R}|}\sum_{r\in\mathbf{R}}|\mathbf{I}(r)|$, where $\mathbf{R}$ is a set of all resource files. In theory, the worst case time complexity is $O(n^2)$ where $|\mathbf{I}(r)| = n$ for all r; which implies that all the apps are identical. Clearly, such assumption is unrealistic. In practice, thanks to the sparseness of the data, in most cases, $|\mathbf{I}(r)| = 1$. In the case of our dataset with $n = O(10^6)$, the expected value was $\langle\mathbf{I}\rangle = 1.72$. Thus, our algorithm works with the time complexity of $O(n)$ if it is applied to data with sparse structure.

3.4 Clustering similar apps and identifying the origin app in a cluster

Using the Algorithm 1, we have extracted app pairs, $\mathbf{T}$, which have the Jaccard index larger than 0.8. Now, we aggregate the apps into clusters, using the greedy clustering algorithm shown in Algorithm 2. Let random($\mathbf{X}$) be a function that returns a randomly selected element in a set $\mathbf{X}$. Note that $\mathbf{T}$ and $\mathbf{U}$ have been computed with the Algorithm 1. Hence, the clustering algorithm is light-weight and works fast.

We note that the obtained clusters are not always optimized. However, through the several trials using different random seeds, we empirically validated that the obtained results are not sensitive to our key findings. Because our objective was to study the origins

of similar apps in the wild, we decided to choose the better scalability rather than the better accuracy. We further discuss the issue in Section 7.

Finally, for each cluster $\mathbf{G}(x)$, we identify an original app with the following criteria: For the official market, we consider an app is the original if it has the maximum number of downloads among the apps in a cluster. The rule is formulated as: For the third-party market, we make use of the ID of apps as a heuristic to that market. Since IDs are sequentially incremented, in the group of similar apps, the app with the least ID is likely an original app. We note that these approaches could fail if the actual original app is missing in our data; i.e., all the apps in a cluster could be all relatives or clones. We will discuss the issue in Section 6.

4. EXTRACTION OF RELATIVES

This section describes how the *APPraiser* framework extracts relatives apps. The key idea is to apply fingerprints that indicate apps are generated by a prolific, identical developer or generated with an application generation framework/service. It is natural that apps developed by a same person are not *clones* in our context. The *clones* we consider are apps that are developed by an outsider who is not associated with the author of original app(s). In the followings, we present the details of each category and how the *APPraiser* framework extracts them.

4.1 Mass-produced apps

It has been reported that there are a few prolific developers who publish a large number of apps [18]. We observed that apps published by such developers tend to be similar to each other. Although not conclusive, we conjecture that such prolific developers need to use a same template, which include common resources, to publish a large number of apps in a short period of time. Also, outsourcing companies that develop Android apps may use a same template or even develop their own app developing framework to generate apps quickly. Use of same template or same add developing framework may introduce some similarity between apps developed. Let us call such apps *mass-produced apps*.

Information about a developer can be obtained from two channels: developer certificate and developer name. A developer certificate can be extracted from an APK file. The format of a digital certificate is X.509 v3. We extract a public key from the given certificate, and use it as a fingerprint. We note that a developer may use different pairs of secret/public keys for signing certificates. To cope with such a case, we relax the condition; we extract key features of a subject from the given certificate. Namely, we generate a tuple, organization name (O) and locality (L), and use it as a fingerprint. Furthermore, developers in an organization such as app developing company may use distinct certificates that are not associated with each other. To cope with such a case, we further relax the condition; we use a developer name, which can be extracted from the app's meta data published on a marketplace.

In summary, to extract mass-produced apps, we obtain certificate and developer name for each app. Next, if there are at least two apps that have exactly same public keys, same subjects of certificates, or developer names, we extract the apps as mass-produced. We will illustrate examples of *mass-produced* apps in Section 6.

4.2 Auto-built apps

There are several cloud-based app building services such as iBuild App [11] or Bizness Apps [9]. These services provide an intuitive web interface and enable a developer to generate a multi-platform app without writing codes for it. In this work, we call apps developed with such services *Auto-built apps*. It is known that *Auto-built* apps tend to unnecessarily install many permissions, and put callable APIs for the permissions into the codes [18]. *Auto-built* apps also tend to be shipped with common resources even though many of them are *not* used. Thus, resources and code of *Auto-built* apps resemble to each other even though they are independently developed by different developers.

