PAPER Special Section on Information and Communication System Security

APPraiser: A Large Scale Analysis of Android Clone Apps*

Yuta ISHII[†a)], **Takuya WATANABE**[††b)], ***Nonmembers***, **Mitsuaki AKIYAMA**[††c)], ***and*** **Tatsuya MORI**[†d)], ***Members***

SUMMARY Android is one of the most popular mobile device platforms. However, since Android apps can be disassembled easily, attackers inject additional advertisements or malicious codes to the original apps and redistribute them. There are a non-negligible number of such repackaged apps. We generally call those malicious repackaged apps "*clones*." However, there are apps that are not *clones* but are similar to each other. We call such apps "*relatives*." In this work, we developed a framework called *APPraiser* that extracts similar apps and classifies them into *clones* and *relatives* from the large dataset. We used the *APPraiser* framework to study over 1.3 million apps collected from both official and third-party marketplaces. 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.

key words: *mobile security, Android, repackaging, large-scale data*

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 [2]. The number of Android apps available on Google play has exceeded 2.3 million, as of August, 2016 [3]. 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. [4] reported that 5 to 13% of apps hosted on third-party marketplaces were repackaged. They also reported in Ref. [5] that "piggybacked apps," which added malicious payloads to legitimate apps, accounted for 0.97% to 2.7% of 5K 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 [6].

As previous studies have revealed [4], [5], 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* is 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 in nature to other apps. However, we need to distinguish between *clones* and *relatives* because the former apps are harmful and should be removed from the marketplace. Given this background, this paper aims to answer the following two research questions through the analysis of Android apps:

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*?

As a solution to the first research question, we developed a light-weight framework called *APPraiser* that automatically extracts similar apps and classifies them into *clones*, *relatives*, and other sub-categories. 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* by using several intrinsic fingerprints, such as developer identities and application package names. Finally, it classifies *clones* using the code difference analysis and antivirus checkers. To address the second research questions, we use the *APPraiser* framework

Manuscript received September 7, 2016.
Manuscript revised January 29, 2017.
Manuscript publicized May 18, 2017.

[†]The authors are with the Dept. of Communication Engineering, Waseda University, Tokyo, 169–8555 Japan.
[††]The authors are with the NTT Secure Platform Laboratories, Musashino-shi, 180–8585 Japan.

*An earlier version of this paper was presented at 2nd ACM International Workshop on Security And Privacy Analytics 2016 [1]. The authors will clear the copyright transfer issues before the publication in case the paper is accepted for publication.

a) E-mail: yuta@nsl.cs.waseda.ac.jp
b) E-mail: watanabe.takuya@lab.ntt.co.jp
c) E-mail: akiyamam@acm.org
d) E-mail: mori@nsl.cs.waseda.ac.jp

DOI: 10.1587/transinf.2016ICP0012

to study over 1.3 million 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 analyses of *clones* and *relatives*. We stress that although our approach has some limitations, which will be discussed in Sect. 7, good scalability of the *APPraiser* framework has enabled us to perform the analysis on 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% are malware.

The rest of this paper is organized as follows. Section 2 presents an overview of the *APPraiser* framework. Sections 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 Sect. 9.

2. Overview of the APPraiser Framework

In this section, we describe the goal and present an 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 a huge number of apps in a scalable manner. To meet this, the *APPraiser* adopts the three-stage strategy we describe below.

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 Sect. 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 metadata of the apps, e.g., the ID number in the market, the number of downloads or published date, etc. In the second stage, the *APPraiser* framework extracts *relatives*, which are composed of two categories; mass-production and auto-built (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 Sect. 4. In the third stage, the *APPraiser* framework extracts and classifies *clones*, which are composed of four categories; malware, adware, suspicious apps, and ad-injected apps. To this end, we adopt antivirus checkers and code difference analysis. The details of extracting *relatives* will be given in Sect. 5. We also discuss the breakdown of the remaining apps, i.e., "other similar apps" in Fig. 1.

