Empirical Study of Android Repackaged Applications

Kobra Khanmohammadi[1], Neda Ebrahimi[1], Abdelwahab Hamou-Lhadj[1], Raphaël Khoury[2]

Abstract The growing popularity of Android applications has generated increased concerns over the danger of piracy and the spread of malware, and particularly of adware: malware that seeks to present unwanted advertisements to the user. A popular way to distribute malware in the mobile world is through repackaging of legitimate apps. This process consists of downloading, unpacking, manipulating, recompiling an application, and publishing it again in an app store. In this paper, we conduct an empirical study of over 15,000 apps to gain insights into the factors that drive the spread of repackaged apps. We also examine the motivations of developers who publish repackaged apps and those of users who download them, as well as the factors that determine which apps are chosen for repackaging, and the ways in which the apps are modified during the repackaging process. Having observed that adware is particularly prevalent in repackaged apps, we focus on this type of malware and examine how the app is modified when it is injected in an app's code. Our findings shed much needed light on this class of malware that can be useful to security experts, and allow us to make recommendations that could lead to the creation of more effective malware detection tools, Furthermore, on the basis of our results, propose a novel app indexing scheme that minimizes the number of comparisons needed to detect repackaged apps.

Keywords Mobile Applications, Application Repackaging, Mining Android Application Repositories, Mobile Security.

1 Introduction

According to the Open Web Application Security Project (OWASP), application repackaging is one of the top ten mobile risks (OWASP 2016). Repackaging of a mobile application (app for short) consists of downloading an app from an app store, decompiling it, changing its content, recompiling it, and finally uploading it again to an app store. This process is facilitated by the existence of open source tools such as APKtool (Wiśniewski 2012).

App repackaging has several negative effects. It threatens a loss of revenue for app developers by republishing paid original app for no cost, or by removing the existing advertisements. To make matters worse, attackers often resort to the use of repackaging to spread malware. Zhou et al. showed that more than 85% of malware are introduced through app repackaging (Zhou and Jiang 2012). Recent incident reports confirm that repackaging is still a popular way for distributing malware. For example, in April 2017, the malware "FalseGuide" was embedded in several repackaged apps available on the Google Play Store (Kumar 2017).

Several studies (e.g., Zhang et al. 2014, Shao et al. 2014 and Jiao et al. 2015) sought to automatically detect repackaged (and malicious) apps. The common practice is to extract attributes from an app such as opcodes, method calls, images, resources, and user interface graphs and use them to detect the corresponding repackaged apps. Despite the advances in the field, repackaging remains a serious threat, partly because it is still not well understood. In this paper, we perform an empirical study in order to understand the factors that make apps vulnerable to being repackaged. We achieve this by empirically examining 15,296 pairs of original and repackaged Android apps, published in AndroZoo (Li et al. 2017a), one of the largest collection of Android apps used in research studies on mobile security.

Kobra Khanmohammadi, Neda Ebrahimi, Abdelwahab Hamou-Lhadj
{k_khanm, n_ebr, abdelw}@ece.concordia.ca

Raphaël Khoury
raphael.khoury@uqac.ca

[1] Department of Electrical and Computer Engineering, Concordia University Montreal, Qc, Canada
[2] Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada

This study addresses five research questions:

RQ1) What is the unfavorable prevailing usage of repackaging?

Repackaging is used for different purposes such as revenue manipulation through advertisement, piracy, and the introduction of malware. After examining the repackaged apps, we found that repackaging is mainly used to introduce adware as opposed to other types of malware such as Trojans, etc. Adware is defined by Erturk (2012) as any software package that automatically presents advertisements to users by guessing from their previous surfing or search activities. Symantec (Chien 2005) additionally stresses that adware might perform other malicious activates, such as redirect a user's searches to advertising websites, and collect personal data about users. Gao et al. (2019) likewise classify adware as a type of malware, and found that adware commonly performs multiple malicious operation, alongside with displaying ads. Adware is commonly classified as a subclass of malware (e.g. (Gupta 2013) and (Xue et. al 2017)) with distinctive objectives and modus operanti. We conclude that the main motivation behind app repackaging is to gain revenues through advertising.

RQ2) How is the code of original apps manipulated to embed adware?

Drawing upon our findings from RQ1, we focus on adware, and study how its introduction in apps leads to modifications in the apps' code. More particularly, we examined the API calls present in the parts of the code manipulated in repackaged apps and compared it to API calls present in the original apps. We found that the API calls that are added to repackaged apps are usually the same ones that are already present in the original apps. As a consequence, malware detection based on an analysis of API calls alone may be impractical.

RQ3) Which type of apps have been exploited for repackaging?

To answer this question, we investigate the relationship between the popularity of an app and its likelihood of being repackaged. To this end, we propose three metrics to quantify the popularity of an app: user rating, the number of downloads, and the popularity of the app store from which the app can be downloaded. We observed that apps that have a rating greater than 3 out of 5 are more likely to be repackaged. Moreover, most repackaged apps are modifications of apps downloaded from the Google Play Store. In addition, we studied the use of obfuscation and applied static analysis to identify the obfuscations techniques used in the original apps. Obfuscation techniques include method name changes, reflection and dynamic loading. Our findings show that most of the apps that get repackaged do not use name changes. This result suggests that the ease of understanding the code of an app is a major factor that leads the app to be targeted for repackaging.

RQ4) Why do users download the repackaged apps when the original versions are available for free?

To answer this question, we examined the popularity of repackaged apps considering their star rating, download number and the app store which they are published. We found that many repackaged apps exhibit a high user star rating, high download number and were published in popular app store, which shows that they were successful in seducing users to download them. In addition, we examined the names of the repackaged apps to understand how they differ from those of the original apps. We found that repackaged apps usually have high star–rating, just like the original ones. In addition, the names of repackaged apps are usually kept similar to the original names, possibly to increase the odds of it being found by search engines. One interesting observation is that when the names of the repackaged app are changed, the new name is often in a different language than the original application, perhaps to target users from a specific region.

RQ5) How are an app's attributes modified in the repackaged version?

We studied how a selected set of apps' attributes are altered after the apps are repackaged. We observed that the size of apps may or may not increase after repackaging. We analyzed the number of app components and their names in the original apps and their repackaged counterparts. We found that the number and the names of components in repackaged apps are similar to those of the original versions. Another attribute that we studied is an app's permissions, which are used to access the mobile device's resources. We found that the number of permissions requested by an app generally remains the same after it is repackaged. Based on these findings, we propose an indexing scheme to record the apps in order to decrease number of comparisons needed to detect repackaged apps.

The remainder of this paper is organized as follows: In Section 2, we provide the background needed to understand the architecture of an Android app. In Section 3, we present the study's methodology. The empirical study is given in Section 4.

We present related work in Section 5. In Section 6, we discuss the threats to the validity of our paper. Concluding remarks are given in Section 7.