Table 1: A list of app building services and their fingerprints.

App building service	fingerprint
Andromo	andromo
Appery.io	appery
appexpress	appexpress
AppMachine	artistapp
Apps Bar	appsbar
AppsBuilder	appsbuilder
Appy Pie	appypie
Bizness Apps	app_***.layout
como	.conduit.
GoodBarber	goodbarber
iBuild APP	appbuilder
MIT App Inventor	appinventor
ReverbNation	reverbnation
vBulletin Mobile Suite	vbulletin

By analyzing the frequencies of package names of apps, we were able to compile a list of such services. Table 1 lists the compiled services and the corresponding fingerprints that are derived from intrinsic keywords included in the package names. Using Table 1, we can extract *Auto-build* apps.We will illustrate examples of *Auto-build* apps in Section 6. We note that this approach clearly has a limitation; i.e., if an app building service provides arbitrary package names, this approach fails. Although the approach seems to work well for the current popular services, we might need to address such cases in future. We envision that app building services should leave some form of footprints in their artifacts.

5. EXTRACTION/CLASSIFICATION OF CLONES

This section describes how the *APPraiser* framework extracts *clones* from the remaining similar apps and classifies them into four categories; malware, adware, suspicious apps, and ad-injected apps. Note that because our dataset is composed of only free apps, we cannot extract another type of clone — a pirated app that cracked an original *paid* app.

5.1 Extraction of malware and adware

We first extract two categories of malicious clones, malware and adware. Our assumption is as follows. If an app B is likely repackaged from a legitimate original app A and the app B is detected as malware/adware, we consider the app B is a malicious clone of the app A. On the basis of this assumption, we first check whether a given similar app is malware or adware. We note that we detect malware/adware clones only if their origin app is legitimate; i.e., the origin app was not classified as malware/adware.

Because the aim of this work is not to propose a new method that detects new malware/adware, we adopt straightforward approach to extract them. We apply VirusTotal [17], which is online anti-virus service that composes of more than 60 different commercial anti-virus checkers. All the remaining similar apps are applied to the VirusTotal. For a given app, if at least one of the anti-virus checkers detect the app as malware, we consider that the app is a malicious clone (malware). If an app is not detected as malware and at least one of the anti-virus checkers detect the app as adware, we consider that the app is a malicious clone (adware).

We note that VirusTotal may introduce detection errors. In addition, we cannot prove that detected malware and adware apps are actually repackaged from the original ones. As we shall in

Table 2: List of dangerous permissions.

Permissions	
ACCESS_FINE_LOCATION	SEND_SMS
ACCESS_COARSE_LOCATION	READ_SMS
ACCESS_LOCATION_EXTRA_COMMANDS	RECEIVE_SMS
READ_LOGS	WRITE_MEDIA_STORAGE
INSTALL_SHORTCUT	RESTART_PACKAGES
SYSTEM_ALERT_WINDOW	INSTALL_PACKAGES
SYSTEM_OVERLAY_WINDOW	ACCESS_WIFI_STATE
RECEIVE_BOOT_COMPLETED	DISABLE_KEYGUARD
CHANGE_NETWORK_STATE	READ_CONTACTS
DOWNLOAD_WITHOUT_NOTIFICATION	READ_PHONE_STATE
MOUNT_UNMOUNT_FILESYSTEMS	

short, however, our manual inspection using randomly sampled apps validated the accuracy of the approach. Therefore, we believe that potential errors due to some limitations, which are made to achieve high scalability, may not affect the overall findings we derived from the analysis. Furthermore, we introduce the following two categories that can catch potential malware/adware that could be missed by VirusTotal.