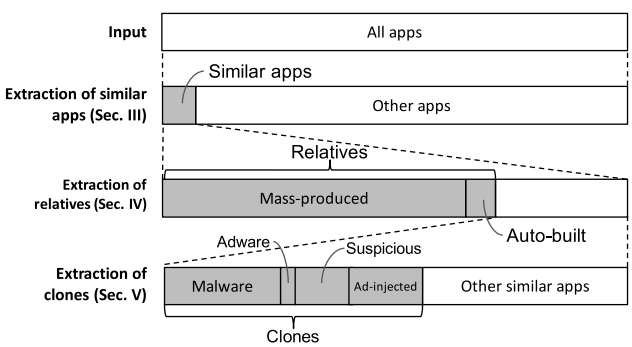


Fig. 1 High-level overview of the APPraiser framework.

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 apps 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 a majority of apps have similar resources except for the superficial appearance such as the name of apps or app icons. Thus, we can assume that most similar apps have the similar appearances to the originals. In fact, several studies such as Ref. [7] and Ref. [8] 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. The Manifest file consists of essential information about an app. For instance, it declares permissions to access resources. By carefully analyzing the Manifest file, we can check which permissions are added/removed from an original app. The DEX file consists of Dalvik bytecode where Dalvik is a virtual machine that executes applications on Android OS. The DEX format file can be disassembled by using a tool such as smali [9]. Again, by carefully analyzing smali code, we can check which API functions are added/removed from an original

app. The resource file and asset file are used to control the appearance of an app. These consist of XML files that define the layout of the screen, image files, sound files, and etc. The way we use all this information will be described below.

3.2 Appearance Analysis

In the following, we present the procedure of extracting similar apps using the resource files. We first compute the MD5 digest for each resource file. In this work, we consider the files in `assets` and `lib` folders also as the resource files. We then apply the DF-thresholding technique, which is widely used for text classification tasks [10]. By applying it, 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 by 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 $\mathbf{R}(x)$. 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 [11].

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

As a threshold to determine the similarity between two apps, we empirically adopt 0.8; i.e., if $J(a,b)$ for a given pair of apps is 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 the Jaccard Index for All Pairs

A naive approach to extract similar apps is to compute the 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 the 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. An algorithm that computes the 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. The algorithm has the time complexity of $O(|\mathbf{A}|\langle\mathbf{R}\rangle\langle\mathbf{I}\rangle) = O(n\langle\mathbf{R}\rangle\langle\mathbf{I}\rangle)$, where $\langle\mathbf{R}\rangle$ is the expected value of $|\mathbf{R}(a)|$ and $\langle\mathbf{I}\rangle$ is the expected value of $|\mathbf{I}(r)|$, respectively. They can be computed as $\langle\mathbf{R}\rangle = \frac{1}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}|\mathbf{R}(a)|$and $\langle\mathbf{I}\rangle = \frac{1}{|\mathbf{R}|}\sum_{r\in\mathbf{R}}|\mathbf{I}(r)|$,where $\mathbf{R}$ is a set of all resource files. In theory, the worst case time complexity is $O(n^2\langle\mathbf{R}\rangle)$, 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, $|\mathbf{I}(r)| = 1$.

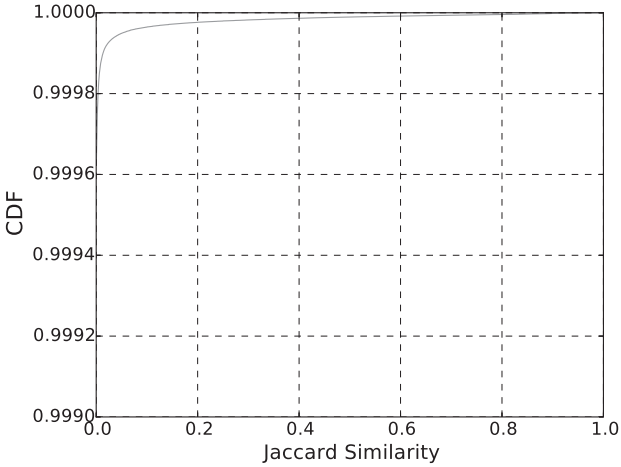


Fig. 2 Jaccard index vs. cumulative fractions of app pairs.