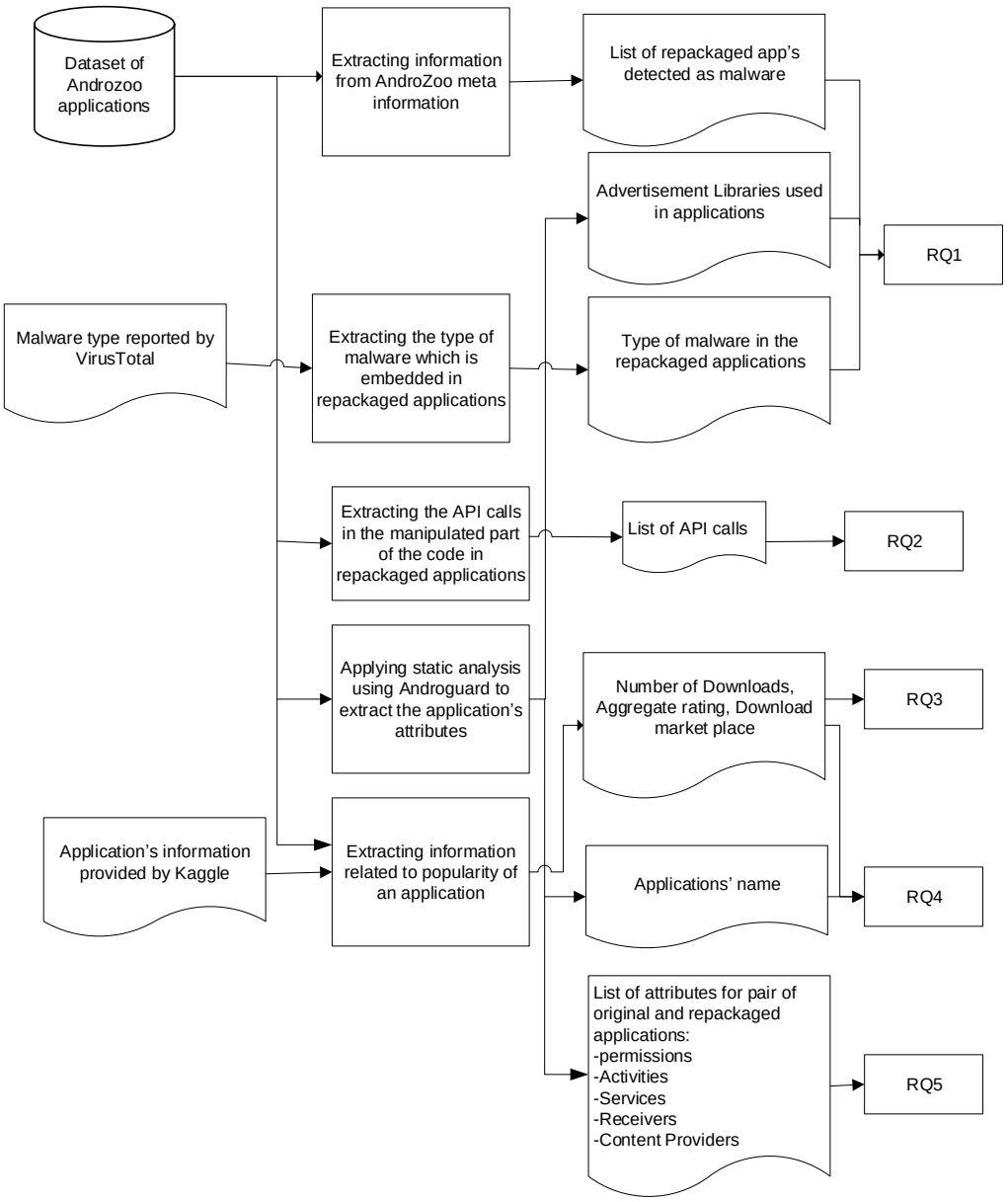

Fig. 1 Overview of our methodology

2 Background

An Android app is uploaded as a zip file with extension ".apk" to an app store. This file generally contains an app program in the form of a classes.dex file, as well as app resources like pictures, music and .xml files, which describe the layout information. It is also required to contain the file AndroidManifest.xml, which contains information about the app- its name, version, access rights, app components and referenced libraries. The file Classes.dex is in the format used by the dalvik virtual machine provided in Android.

When an app is about to be installed on a device, the user is prompted to give it permissions to access device resources such as the network. Users can either accept all the requested permissions or refuse the installation of the app. The tag <*uses-permissions*> in the AndroidManfest.xml lists the permissions that are needed by the app. Note that from Android Marshmallow 6.0 (released in October 2015), apps can also ask for permission at runtime.

Android apps are usually written in Java and contain four types of components: Activities, Services, Content Providers and Broadcast Receivers. In fact, there are four main classes named: Activity, Service, Receiver, Content Provider. Each component can inherit from them. Activities represent what a user can do with the app. They represent the user interface. Services run in the background and usually contain long running operations. Content providers manage shared sets of an app's data. Broadcast receivers are used to respond to system-wide broadcast announcements such as *"the battery is low"*.

The malicious operations in Adware may be limited to showing Advertisements in a way that is not concordant with the Advertisement Network policies (e.g., AdMob (2019)) or may include other malicious operations such as reading the device IMEI, the user account list and the device's location and sending this information to a third party. For example, the listing below, in Fig. 2, shows a segment of code from an adware called "Appenda" (Symantec 2014). This adware contains a service called AppNotify. The service is started as soon as the system is booted. AppNotify pulls ads and show them to the user as notifications. This behavior contravenes to the ad network policies, which states that advertisement can only be shown to users in the apps' user interface, and only while the application is running. Note that out of a concern for clarity and concision, we edited the code by removing some condition checking related to time and network status.

```
public class AppNotify extends Service
{
    public void onStart(Intent intent, int i)
    {
        slinger = new Appenda(getApplicationContext());
        slinger.activateAlarmNotifications();
    }
    private Appenda slinger;
}

public class Appenda //class for pulling and showing advertisements
{
   Appenda(Context c){
    currentServer=
(Appenda)com/appenda/Appenda.getClassLoader().loadClass("com.appenda.AppendaServer").newInstance();
    currentServer.setCurrentContext(c);
    if (Condition) //check the time and network availability and update data frequently
         updateVersion(true);
    SharedPreferences = getCurrentContext().getSharedPreferences(SETTINGS_FILE, 0);
    currentServer.setApp_id(getApp_id(sharedpreferences)); // the data was already saved in a
SharedPreference
    currentServer.setPublisher_id(getPublisher_id(sharedpreferences));
    currentServer.setSubid(getSubid(sharedpreferences));
    currentServer.setPublisher_key(getPublisher_key(sharedpreferences));
   }

   public void activateAlarmNotifications()
   {
     while(!isNetworkAvailable() || currentServer == null)
        return;
     currentServer.displayWebAd();//show the advertisement
     updateVersion(true);

   }

   private void updateVersion(boolean flag)
   {
     if(isNetworkAvailable()) {
       SharedPreferences sharedpreferences = getCurrentContext().getSharedPreferences(SETTINGS_FILE,
0);
        android.content.SharedPreferences.Editor editor = sharedpreferences.edit();
                    // read information and save in a sharedPreference
        String phoneNumber= "";
        if(!DEBUG)
```

```
            phoneNumber=
((TelephonyManager)getCurrentContext().getSystemService("phone")).getLine1Number();
         else
            phoneNumber = "8885550001";
            editor.putString("phone_number" , phoneNumber);
            editor.commit();
      }
   }
}
```

Fig. 2 Snippet code from adware Appenda

3 Methodology

Fig. 1 shows an overview of our methodology, highlighting the process for extracting information from the dataset to address each of the five research questions. For RQ1, we used the information that already exists in the AndroZoo dataset. We also used the tool Androguard (Desnos 2015) to extract information from the apps' Androidmanifest.xml file. Some of the information extracted includes the app name, name of components and the app's permissions. We also used Androguard to perform a static code analysis to extract the advertisement libraries present in each app. In answering RQ2, we needed to identify the parts of the code that had been manipulated in repackaged apps by comparing the code of repackaged apps with that of the original version. The process needed to accomplish this task is detailed in section 4. After finding the manipulated part of the code in original and repackaged apps, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls present in the manipulated part of the code. For RQ3 and RQ4, we used the information extracted from Kaggle (Leka 2016) as well as the obfuscation information got through static analysis. We also studied the name of apps extracted from AndroidManifest.xml file of apps. For RQ5, we used the information obtained from the AndroidManifest.xml file of each app.

The dataset used in this study is based on AndroZoo (Li et al. 2017a), one of the largest datasets of Android apps. It was collected from various sources, including the official Google Play app market. AndroZoo currently contains more than 5 million different APKs. Each app is scanned by at least ten different anti-virus products and the results of these scans are reported in the dataset.

AndroZoo contains a list of pairs of original and repackaged apps. Each row in the list is described with an SHA (a unique key belonging to each app) of the original app, followed by the SHA of the repackaged app. AndroZoo contains 15,296 repackaged apps of 2,776 original apps, organized in pairs. One original app may have multiple repackaged versions. We used these original-repackaged app pairs as the dataset for our study.