5.2 Extraction of suspicious apps/ad-injected apps

Next, we extract two other categories: suspicious apps and ad-injected apps, which are aimed to cover malware and adware that are not detected by anti-virus checkers, respectively. After employing VirusTotal, we perform the static code analysis. The *APPraiser* framework extracts and analyzes the following features; i.e., permissions, API calls associated with privacy-sensitive permissions, and FQDN used for ad-libraries. These features are extracted from the Manifest file or disassembled DEX file. We then check the differences of features between the two given apps, *A* and *B*, which represent origin and the app similar to the origin, respectively.

Table 2 lists the dangerous permissions, which could be added to an app *A*. If an app *B* adds at least one of the permissions listed in Table 2, and the added permission was not present in the app *A*, we consider that the *B* is *suspicious*. We also check APIs. If an app *B* adds at least one of the APIs associated with the permissions listed in Table 2, and the added API function was not present in the app *A*, the app *B* is considered as *suspicious*. Here, we made use of the API calls for permission mappings extracted by a tool called PScout [14], which was developed by Au et al. [8]. To check the existence of APIs, we checked whether a set of APIs is included in the disassembled code of an APK file.

Similarly, we check whether an app *B* adds a new FQDN associated with ad-library. Let denote such FQDN as ad-FQDN. The key idea of our approach is to make use of a list of ad-FQDNs that are compiled to block network communications invoked by ad libraries. We first collected such list of ad-FQDNs from popular ad-block sites such as AdAway [4]. We then pruned FQDNs that were clearly wrong records such as `schema.android.com`. In total, number of ad-FQDNs we compiled was $1,027$. Finally, we explore disassembled codes of apps, and check ad-FQDNs. If an app *B* adds at least one ad-FQDN, which was not present in the app *A*, the app *B* is considered as *ad-injected*.

6. ANALYSIS

In this section, we present our key findings through the analysis of huge number of Android apps in the wild. We first illustrate the data we used for our analysis. Finally, we try to answer **RQ2** by applying the *APPraiser* framework to the entire data set. Finally, we demonstrate the validity of our methodology using randomly sampled APK files.

Table 3: Summary of Android apps used for this work.

marketplace	# of APK files	Data collection periods
Google Play	1,296,537	Oct 2014
Anzhi	74,185	Nov 2013 – Apr 2014
Total	1,370,722	–

Table 4: Numbers/fractions of detected similar apps.

	Google Play	Anzhi
Similar apps	78,919 (6.1%)	19,206 (25.9%)

Table 5: Breakdown of similar apps.

	Google Play	Anzhi
relatives	62,164 (78.8%)	8,121 (42.3%)
clones	6,076 (7.7%)	9,545 (49.7%)
unknown	10,679 (13.5%)	1,540 (8.0%)

Table 6: Breakdown of relatives.

	Google Play	Anzhi
Mass-produced	55,722 (89.6%)	8,121 (100.0%)
Auto-built	6,442 (10.4%)	0 (0.0%)

6.1 Data

We collected Android apps from the official marketplace [13] and a third-party marketplaces [6]. Both of these marketplaces have huge user bases. Note that these were all free apps. Although we might see some disparity between free and paid apps, we leave this issue open for future research. For Android apps published on official market in particular, we made use of the data presented in Ref. [16]. Since the original dataset included versions of an app, we adopt only the latest version for a given app. We also eliminate apps that are likely corrupted for some reasons.

Using the data, we can study the qualitative differences between the two types of marketplaces, official market and third-party market. It has been reported that the official marketplace has installed a special defense mechanisms called Bouncer [12]. Therefore, as previous studies have reported, official market tends to have less number of malicious apps, compared to those in third-party markets [22]. It is noteworthy that in China, which is a country with the highest population, official Google Play market has been unavailable. Therefore, people who hope to enjoy popular apps published in Google Play may have incentive to import the clones into third-party market. In fact, as Zhou et al. [22] reported, 5 to 13 % of apps hosted on third-party marketplaces were repackaged. In this work, we will study the differences of two types of markets with a lens of similar apps.

6.2 Classification of apps and their properties

As an answer to **RQ2**, we now present the results of extraction/classification of apps, using the *APPraiser* framework. First, Table 4 shows the numbers/fractions of detected similar apps in each market. As we expect, the fraction of the similar apps is much higher in the third-party market; this observation generally agrees with the previous reports. We note that even in the official market, non-negligible numbers of apps are categorized into similar apps.