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
18         if b ∉ U then
19             add b into U
```

Algorithm 2: A greedy clustering algorithm.

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

In fact, in the case of our dataset with $n = O(10^6)$, the expected value was $\langle \mathbf{I} \rangle = 1.72$. Note that we have already eliminated the top-K resources as described in Sect. 3.2. In addition, the average number of resource files for the APK was $\langle \mathbf{R} \rangle = 140.2$, which is not directly associated with n. Thus, our algorithm works with the time complexity of $O(n)$ in practice if it is applied to data with sparse structure.

3.4 Clustering Similar Apps and Identifying the Origin App in a Cluster

Using the greedy clustering algorithm shown in Algorithm 2, which is equivalent to the special case of the DBSCAN algorithm [12], we aggregate apps into clusters. We note that the obtained clusters are not always optimized. This comes from the fact that the DBSCAN algorithm is not deterministic, i.e., the clustering result can depend on the order of the samples. However, through 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 Sect. 7.

Finally, for each cluster $\mathbf{G}(x)$, we identify an original app with the following criteria: For the official market, we consider that an app is original if it has the maximum number of downloads among the apps in a cluster. For the third-party market, we make use of the ID of apps as a heuristic to that market. Since ID's are sequentially incremented, in the group of similar apps, the app with the least ID is likely to be 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.

4. Extraction of Relatives

This section describes how the *APPraiser* framework extracts relative 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 the same person are not *clones* in our context. We consider *clones* to be apps that are developed by an outsider who is not associated with the author of the original app(s). In the following, 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 [13]. We observed that apps published by such developers tend to be similar to each other. Although this is not conclusive, we conjecture that such prolific developers need to use the same template, which includes common resources, to publish a large number of apps in a short period of time. Also, outsourcing companies that develop Android apps may use the same template or even develop their own app-developing framework to generate apps quickly. Use of the same template or the same app-developing framework may introduce some similarity between the 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 an 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 metadata published on a marketplace.

In summary, to extract mass-produced apps, we obtain a certificate and developer name for each app. Next, if there are; at least, two apps that have exactly the same public keys, same subjects of certificates, or developer names, we extract the apps as mass-produced.

4.2 Auto-Built Apps

There are several cloud-based app building services such as iBuild App [14] or Bizness Apps [15]. 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 [13]. *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 each other even though they are independently developed by different developers.

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-*

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`

build apps. 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 the 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 that the app B is a malicious clone of 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 a straightforward approach to extract them. We apply VirusTotal [16], which is an online antivirus service composed of more than 60 different commercial antivirus checkers. All the remaining similar apps are applied to the VirusTotal. For a given app, if at least one of the antivirus checkers detects 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 antivirus 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. 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.