Table 1 An example of an app's attributes

Attribute Name	Attribute Value
SHA256, SHA1, MD5	00FAFD8DCDBD59FF035117FF1E19C8329A92AB797E452304FBE82AA9027ACF24,E85B066F301 ADD14BED013DB463EE5850316552D, D5AD33B451B84BEF59DA4CB80164544D
APK size	1375601
Market	Slideme
Package Name	kr.mobilesoft.yxplayer
VT Detection	0
VT Scan Date	2013-07-17 20:17:35

An AndroZoo app is described according to the following attributes (see Table 1 for an example):

- App identifier: The app is identified using SHA256, SHA1, and MD5 hash values.
- File size: The size of the APK file in bytes.
- Market: The market where the app is published. An app may be published in more than one market.
- Package name: The name of the Android app package, as reported in the manifest file.
- Version code: The app version number, as reported in the manifest file.

- VT Detection: The number of anti-viruses from VirusTotal (2018) that detect malware in the app's code.
- VT Scan date: A timestamp that indicates when an app was scanned by VirusTotal.

In addition to the data provided by AndroZoo, we also used a database of about 300,000 apps, gathered by Leka and made available on Kaggle (Leka 2016). This database contains general information about Android apps including the ones used in this study, gathered from Google Play Store. The information provided includes the APK name, APK file size, price, number of downloads of a given app, and the average rating. We used Kaggle to extract supplementary information about the apps in our dataset when this information was not provided in AndroZoo. The information available in Kaggle includes (see Table 2 for example):

- Name: The app's name as reported in the manifest file.
- Number of downloads: Google Play Store provides a range of downloads. In Kaggle's dataset, only the minimum number of downloads is provided.
- Aggregate Rating: Users can rate apps using 1 to 5 stars. The aggregate rating is the average of the ratings assigned by all users.

Table 2 An example of an app's attributes in the Kaggle database

Name	Yxplayer
Minimum number of downloads	1,000,000
Aggregate rating	3.335

Finally, we identified the parts of the code that have been manipulated in repackaged apps by comparing the code of each repackaged app with that of its original pair. The details related to the manipulation of the code are explained later in RQ2. After identifying the manipulated part of the code in the original and repackaged apps, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls from the manipulated part of the code.

4 Empirical Study

In this section, we present, for each research question, the motivation behind the question, our findings and a discussion placing our findings in the broader context of app security.

4.1 RQ1. What is the unfavorable prevailing usage of repackaging?

Motivation: Previous studies (e.g., (Zhou and Jiang 2012) and (Zhou et al. 2012b)) have identified three main motivations for app repackaging, namely sharing paid apps with no cost, spreading malware, and embedding advertisements. These studies, however, did not provide any statistics that would permit estimating the relative prevalence of each of these motives. Since every app in our dataset is free, we cannot glean any insights related to the prevalence of repackaging for sharing paid apps. We will therefore only focus on malware spreading and advertisement manipulation.

Approach: Each app in AndroZoo is scanned using VirusTotal, a powerful tool that scans apps using more than 30 anti-viruses. The number of anti-viruses that classify an app as malware is included in the AndroZoo dataset. We used this information to determine the number of repackaged apps that introduce malware. To study the presence of advertisements, we compared the advertisement libraries used by the original apps with those present in their repackaged counterparts. We achieved this by applying static analysis over the source code using Androguard. We verified the list of extracted advertisement libraries by cross-referencing them with the ones described by Book et al. (2013). These authors provided a detailed list of advertisement packages commonly used in Android app development. Moreover, we retrieved the name and type of malware in repackaged apps from the work done by Hurier et al. (2017). Their results clarified the percentage of repackaged apps identified as adware.

Findings:

F1: 52.22% (7,988 out of 15,296) of repackaged apps are classified as malware by at least two anti-viruses according to VirusTotal reports. All of the apps in our dataset are classified as malware by at least one anti-virus. This is the criterion for inclusion in the dataset used by Li et al. (2017a). Since in some studies (e.g., (Arp et al. 2014) (Canfora et al. 2013)), only the

apps detected by at least two anti-virus products are used in their malware sample, we provide data about the distribution of apps detected as malware in our dataset by the number of anti-viruses detecting them in Fig. 3. As can be seen, 52.22% of repackaged apps are classified as malware by at least two anti-viruses. This suggests that even by the stricter standard used by the papers mentioned above, over 50% of repackaged apps contain malware. Fig. 4 shows the distribution of repackaged apps, all containing malware, between the years 2010 and 2014. As can be seen in that Fig. 4, the prevalence of malware in repackaged apps seems to be increasing. While this increase may be partly attributed to the manner in which the data was collected, it does confirm the findings of Zhou and Jiang on this topic. (2012). In 2012, they showed that repackaging is a common vector of malware distribution, based on a dataset of 1,260 apps. Fig. 4, which shows that the incidence of repackaging in the Androzoo dataset, provides further indication that this trend continued after 2012. Moreover, as shown in Fig. 5, these repackaged apps are often also published in trusted app stores such as Google Play Store.

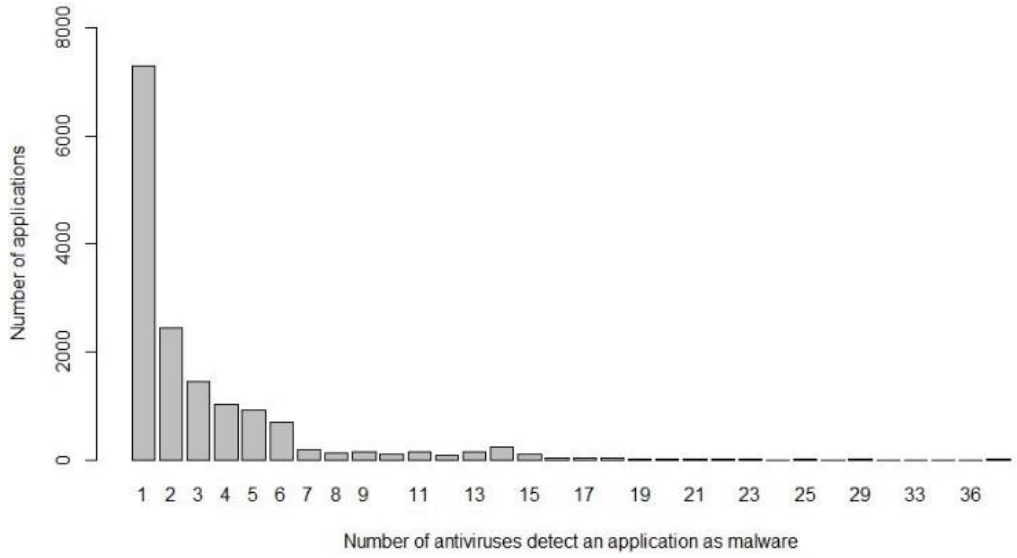

Fig. 3 Distribution of apps by the number of anti-viruses that identify them as malware

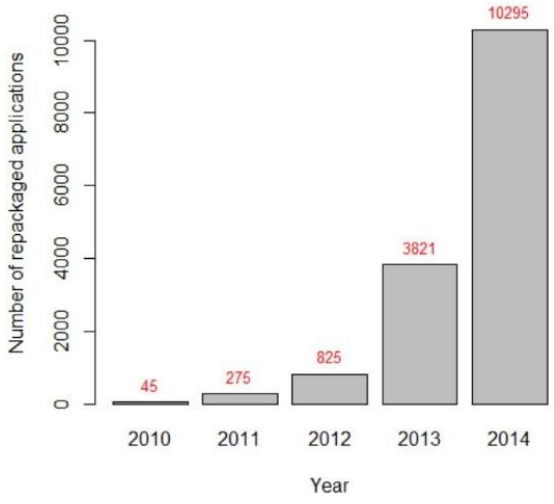

Fig. 4 Distribution of repackaged app containing malware between the years 2010 and 2014