Next, Table 5 shows the breakdown of the detected similar apps. We first notice that the fraction of *relatives* is significantly high in Google Play. The result indicates that most of similar apps detected with the resource-based approach are attributed to *relatives*, but not *clones*, which should require more attention. Our framework, *APPraiser*, enabled us to systematically distinguish the two categories. We also notice that the fraction of *clones* in Anzhi is much higher than that in Google Play. Again, the observation gen-

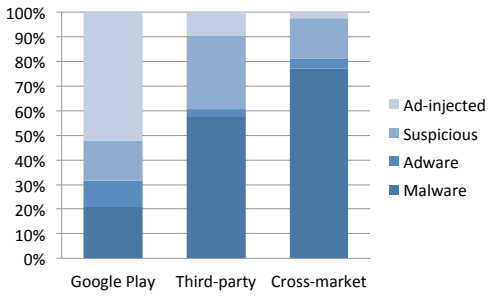


Figure 3: Breakdown of the clones.

Table 7: Mean and median of # downloads of origin apps.

	mean	median
All	37,439	50 ~ 100
clones	364,329	10,000 ~ 50,000

erally agrees with the previous reports. The further breakdowns of these categories will be shown soon.

Table 6 shows the breakdown of *relatives*. Clearly majority of *relatives* is attributed to *Mass-produced*; i.e., prolific developers tend to publish many similar apps, possibly using a same template. In this work, we were not able to find out popular app building services for the third-party market. As a result, number of *Auto-built* apps in the third-party market was zero. We need to come up with other heuristics to detect app-building services popular in the third-party marketplace. We leave the issue for our future study.

Figure 3 shows the breakdown of *clones*, where we excluded the origin apps. As a cross-market analysis, we consider the case where apps published on the official market were repackaged, and published on the third-party market. For the official marketplace, roughly half of clones were attributed to *Ad-injected* while the fraction of malware is around 20%. This may correlate to the existence of defense mechanisms installed on the official marketplace – Bouncer [12]; i.e., in the official market, fraction of malware is kept fairly lower than the third-party marketplaces. We can also observe that roughly 70% of clones were *not* detected by commercial anti-virus checkers; thus our code analysis worked effectively in catching such potential malware/adware. We will present examples of those apps later. For the third-party marketplace, roughly 60% of clones were attributed to *Malware*. Furthermore, for the cross-market, roughly 80% of clones were attributed to *Malware*. This implies that majority of malicious *clones* found in the third-party market repackaged apps originally published on the official market.

We study which categories of apps are more likely cloned. Figure 4 presents the distributions of apps per category, which is defined in the official marketplace. We notice that while the distributions are mostly similar among three types of apps: all apps, similar apps, and clones, clones are more likely repackaged from Game apps. We conjecture that the authors of clones tend to repackage popular apps. In fact, among all apps, the Game category was the most popular. The results shown in Table 7 also support the conjecture. That is, average/median number of downloads for the origin apps that were cloned is higher than total average/median. Note that on Google Play, number of downloads is expressed with the discretized ranges; e.g., 0 ~ 10, 10 ~ 50, 50 ~ 100, etc. Thus, clones tend to target more popular apps so that they can attract victims.

6.3 Validity of extracted/classified clones

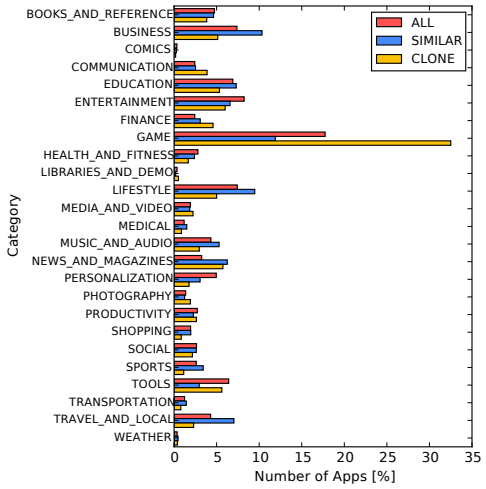


Figure 4: Distributions of apps per category (Google Play).