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	

5.2 Extraction of Suspicious Apps/ad-Injected Apps

We extracted two categories, suspicious apps and ad-injected apps, which are aimed at covering malware and adware that are not detected by antivirus checkers. 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 API's. If an app B adds; at least, one of the API's associated with the permissions listed in Table 2 and the added API function was not present in the app A, then the app B is considered as *suspicious*. Here, we made use of the API calls for permission mappings extracted by a tool called PScout [17], which was developed by Au et al. [18]. To check the existence of API's, we checked whether a set of API's is included in the disassembled code of an APK file.

Similarly, we checked whether the app B added a new FQDN associated with an ad library. Let's denote such FQDN as ad-FQDN. The key idea of our approach was to make use of a list of ad-FQDNs that were 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 [19]. We then pruned FQDNs that were clearly wrong records, such as `schema.android.com`. In total, the number of ad-FQDNs we compiled was 1,027. Finally, we explored disassembled codes of apps and checked ad-FQDNs. If the app B added at least one ad-FQDN, which

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	–

was not present in the app A, the app B was considered as *ad-injected*.

6. Analysis

In this section, we present our key findings through the analysis of a huge number of Android apps in the wild. We first illustrate the data we used for our analysis. Next, 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.

6.1 Data

We collected Android apps from the official marketplace [20] and third-party marketplaces [21]. 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 markets, in particular, we made use of the data presented in Ref. [8]. 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 reason.

Using the data, we can study the qualitative differences between the two types of marketplaces, the official market and the third-party market. It has been reported that the official marketplace has installed special defense mechanisms called Bouncer [22]. Therefore, as previous studies have reported, the official market tends to have fewer numbers of malicious apps, as compared to those in third-party markets [4]. It is noteworthy that in China, a country with the highest population, the official Google Play market has been unavailable. Therefore, people who hope to enjoy popular apps published in Google Play may have an incentive to import the clones into a third-party market. In fact, as Zhou et al. [4] reported, 5 to 13% of apps hosted on third-party markets were repackaged. In this work, we will study the differences between 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.

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%)

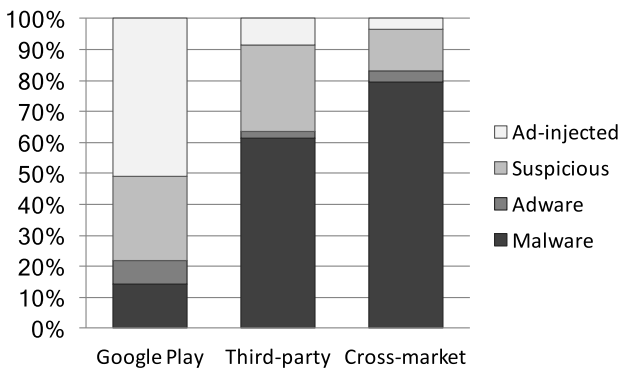


Fig. 3 Breakdown of the clones.

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 the 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 distinguish the two categories systematically. We also notice that the fraction of *clones* in Anzhi is much higher than that in Google Play. Again, the observation generally agrees with the previous reports. The further breakdowns of these categories will be presented later in the paper.

Table 6 shows the breakdown of *relatives*. Clearly, the majority of *relatives* is attributed to *mass-produced*; i.e., prolific developers tend to publish many similar apps, possibly using the same template. In this work, we were not able to find popular app-building services for the third-party market. As a result, the 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 in the official mar-

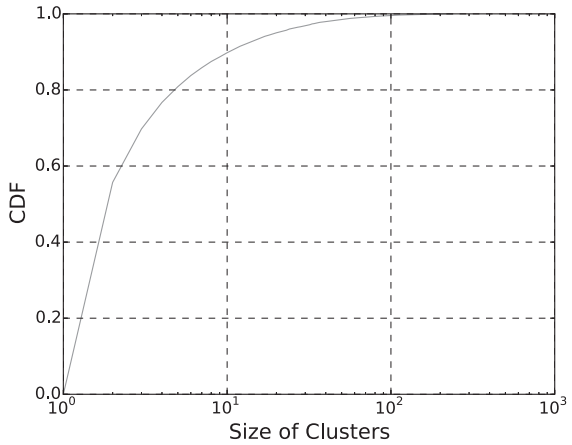


Fig. 4 CDF of cluster size (Google Play).