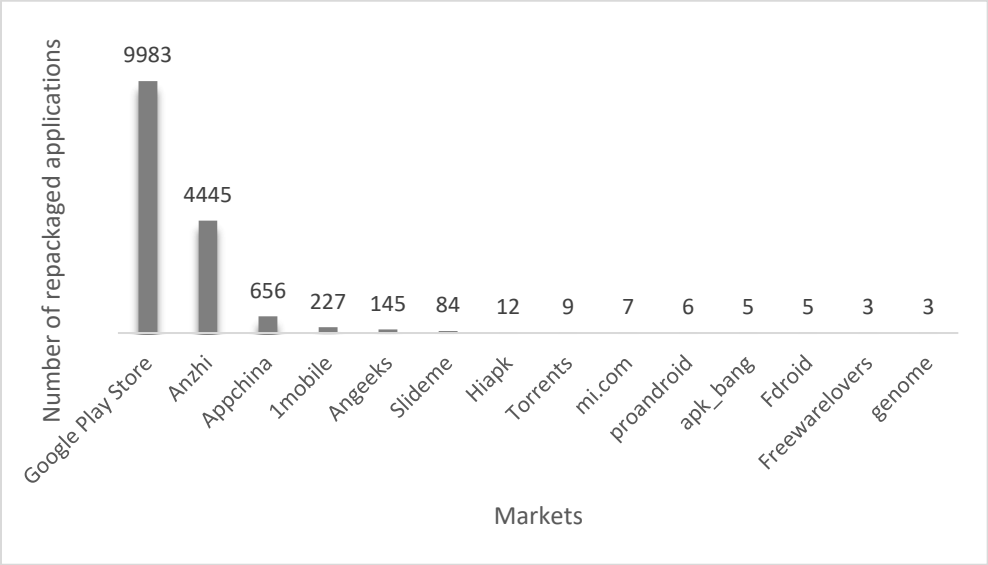

Fig. 5 Distribution of repackaged apps in a variety of markets

F2: 15.47% (2,367 out of 15,296) of the repackaged apps contain additional advertisement libraries in comparison to their original apps. In 1,968 of these (i.e., 83%), additional advertisement libraries were added to apps that originally had advertisement libraries. We observed that only 399 repackaged apps out of 2,367 (i.e., 17%) had advertisement libraries added while the original versions did not contain any advertisements at all.

In addition, we examined the advertisement packages used in repackaged apps as well as original apps (see Fig. 6). The data is sorted according to the difference between the frequency of occurrence of each library in repackaged apps and in original apps. We found that com.revmob, com.google.android.gms.ads, and com.mobclix are the advertisement libraries that are used most frequently (in 80% of all cases) in the repackaged apps. Indeed, these specific libraries are also much more likely to be used in repackaged apps than in the original ones.

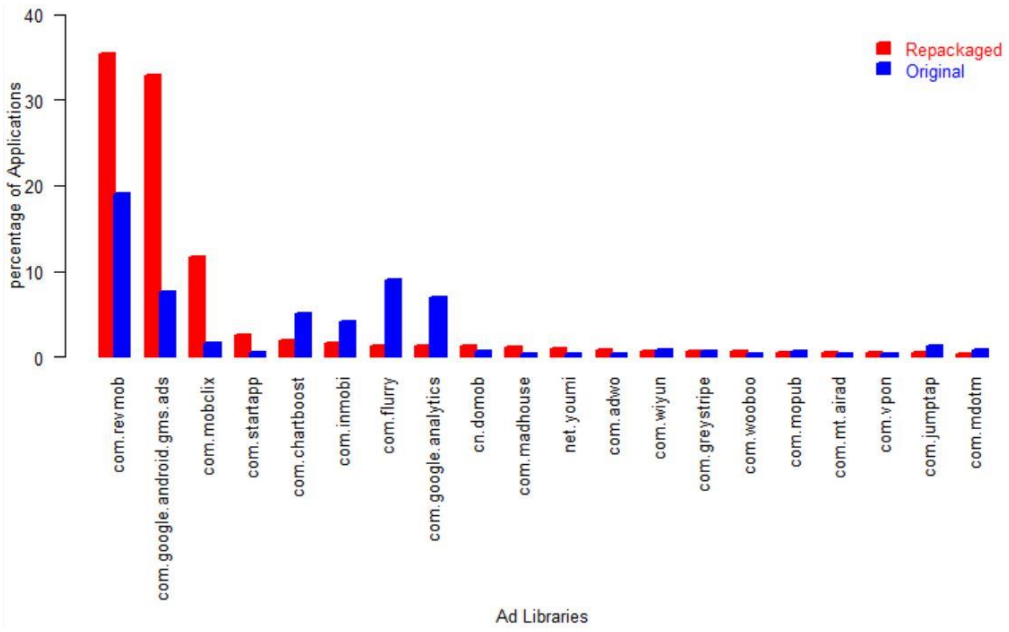

Fig. 6 Advertisement libraries used in repackaged apps

F3: 77.84% (7,954 out of 10,218) of repackaged apps contain malware of the type "Adware". To identify the type of malware present in repackaged apps, we relied upon a study by Hurier et al. (2017), in which the authors developed an approach to identify the type of malware embedded in repackaged apps. Their approach is based on text mining of VirusTotal reports. However, we were only able to uncover the type of malware for 10,218 of the repackaged apps present in our dataset. We believe that this is because of the assumptions made about the patterns for the malware label which is usually reported in anti-virus's reports. It may happen, for instance, that some of the reports provide a label which does not match with any of the patterns accepted by the tool (Hurier et al. 2017). In some cases, the report may contain the name, but not the type, or vice versa. Here, we considered all the apps for which we were able to obtain both the name and the type of malware it contains.

We found that 7,954 out of 10,218 (77.84%) repackaged apps have malware of type "Adware". We also found that only 1,691 out of 10,218 (16.54%) repackaged apps have malware of type "Trojan". The remaining apps (5.62%) have other types of malware such as backdoors, spywares, worm, etc. Li et al. (2017b) used the most frequently appearing names in the reports of anti-virus engines in order to classify the malware types. They have shown that 888 out of 1,575 (56.38%) apps in their dataset are adware. Our result shows a higher percentage.

Analysis, discussion, and implication: All of the repackaged apps in the AndroZoo dataset are detected as malware by at least one anti-virus and 52.22% are detected as such by at least two anti-viruses (the standard used in multiple studies). It shows the bias in the dataset that focuses on repackaged apps that are detected as malware. Even so, as shown in Fig. 3 and Fig. 4, the considerable number of repackaged apps and their increasing trend suggest that repackaging is a common way to distribute malware even in trusted stores such as Google Play Store. F2 and F3 indicate that the repackaged apps are widely used to spread a specific form of malware, namely adware. This finding has implication for malware detection since adware often differs from malware in ways that make detection more difficult, as we will show later in this paper (Finding F6). As a consequence, we have therefore focused this study on identifying the parts of the code that are manipulated when adware is injected in repackaged apps. A careful study of adware, and of how it differs from other classes of malware, will guide the creation of more effective methods of malware detection.

4.2 RQ2: How is the code of original apps manipulated to embed adware?

Motivation: Findings in RQ1 have shown that a large number of repackaged apps are adware which is distributed in trusted app stores such as Google Play Store. The approaches proposed in the literature focus on detecting malware without differentiating adware from other types of malware. Mariconti et al. (2017) proposed an approach called MaMaDroid to detect malware. Interestingly, amongst the apps not detected as malware by MaMaDroid 45% were adware. Therefore, providing an effective approach for detecting adware is an important challenge. But in order to effectively detect adware, we first need to study the code of repackaged apps and examine thoroughly how the code in adware components is changed.

Approach: In order to study parts of code that are manipulated when adware is embedded into repackaged apps, we first need to identify and extract such code fragments. To accomplish this task, we relied upon the approach presented in Fig. 7.

We first grouped the repackaged apps that contain adware based on the name and type if the present, such that two apps containing the same malware type and name (for example all apps with the malware called *Plangton* and whose type is *addisplay)* will be in the same group. In total, there are 106 groups of repackaged apps with different adware name and types. To avoid repetition, we randomly chose only one pair of original and repackaged apps from each group. This is because the changes that occur after a given adware is injected in a repackaged app tend to always the same, for all apps that receive this same adware. Furthermore, sine the number of apps in groups varies greatly, if we considered all apps equally, our results would be skewed towards the changes associated with specific adware, rather than represent a more systematic overview of the types of changes that can occur when adware is added to repackaged apps.