Table 8: Accuracies of clone detection.

clone category	# of classified apps	# of actual clone apps
malware	30	29
adware	30	25
suspicious	30	23
ad-injected	30	28
Total	120	106

While the classification accuracy of *relatives* should be high because we use intrinsic signatures to detect them, we need to validate the classification accuracy of *clones*. Since there are no ground-truth database, we validate the accuracy through manual inspection, which include in-depth static analysis and dynamic analysis. Of the samples that were classified as *clones*, we randomly picked up 30 samples for each category of *clones*, i.e., malware, adware, suspicious, and ad-injected. In total, we picked up 120 samples for validation. We then checked whether the 120 samples were actually clones with manual inspection. Table 8 summarizes the results. As we see, the accuracies were generally good over the categories. We further analyzed the falsely classified samples carefully and found that many of them should have been classified as *relatives*. Such apps used common, but minor UI frameworks that made them look similar in their code bases. It is an another issue that we need to address in our future work. Other than the small number of errors, the classification of *clones* worked successfully.

In the following, we picked up typical samples for each category, and present what we found through the manual inspection.

6.3.1 *Malware*

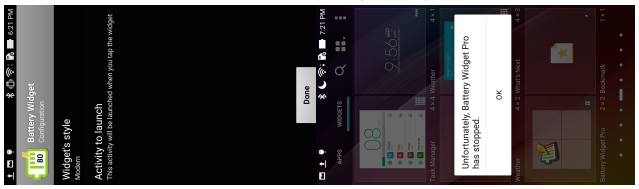
We show two samples here. The first example shown in Fig. 5(a) is taken from cross-market clone (malware). It clearly adds a large advertisement windows on the initial screen of a game app. Furthermore, it asks to install an additional app. In the second example shown in Fig. 5(b), number of downloads for the origin app was 10,000 ~ 50,000 while that for clone was 50 ~ 100. As shown in the screenshots, the clone app was unexpectedly quit soon after it launched. Both origin and clone were still available on the market as of July 2015.

6.3.2 *Adware*

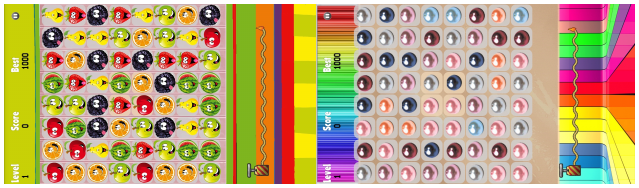
The example is shown in Fig. 5(c). The clone was repackaged from a puzzle game app. Although the clone uses different icons and images, the structure of app was identical. The clone has been removed from the marketplace.

(a) Malware (cross market)

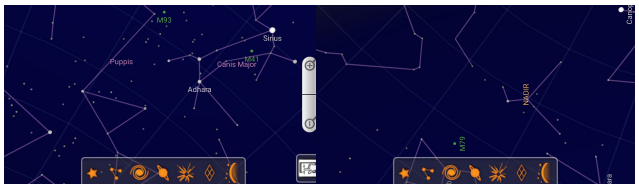
(b) Malware (Google Play)



(c) Adware (Google Play)



(d) Suspicious (Google Play)



(e) Ad-injected (Google Play)

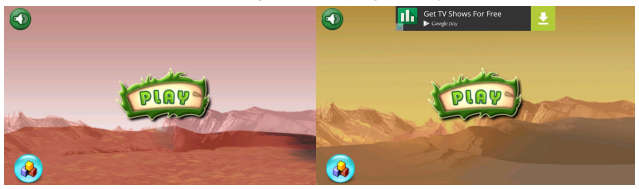


Figure 5: Screenshots: original apps (left) and clones (right).

6.3.3 Suspicious

The example shown in Fig. 5(d). An app for exploring constellations. Although appearance looks identical, the suspicious clone added the following new permissions that were not present in the origin app: ACCESS_WIFI_STATE, GET_TASKS, READ_PHONE_STATE, RECEIVE_BOOT_COMPLETED, and WRITE_EXTERNAL_STORAGE. The clone also adds several additional APIs such as `getDeviceId()` and new additional services such as `PushMessageService`. The clone has been removed from the market.