ket were repackaged and published in the third-party market. For the official marketplace, roughly half of the *clones* were identified as *ad-injected* while the proportion of malware was around 20%. This may correlate to the existence of defense mechanisms, e.g., Bouncer, installed in the official marketplace; in the official market, the proportion of malware is lower than in the third-party marketplaces. We can also observe that roughly 70% of *clones* were *not* detected by commercial antivirus 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 the majority of malicious *clones* found in the third-party market repackaged apps had been originally published on the official market.

Figure 4 plots CDF of cluster size for Google Play. While more than 50% of clusters consist of just two apps, which is the minimum value to form a cluster, more than 10% of clusters consist of 10+ apps. We manually inspected these large clusters and found that most of the apps in such a large cluster are classified as *relatives*.

We studied which categories of apps were more likely cloned. Figure 5 presents the distributions of apps per the category 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, the average/median number of downloads for the original apps that were cloned is higher than the total average/median. Note that on Google Play, the 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.

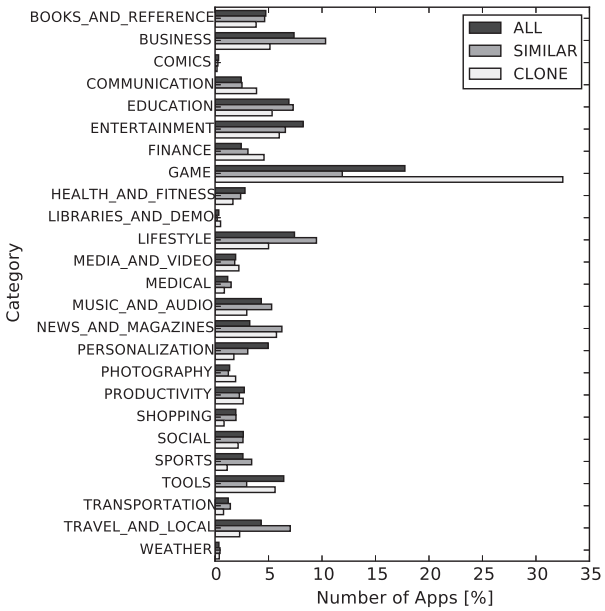


Fig. 5 Distributions of apps per category (Google Play).

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

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

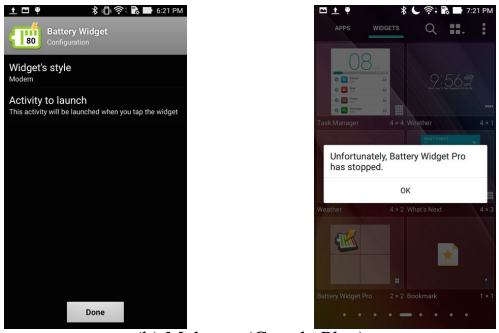
Table 8 Accuracies of clone detection.

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

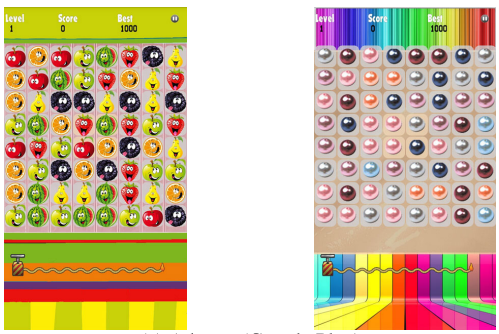
6.3 Validity of Extracted/Classified Clones

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 is no ground-truth database, we validate the accuracy through a manual inspection, which includes 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 actual *clones* by manual inspections. 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

(a) Malware (cross market)

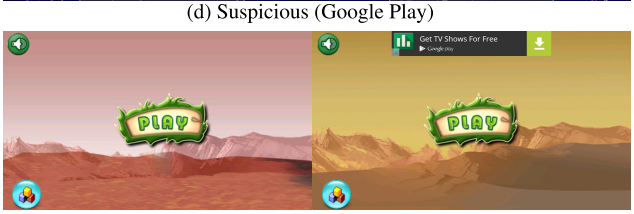


(b) Malware (Google Play)



(c) Adware (Google Play)

(d) Suspicious (Google Play)



(e) Ad-injected (Google Play)

Fig. 6 Screenshots: original apps (left) and clones (right).

errors, the classification of *clones* worked successfully. In the following, we picked up typical samples for each category and presented what we found through the manual inspection.

1) *Malware:* We show two samples here. The first example, shown in Fig. 6 (a), is taken from a cross-market clone (malware). It clearly adds a large advertisement window on the initial screen of a game app. Furthermore, it asks to install an additional app. In the second example, shown in Fig. 6 (b), the number of downloads for the original app was $10,000 \sim 50,000$ while that for the clone was $50 \sim 100$. As shown in the screenshots, the clone app was unexpectedly quit soon after it launched. Both the origin and the clone were still available on the market as of July, 2015.

2) *Adware:* The example is shown in Fig. 6 (c). The clone was repackaged from a puzzle game app. Although the clone uses different icons and images, the structure of the app was identical. The clone has been removed from the marketplace.

3) *Suspicious:* The example is shown in Fig. 6 (d). An app for exploring the constellations. Although its 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 API's such as getDeviceId(), and new additional services, such as DownloadService and PushMessageService. The clone has been removed from the market.

4) *Ad-injected:* The example shown in Fig. 6 (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 antivirus checkers. The clone has been removed from the market.

7. Discussion