In order to study the manipulated part of the code in repackaged apps that are categorized as adware, we need to extract those parts from the code. To this end, we compared the code of each repackaged app with that of the original app. Fig. 7 shows the approach we used to achieve this task. First, we used the tool "Dex2jar" to extract the jar file from the dex code of the app. Second, we used a Java decompiler to retrieve the Java source code from the jar file. After obtaining the Java code of every repackaged and original apps, we used piece-wise hashing to generate a digest of files. Piece-wise hashing is a fuzzy hashing method that divides the content into pieces and makes the hash of each piece in order to compute the final hash. Piece-wise hashing provides an almost similar hash if the content of two files have only minor changes, but generates more different hashes

if the compared files contain substantial differences (Kornblum 2006). Using this technique, we are able to find pairs of similar files in the code of original and repackaged apps. To hash the java files of an app, we used Ssdeep (Kornblum 2006), a piece-wise hashing tool developed by Kornblum. Note that it is not feasible to simply rely upon the class file names because, in some cases, the names of the files are altered during the repackaging process, perhaps by using obfuscation tools.

An alternative option to piece-wise hashing is to use code-based similarity techniques. However, it has been shown that code similarity incurs higher time complexity (Huang 2008). To do the comparison in a reasonable time period, piece-wise hashing has been proposed as a data reduction technique that limits the comparison of the content of the entire file to a geenrated hash fingerprint (Kornblum 2006; Li et al. 2015).

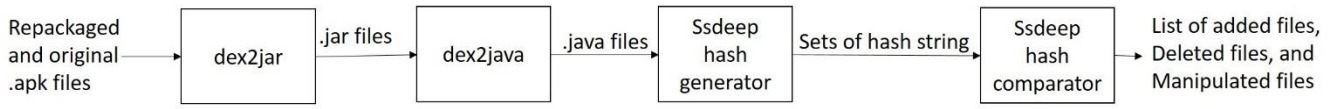

Fig. 7 Approach for finding the manipulated part of the code in repackaged apps

At this stage, we mapped the apps to a set of hash strings, where each string is the hash of a Java file in the app. We then used Ssdeep to find the files that have been manipulated in the repackaged apps. Ssdeep computes the match score between two hash strings and returns a number between 0 (no similarity) and 100% (full similarity). We compared each hash string in the set of hash strings of a repackaged app with all other hash strings in the set of hash strings of its original app (the one which is identified as the original app of this repackaged app in the AndroZoo dataset).

More precisely, consider an original app, O, and its repackaged version, R. Let us assume that the number of files in the repackaged app is M and the number of files in the original app is N. Using Ssdeep, we compute the match score between each two files, f_i and b_j of R and O, respectively with i:1..M and j:1..N. The objective is to identify:

- Manipulated Files: The files in the repackaged app, R, which are manipulations of files in the original app, O.
- Added Files: The files in the repackaged app, R, that do not exist in the original app, O.
- Deleted Files: The files in the original app that were removed from the repackaged app.

To achieve this, we use two thresholds t1 and t2 (t1 < t2):

- If a hash string of a file in the repackaged app matches a file in the original app with a score of t2 or higher then we consider the two files as identical in both R and O. The underlying classes require no further study and we remove them from both sets.
- If a hash string of a file in the repackaged app matches a file in the original app with a score between t1 and t2 then we conclude that a file has been manipulated. We will examine such files later on.
- If a hash string of a file in the repackaged app does not match any hash string of any file in the original app with a score above t1, then the mapped repackaged app file is considered as a file added in the repackaged app.
- If a hash string of a file in the original app does not match any hash string of any file in the repackaged app with a score above t1, then the mapped original app file is considered a file deleted after repackaging.

We determined t1 and t2 through experimentation. We varied t1 from 10% to 100% with a step of 5% and found that when setting t1 to 70%, we obtain at most one file of the repackaged app that is a manipulation of a file of its corresponding original app. This threshold was also used by Zhou et al. (2012a) when detecting similar apps for identifying repackaged apps in third-party marketplaces. As for t2, we opted for 98% similarity because we found that it is the highest value to correctly match classes of the app files while being tolerant to very minimal alterations such as changing the name of the class. A lesser threshold could be used, in which case, we may end up excluding files in the repackaged apps that are manipulations of files in the benign apps. To reduce this risk we decided to keep 98%.

After obtaining a list of files, which have been manipulated, deleted, or added, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls in the manipulated part of the code. Soot allows us to avoid the drawbacks of using tools for decompiling a dex code to its java code by providing static analysis over the dex code directly. In particular, Soot can generate a listing of all API calls present throughout an app's code. Using this functionality, we listed API calls present in each class in pairs of original and repackaged apps and compared them. For the manipulated files, after extracting the API calls, we

used the module "sets" in Python to extract the different API calls in pairs of original and repackaged manipulated files. In the manipulated files, the subset of API calls that exists in the original app but not in the repackaged ones corresponds to deleted API calls. Conversely, a subset of API calls that exist in the repackaged app but not in its original app pair corresponds to added API calls. Note that using this method for identifying differences understates the changes that occur in the repackaging process since some manipulated API calls will be present in both the original and repackaged apps, but with different input parameters or calling context.

The main benefit of the approach we propose is that hashing can be performed over the entire content of the files instead of only on their names, or on the names of the methods and classes. Therefore, even if the name of files and contained classes is changed (a common obfuscation technique), our approach can still determine that the different files are related based on the piece-wise hashing of their contents.

Note that in the presence of obfuscation, an API may appear to have been simultaneously removed from the original app and added to the repackaged app, when in fact the code is unchanged, only obfuscated. This fact does introduce a small possibility of error. However, in our dataset, this scenario is actually rather uncommon. In fact, less than 4% of the repackaged apps (see Table 4) exhibit obfuscation of a type that would introduce this error, such as name changing.

Findings:

F4: Some permissions are more frequently requested in adware. Fig. 8 shows the top 20 most frequently added permissions in adware samples in comparison to the rest of malware. To improve readability, we have removed the first two parts of the permission's name which usually are "android.permission" or "com.android". In total, there are 260 out of 7,954 adware and 95 out of 1,691 samples with other types of malware (i.e., not adware) in our dataset, which contain a set of permissions added in the repackaged version in comparison to its original pair. The frequently added permission "android.permissions.GET_TASK" is used to obtain information about the processes that are executed by apps. It was deprecated in API level 21 due to security violations it may cause. Permission "com.android.vending.CHECK_LICENCE" is frequently used in adware, but no so much in Trojans. Applications need this permission to use a Google service that verifies the license of applications. It provides an infrastructure to allows apps to connect to a distant host, and its presence suggests that the developers of repackaged apps might wish to connect to the devices running their apps for some reason. More research is needed to elucidate the motivation of repackaged app developers in activating this functionality. Permission "com.android.vending.BILLING" is used for purchasing in-app products such as additional game levels, media files, or online magazine services. It has been used in 17 out of 260 adware samples. Obviously, if an app has this permission, it can also use it to perform payments not authorized by the user. Some studies (Mulliner et al. 2014, Raynoud et al. 2012 and Dong et al. 2018a) focus on the threat related to detouring in-app purchases in order to obtain free services without paying. To the best of our knowledge, there is no study showing that in-app purchase is exploited by adversaries to get revenue. More studies are needed to clarify how this permission is exploited and what are the possible protections against this threat. Another permission commonly used in adware is "android.permission.VIBRATE". It allows access to the vibration setting and can be useful to get users' attention, but can be annoying to some users. The other permissions frequently added in repackaging serve to access device resources, such as Internet connection or logs.