6.3.4 Ad-injected

The example shown in Fig. 5(e). As shown in the screenshots, advertisement modules that were not present in the original version were added in the clone, which was not detected as adware by anti-virus checkers. The clone has been removed from the market.

7. DISCUSSION

In this section, we discuss several limitations of the *APPraiser* framework. We also outline several future research directions that can help extend our framework. First, to cope with the high volume of data, we adopt a simple algorithm to find similar apps. Therefore, clusters generated by the algorithm are not always optimized. In our future work, we will try some scalable clustering algorithms and see whether we see some difference. Second, because we limit our analysis on free apps, we were not able to find pirate apps that cracked paid apps. To fully understand the problems of app theft with clones, we may need to shed lights on paid apps as well. We leave the issue for future study. Finally, as we mentioned earlier, we adopt an approach of using anti-virus checkers to detect malware and adware. However, the use of anti-virus checker is prone to detection errors. Here, we note the case where anti-virus checker is useful in finding malicious clones, which are difficult to find otherwise. In the third-party marketplace, we observed that non-negligible number of malicious clones are encrypted using a tool called SecAPK [2], which encrypts bytecode to evade reverse engineering. In our dataset, all the apps encrypted with the SecAPK were detected as malicious with VirusTotal. There are no clear reasons that a developer, who is not associated with the author of the original app, repackaged an originally legitimate app using such an encryption tool. Therefore, the detected apps encrypted with SecAPK are likely malicious clones if it originates from a legitimate app developed by other author.

8. RELATED WORK

There have been several studies that work on analyzing *similar* Android apps. They are broadly classified into two categories: code-base approaches and resource-based approaches. We present an overview of studies for each category. We also discuss the differences between the previous studies and ours.

8.1 Code-based approach

DroidMOSS [22] is a framework that detects repackaged apps. The key idea was to make use of opcode in the disassembled code. It uses the features derived from opcode to detect repackaged apps by leveraging fuzzy hashing in calculating the edit distance between apps. Since the analysis requires pairwise computation, it has the time complexity of $O(n^2)$. DNADroid [10] is a framework that detects cloned apps. It makes use of program dependency graph (PDG) to characterize an app. The framework compares PDGs between methods in a pair of apps. Again, since the analysis requires pairwise computation, it has the time complexity of $O(n^2)$. PiggyApp [21] is a framework that detects "piggybacked app", which is a repackaged app that injects new malicious code into the original app. The key idea of the PiggyApp framework was to adopt a technique called module decoupling, which partitions the app code into primary and non-primary modules. They also proposed a scalable approach that extracts semantic features from the decoupled primary modules. The approach has the time complexity of $O(n \log n)$.

In general, the computation cost of code-based approaches is high. For instance, although the time complexity of PiggyApp is $O(n \log n)$ with respect to the number of apps to be analyzed, module decoupling requires additional computation costs in constructing PDG for each app. Due to the high computation cost, the numbers of apps analyzed with these approaches are limited; i.e., $n = 68,817$ for DroidMOSS, $n = 75,000$ for DNADroid, and $n = 84,767$ for PiggyApp. Another limitation of the code-based approach is that it is difficult to cope with the obfuscated/encrypted apps. Based on these observations, the resource-based approach has attracted attentions because it can detect similar apps with low cost and is not affected with code obfuscation/encryption. We will summarize such works in the next subsection.

8.2 The Recourse-based approach

Viennot et al. [16] developed a system called PlayDrone, which efficiently crawls the official Google Play Store. Using roughly 1 million of apps collected with PlayDrone, they performed various analysis of Android apps, including the analysis of similar apps. To this end, they used resources as a feature to search apps that are similar to each other. They revealed that roughly 25% of apps had duplicated content in various reasons such as application re-branding or application cloning. Yury et al. [20] proposed a framework called FSquaDRA, which detects similar apps using resource information. They aimed to speed up hash calculations of resources by leveraging SHA1 digest of each file that are included in the Manifest file. They evaluated the effectiveness of their approach using $n = 55,779$ apps. The time complexity of the algorithm was $O(n^2)$.