In this section, we discuss several limitations of the *AP-Praiser* 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, the 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 thefts and clones, we may need to shed lights on paid apps as well. We leave the issue for future study. Finally, as we have mentioned earlier, we adopted an approach of using antivirus checkers to detect malware and adware. However, the use of an antivirus checker is prone to detection errors. Here, we note the cases where an antivirus checker is useful in finding malicious clones, which are difficult to find otherwise. In the third-party marketplace, we observed that non-negligible numbers of malicious clones are encrypted using a tool called SecAPK [23], 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 they originate from a legitimate app developed by another author.

8. Related Work

There have been several studies that work on analyzing *similar* Android apps. They are broadly classified into two categories: code-based 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 [4] 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 a time complexity of $O(n^2)$. DNADroid [24] 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 [5] 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 attention because it can detect similar apps at a low cost and is not affected with code obfuscation/encryption. We will summarize such work in the next subsection.

8.2 Recourse-Based Approach

Viennot et al. [8] developed a system called PlayDrone, which efficiently crawls the official Google Play Store. Using roughly 1 million apps collected with PlayDrone, they performed various analyses 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 for various reasons, such as application re-branding or application cloning.

Yury et al. [7] proposed a framework called FSquaDRA, which detects similar apps using resource information. They aimed to speed up hash calculations of resources by leveraging the SHA1 digest of each file that was 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. Our new contributions, which have not been established in [8], are as follows. First, unlike Ref. [8], we categorized similar apps into two primary categories: relatives and clones, which are further subcategorized. The detailed classification enables us to clarify the proper actions we need to take against similar apps. For instance, while relative apps are likely legitimate, clones include pirated apps, which should be eliminated from the marketplace. Second, while Ref. [8] studied the official marketplace, we expanded the analysis to the third-party marketplaces. We employed the correlation analysis using the apps collected from both types of marketplaces. This cross-market analysis enabled us to understand the differences in managing similar apps in the marketplace. We note that the fraction of similar apps we discovered in the official marketplace was lower than the one shown in Ref. [8], i.e., 6.1% vs. 25%. We speculate that this is because we considered only the latest version for each app, while the Playdrone dataset comprises duplicated apps with different versions.

9. Conclusion

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 a three-stage strategy: (1) extraction of 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 antivirus checkers and code difference analysis. To answer the second research question, we applied the *APPraiser* framework to the over 1.3 million 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*. 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 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 Android apps, which is equivalent to the size of the official market. To perform such a huge-scale analysis, we also developed lightweight algorithms that can extract similar items from a huge, sparse dataset with the time complexity of $O(n)$.

Acknowledgments

A part of this work was supported by JSPS Grant-in-Aid for Scientific Research B, Grant Number JP16H02832.

References

[1] Y. Ishii, T. Watanabe, M. Akiyama, and T. Mori, "Clone or relative?: Understanding the origins of similar android apps," Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp.25–32, ACM, 2016.