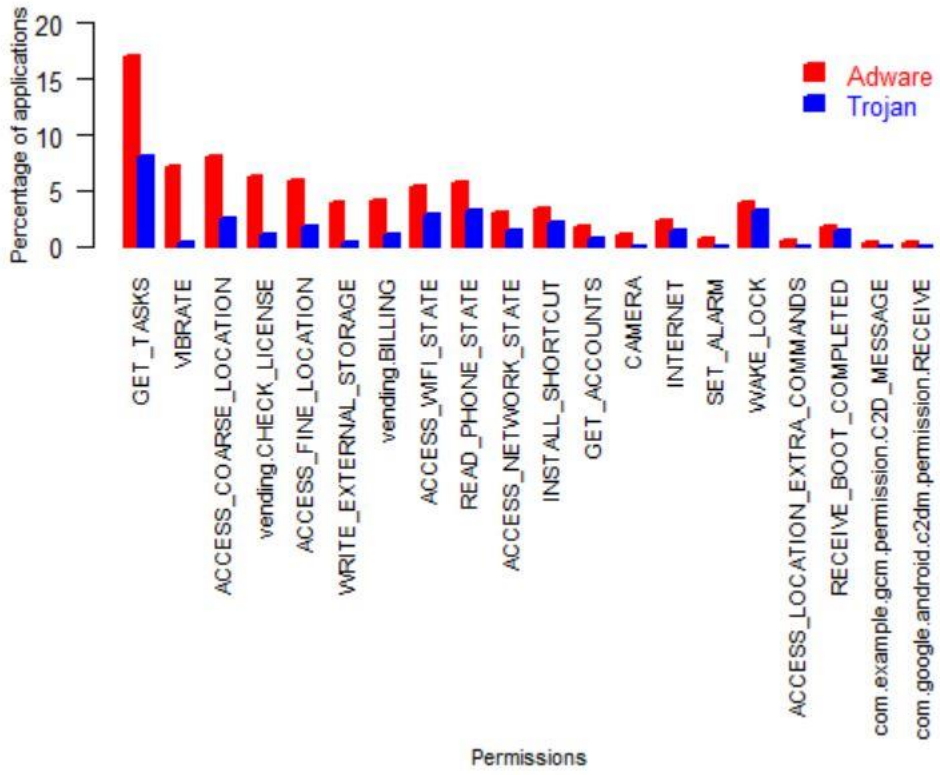

Fig. 8 The frequency of the top 20 permissions most frequently added to adware as opposed to trojan

F5: In 27% of repackaged apps containing adware with no call to APIs that require permission to execute. We studied the API calls extracted from files added in repackaged apps as well as the APIs present in the manipulated files from repackaged apps but not its original pair. Among the extracted APIs, we identified the APIs that require permissions to execute by relying upon a study by Au et al. (2012). From 106 adware samples, each selected randomly from each group of repackaged apps with different adware name and type, 26 (24.52%) did not require permissions to run. This suggests that these adware samples do not perform malicious operations such as leaking confidential data or executing commands in devices. We studied the java code of these samples. We found that reflection calls are present in 5 out of 26 samples in original and repackaged version, but with different parameter strings. Those samples also contain name changing obfuscation. For the rest of the samples, we found that the files changed in the repackaged apps are in fact the obfuscated version of files in the original version. The obfuscation only name changing and changing the name of some of the variables. It seems that the developers of the repackaged app obtain revenue by publishing the app under his own name. There remains the question determining how VirusTotal identified the repackaged app, but not the original app, as adware. While it was not the case in our dataset, there may be cases in which the permissions that are requested at run time and where malicious code is added dynamically at run time. Therefore, using static analysis is not sufficient to detect these kinds of adware.

F6: Detection of adware based on studying only API calls may not be effective. In Fig.9 and Fig. 10, we used the package of API calls instead of the API calls themselves to present a summary of manipulated API calls that require permissions in order to execute. As shown in Fig. 9 and Fig. 10, the most frequently added API calls and most frequently deleted API calls have considerable overlap. In each pair of original and repackaged apps, we extracted the APIs that have been added to and deleted from the original app. We found that that most of the API calls exhibit a similar distribution in original apps and the repackaged adware samples.

Previous research used API calls for malware detection (Alazab et al. 2010; Islam and Altas 2012; Wu et al. 2012; Chen et al. 2015a; Zhou et al. 2012a; Enck et al. 2014; Grace et al 2012; Mariconti et al. 2017; Aafer et al. 2013; Yang et al. 2014). In these papers, the effectiveness of the detection is evaluated using apps randomly selected from app stores. We should, however,

raise two important points with respect to these investigations. First, none of these methods is based exclusively on the frequency of API calls. For example, Aafer et al. (2013) used other features including parameters passed to API calls, or Mariconti et al. (2017) used a call graph of API calls. Furthermore, it is still unclear how well they will work when the testing set contains the original version of repackaged apps. Indeed, as shown in Fig. 9 and Fig. 10, highly frequent API calls in added files in repackaged apps are similar to those in deleted files in original apps. Therefore, the accuracy of any proposed malware detection based on API calls must be tested with dataset that contains matching pairs of original as well as repackaged apps.
Finally, note that the literature does not differentiate between adware with other types of malware; our finding F3 shows that adware samples form a large portion of malware (77.84%). Therefore, the similarity of API calls in added and deleted files described above may occur in large portion of malware of type adware.

Analysis, discussion and implication: Findings F4, F5 and F6 further suggest some additional assertions about adware. First, permissions relating to in-app purchases suggest that adware may generate revenue for their creators through the use in-app purchases, especially since every repackaged app in our dataset is free. Second, since API calls deleted from original apps and API calls added in repackaged apps exhibit a very similar distribution, detection of adware based only on the distribution of API calls seems impractical. Third, as shown in Fig. 10, some API calls, present in the files that are deleted from the original apps, are related to the connection to a remote server. This confirms the expectation that adware developers need to manipulate those API calls or delete them. However, we leave a complete study of the parameter of API calls for future work. It is very promising to understand to what extend the value of parameters in API calls are changed during repackaging. Fourth, the deleted API calls include API calls that establish a connection to a remote server and in doing so might reveal the origin of the app. In this regard, any kind of code obfuscation that hinders code comprehension can be useful in preventing repackaging. Since we observed in F5 that a considerable proportion of adware does not require permissions to perform their operation, it is not practical to rely upon permissions for adware detection. Finally, since the similarity of API calls in manipulated part of the code in adware samples and their original version is very high, we also suggest that the developers of adware detection methods test their approaches' accuracy over a testing dataset that contains both original and repackaged version of the malware. However, an effective detection could be performed by recognizing these apps as repackaged clones of other available apps. In RQ5, we propose an app classification scheme that allows the detection of clone apps to be performed in tractable time.