8.3 Key differences between past studies and ours

As we presented earlier, our framework *APPraiser* combined both the code-based and resource-based approaches. This idea enabled us to establish *high scalability* with the resource-based approach and fast algorithm and *fine-grained analysis* of similar apps with the code-based analysis. Our key algorithms, which leveraged sparseness of the data, had the time complexity of $O(n)$ and worked efficiently over $n = 1,370,000$ apps. We also categorized similar apps into two primary categories: relatives and clones, which are further sub-categorized. Such detailed categorization clarifies the actions we need to take against similar apps.

9. SUMMARY

In this paper, we aimed to answer the following two research questions: (**RQ1**) *How can we distinguish between clones and relatives*? (**RQ2**) *What is the breakdown of clones and relatives in the official and third-party marketplaces*? Our solution to the first research question was achieved with the *APPraiser* framework that systematically extracts similar apps and classifies them into *clones* and *relatives*. The key idea of the APPraiser framework was to adopt three-stage strategy; (1) extraction similar apps using the appearance analysis, (2) extraction of *relatives* using several intrinsic fingerprints, and (3) extraction and classification of *clones*, using the outcomes of anti-virus checkers and code difference analysis. To answer the second research question, we applied the *APPraiser* framework to the over 1.3 millions of apps collected from official and third-party marketplaces. Our key findings are summarized as follows: In the official marketplace, 79% of similar apps was attributed to *relatives* while, in the third-party marketplace, 50% of similar apps was attributed to *Clones*. Majority of *Relatives* are apps developed by prolific developers in both marketplaces. We also found that in the third-party market, of the *clones* that were originally published in the official market, 76% of them are malware.

The key contributions of this work can be summarized as follows: First, we clarified the breakdown of "similar" Android apps with the notion of *clones* and *relatives*. Such clarification enables us to take proper actions against apps with content duplications. Second, we quantified the origins of similar apps using over 1.3 million of Android apps, which is equivalent to the size of official market. To perform such a huge-scale analysis, we also developed light-weight algorithms that can extract similar items from a huge, sparse dataset with the time complexity of $O(n)$.

10. REFERENCES

[1] Fake apps: Feigning legitimacy. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fake-apps.pdf.

[2] The gray-zone of malware detection in android os. https://blog.avast.com/2014/03/31/the-gray-zone-of-malware-detection-in-android-os/.

[3] Stragety analytics. https://www.strategyanalytics.com/strategy-analytics/blogs/devices/smartphones/smartphones/2015/03/11/android-shipped-1-billion-smartphones-worldwide-in-2014, Jan. 2015.

[4] AdAway. http://adaway.org/hosts.txt.

[5] Androguard. https://code.google.com/p/androguard/.

[6] anzhi.com. http://www.anzhi.com/.

[7] AppBrain. Android operating system statistics. http://www.appbrain.com/stats/.

[8] K. Au, W. Yee, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proc. of ACM CCS*, pages 217–228, 2012.

[9] Bizness Apps. https://www.biznessapps.com/.

[10] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proc. of the 17th European Symposium on Research in Computer Security*, pages 37–54, 2012.

[11] iBuildApp. http://ibuildapp.com/.

[12] J. Oberheide and C. Miller. Dissecting the android bouncer. SummerCon, Brooklyn, NY., 2012.

[13] G. Play. http://play.google.com/.

[14] PScout. Analyzing the Android Permission Specification. http://pscout.csl.toronto.edu/.

[15] smali. https://code.google.com/p/smali/.

[16] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. Proc. of ACM SIGMETRICS 2014, June 2014.

[17] VirusTotal. https://www.virustotal.com/.

[18] T. Watanabe, M. Akiyama, T. Sakai, H. Washizaki, and T. Mori. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *Symposium on Usable Privacy and Security (SOUPS)*, 2015.

[19] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 412–420, 1997.

[20] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina, and E. Moser. Fsquadra: Fast detection of repackaged applications. Proc. of IFIP DBSec '14, pages 131–146, 2014.

[21] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proc. of the third ACM CODASPY 2013*, pages 185–196.

[22] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the second ACM CODASPY 2012*, pages 317–326.