[2] "Stragety analytics." https://www.strategyanalytics.com/strategy-analytics/blogs/devices/smartphones/smart-phones/2015/03/11/android-shipped-1-billion-smartphones-worldwide-in-2014, Jan. 2015.
[3] AppBrain, "Android operating system statistics." http://www.appbrain.com/stats/.
[4] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," Proc. of the second ACM CODASPY 2012, pp.317–326.
[5] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," Proc. of the third ACM CODASPY 2013, pp.185–196.
[6] "Fake apps: Feigning legitimacy." http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fake-apps.pdf.
[7] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "FSquaDRA: Fast Detection of Repackaged Applications," Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy, DBSec '14, pp.131–146, 2014.
[8] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," Proc. of ACM SIGMETRICS 2014, June 2014.
[9] smali. https://github.com/JesusFreke/smali.
[10] Y. Yang and J.O. Pedersen, "A comparative study on feature selection in text categorization," Proceedings of the Fourteenth International Conference on Machine Learning, ICML '97, pp.412–420, 1997.
[11] Androguard. https://github.com/androguard/androguard/.
[12] M. Ester, H.P. Kriegel, J. Sander, X. Xu, et al., "A density-based algorithm for discovering clusters in large spatial databases with noise.," Kdd, pp.226–231, 1996.
[13] 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," Symposium on Usable Privacy and Security (SOUPS), 2015.
[14] iBuildApp. http://ibuildapp.com/.
[15] Bizness Apps. https://www.biznessapps.com/.
[16] VirusTotal. https://www.virustotal.com/.
[17] PScout, "Analyzing the Android Permission Specification." http://pscout.csl.toronto.edu/.
[18] K. Au, W. Yee, Y.F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," Proc. of ACM CCS, pp.217–228, 2012.
[19] AdAway, "http://adaway.org/hosts.txt."
[20] Google Play. http://play.google.com/.
[21] anzhi.com. http://www.anzhi.com/.
[22] J. Oberheide and C. Miller, "Dissecting the android bouncer." SummerCon, Brooklyn, NY., 2012.
[23] "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/.
[24] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," Proc. of the 17th European Symposium on Research in Computer Security, pp.37–54, 2012.

Yuta Ishii received B.E degree in computer science from Waseda University in 2011. He is currently a 2nd-year graduate student in the Department of Computer Science and Engineering, Waseda University. His research intrerest is network security and mobile security.

Takuya Watanabe recieved M.E. degree in computer science and engineering from Waseda University, Japan in 2016. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2016, he has been engaged in research and development of mobile security. He is now with the Cyber Security Project of NTT Secure Platform Laboratories.

Mitsuaki Akiyama received the M.E. degree and Ph.D. degree in Information Science from Nara Institute of Science and Technology, Japan in 2007 and 2013, respectively. Since joining Nippon Telegraph and Telephone Corporation NTT in 2007, he has been engaged in research and development of network security, especially honeypot and malware analysis. He is now with the Network Security Project of NTT Secure Platform Laboratories.

Tatsuya Mori is currently an associate professor at Waseda University, Tokyo, Japan. He received B.E. and M.E. degrees in applied physics, and Ph.D. degree in information science from the Waseda University, in 1997, 1999 and 2005, respectively. He joined NTT lab in 1999. Since then, he has been engaged in the research of measurement and analysis of networks and cyber security. From Mar 2007 to Mar 2008, he was a visiting researcher at the University of Wisconsin-Madison. He received Telecom System Technology Award from TAF in 2010 and Best Paper Awards from IEICE and IEEE/ACM COMSNETS in 2009 and 2010, respectively. Dr. Mori is a member of ACM, IEEE, IEICE, IPSJ, and USENIX.