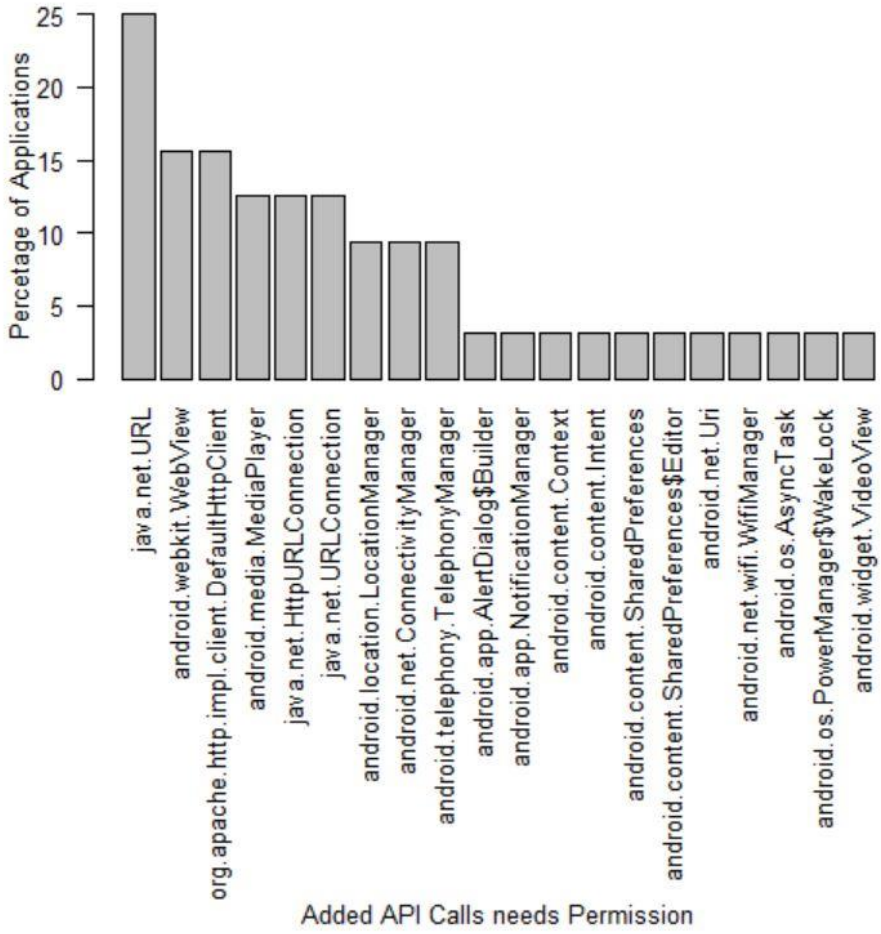

Fig. 9 Top 20 frequently APIs requiring permissions to execute, added in repackaged adware samples in comparison to their original apps

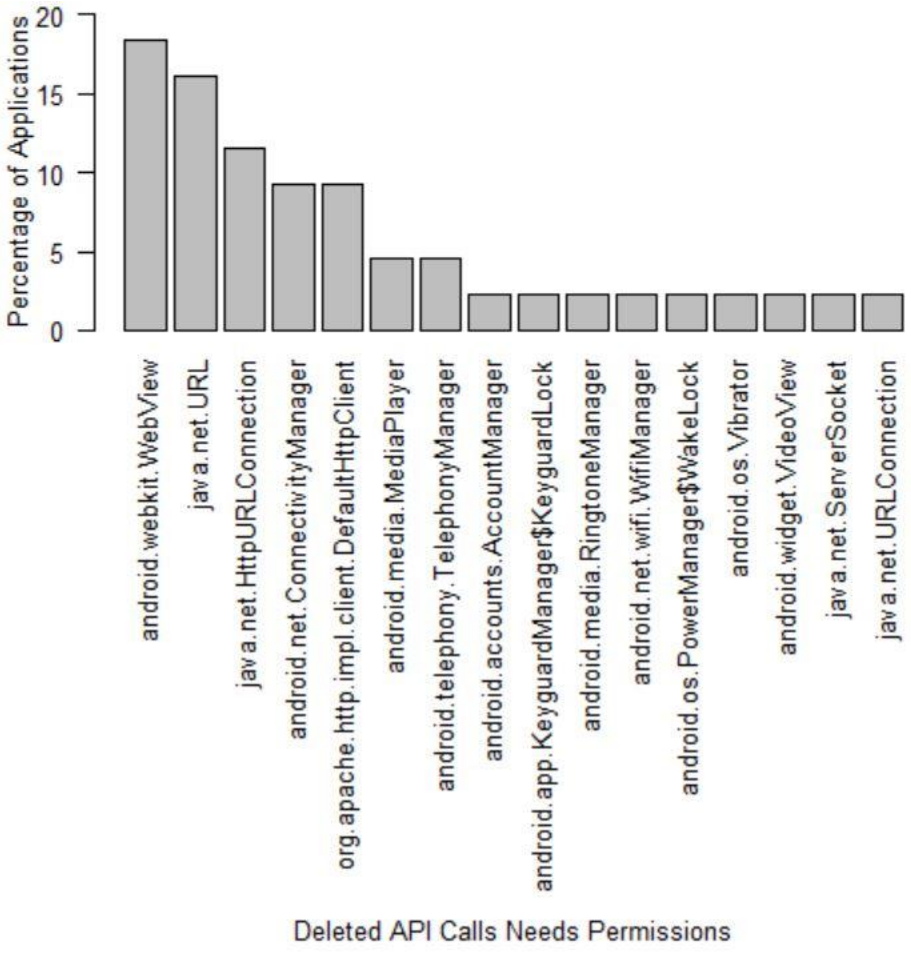

Fig. 10 The top 20 frequently APIs requiring permissions to execute that are deleted in repackaged adware samples in comparison to their original apps

4.3 RQ3. Which types of apps have been exploited for repackaging?

Motivation: Previous literature identified the popularity of an app as the main criterion for an app being chosen for repackaging and for spreading malware (Li et al 2017b). Hence, we examined the popularity of the apps exploited for repackaging in our dataset.

Apart from the popularity of apps, we also sought to shed light on code comprehension, as it applies to embedding malware into apps. Obfuscation tools, such as Dexguard[3], can make it harder to understand the code, for example by changing the name of methods and variables and by using reflection. There is also the question of whether one needs to understand the code to manipulate it. We studied the use obfuscation in the apps of our dataset to determine if its usage negatively correlated with repackaging.

Approach: We defined three metrics by which to measure the popularity of an app: the star rating of the app, its number of downloads, and the market where the app is located.

In the Google Play Store, users can rate apps by assigning them from 1 to 5 stars. The average number of stars is called the aggregate rating and it is recorded in the app's page in the Google Play Store. We used the aggregate rating of each app as our

[3] http://www.guardsquare.com/en/dexguard

first metric of popularity. We were unable to extract this information directly from the Google Play Store because many apps in our dataset are no longer available. However, as mentioned in Section 3.3, this information was available in the Kaggle dataset (Leka 2016) gathered in May and June 2014, but, unfortunately, for only 686 original apps out of the 2,776 original apps in our dataset.

We used the number of downloads as a second metric for measuring the popularity of apps. The third criterion is the market where the original apps are located. This information is provided by AndroZoo for all 2,776 original apps. Note that an app may be published in more than one market. We considered every market in which an app is published in our statistics. For example, if an app is available in the Google Play Store as well as in Anzhi, we count it once in the Google Play Store and once in Anzhi.

Finally, we studied whether obfuscation (name changing, reflection, and dynamic loading) is present in the apps. The most popular obfuscation tools rename program methods using a simple alphabetic change. Methods are first renamed with a single letter such as *a*, *b*, *c*,…. When the alphabet is exhausted, the algorithm proceeds with two letter names such as *aa*, *ab*, ... and so on. We used soot to perform static analysis over the dexcode of apps to find evidence of obfuscation, such as names in the above-described format. The presence of methods using the class "java.lang.reflect" and "dalvik.system", which indicate the presence of reflection and dynamic loading in the app code, respectively, are taken as evidence of obfuscation. The later assumption is justified by the fact that according to the Android developer documentation (2018), every class of the dalvik.system implements a classloader or some classloader functionality. Several other papers identify dynamic loading as the purpose of the dalvik.system class (see for example (Maly and Kriz 2015), (Zhao and Qian 2018), (Vigna et al. 2014), and (Zhou et al. 2012b)).

We compared the results for the original apps with the information obtained for all other free apps available in Kaggle dataset. Kaggle provides information including app's name, publication date, file size, star rating, number of downloads, package name and price. There are 245427 free apps (with price 0.0). We examined free apps exclusively because all the repackaged and original apps in our dataset (AndroZoo dataset) are also free.

Findings:

F7: Apps with higher star ratings are not more likely to be repackaged. Fig. 11 presents the boxplot of the aggregate star-rating for original apps that have been repackaged compared with that of the free apps in Kaggle dataset. To mitigate the threat of time inconsistency, we grouped the apps by year of release date, since apps that have been in a store longer time are likely to have received a longer number of user reviews. This comparison reveals that median aggregate star-rating for both is almost identical. However, the aggregate star-rating of original apps varies in a much narrower range than that of all apps. However, the Wilcoxon-Mann-Whitney test only shows a significant difference for the aggregate rating between original apps and all free apps (with p-value=0.03054) in 2012. It does not show a significant difference for apps in 2013 and 2014 because the p-value is above 0.05. These results do not fully support previous research that indicated that popular apps are more likely to be repackaged (Li et al 2017b) based on the metric aggregate star-rating.

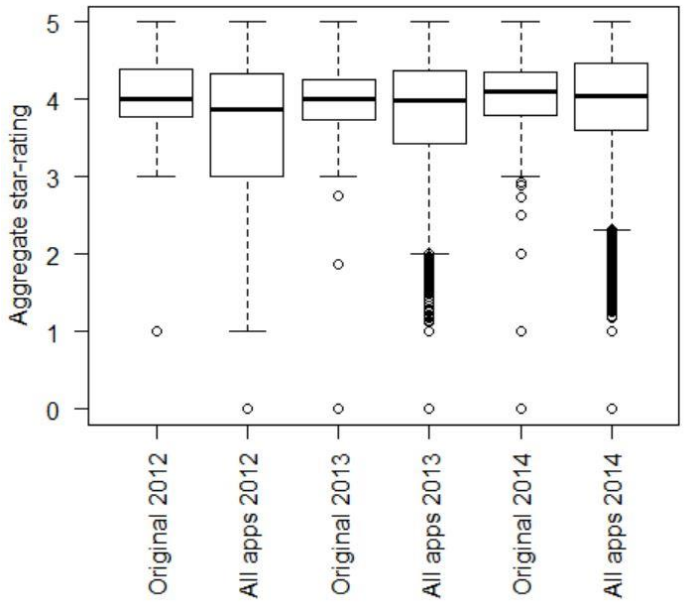

Fig. 11 Boxplot showing the aggregate rating of the original apps vs all the apps

F8: Apps with a high number of downloads are more likely to be repackaged. Fig. 12 shows the distribution of original apps versus all free apps according to their number of downloads. To mitigate the threat of time inconsistency, the apps are grouped according to their release date. Fig. 12 shows that the number of downloads for original apps is skewed to the right, where the number of downloads is increased. But, for all apps, the number of downloads is skewed to the left (where the number of downloads is lower). Moreover, using Wilcoxon-Mann-Whitney test over the set of number of downloads for original apps and number of downloads for all free apps in the same year shows that these two sets are significantly different with the p-value<0.1e-6. These results were expected, and confirm that malware developers target popular apps in order to spread malware.

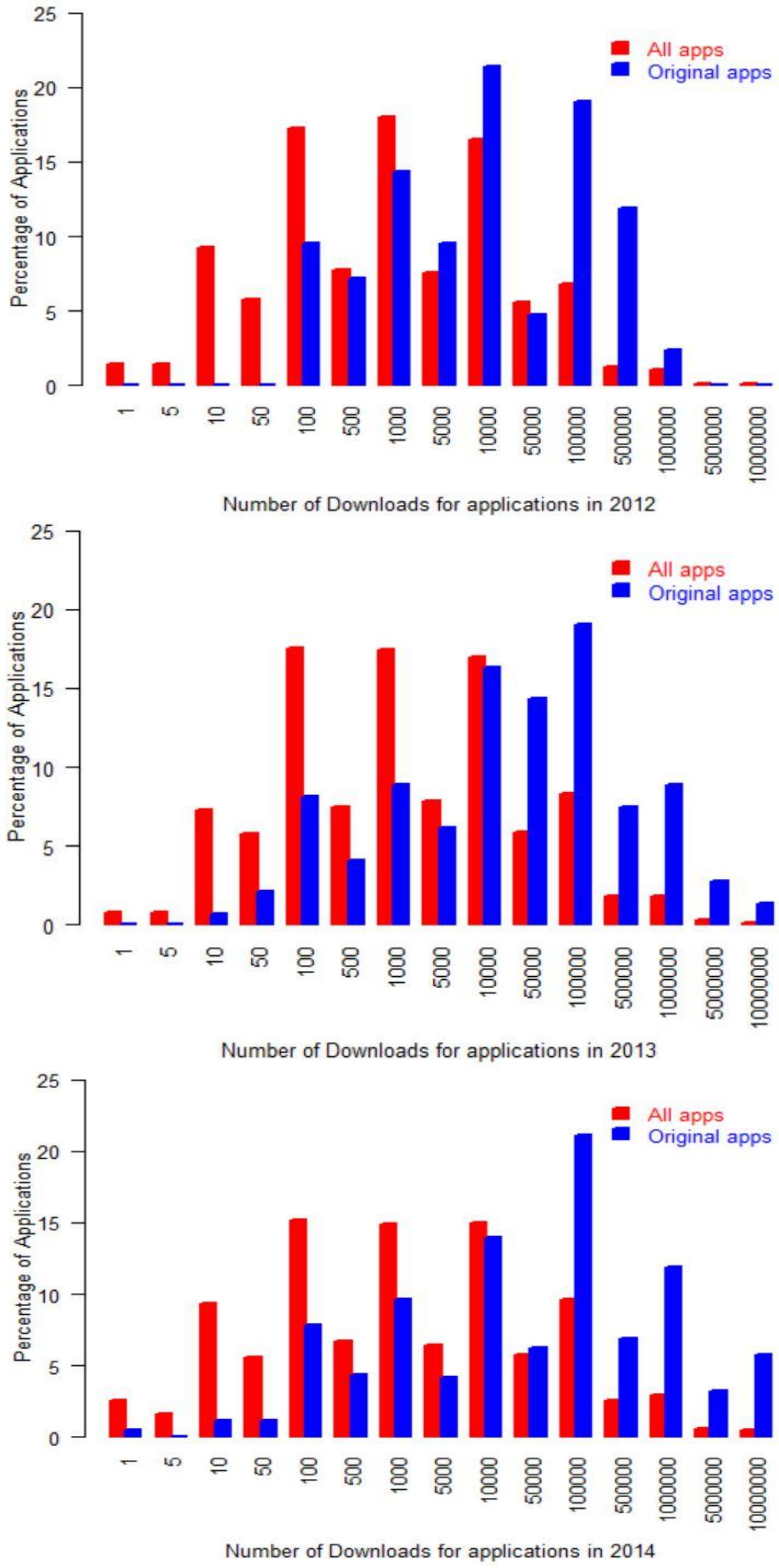

Fig. 12 Frequency of the number of downloads of apps in a variety of ranges

F9: Apps obfuscated with name changes are less likely to be repackaged. Table 4 shows the percentage of original and repackaged app pairs subject to various obfuscation techniques including name changes, reflection, and dynamic loading. The results of reflection and dynamic loading do not seem to yield meaningful information. However, the results related to name-changing are striking. Indeed, 95.94% of app pairs exhibit no name changes between the original code and that of the repackaged app. Moreover, in an additional 0.16% of pairs, the original apps do not exhibit obfuscation though name changing while the repackaged version does. In total, 96.1% of original apps in our dataset are not obfuscated though method name changes. Dong et al. (2018b) studied the obfuscation techniques used in Android apps. We summarized their results in Table 3. It shows that 43% of apps in Google Play and 73% of apps in third-party contain obfuscation of type name changing. Therefore, 57% and 27% of apps in Google Play and third-party apps, respectively, are not obfuscated. By using the Chi-square test, the ratio of original apps that are not obfuscated with type name changing is significantly different from the ratio of apps in Google Play with p-value = 2.839e-15. It is also significantly different from third-party apps with p-value=1.268887e-54. This result suggests that the designers of repackaged apps deliberately target apps whose code has not been obfuscated and that such apps are much more likely to be repackaged.

Table 3 Obfuscation techniques (Dong et al. 2018b)

App Store Name	Obfuscation	
App Store Name	Name changing	Reflection
Google Play apps	43.0%	48.3%
Third party apps	73.0%	49.7%

F10: Dynamic loading is added during repackaging up to 1/6 (16.46%) of the time. We studied two other types of obfuscation, namely reflection and dynamic loading, and summarized our results in Table 4. The first two rows show that the existence of each type of obfuscation in the original and repackaged app. It shows that 16.46% of repackaged apps exhibit dynamic loading while the original app does not (last row). This suggests that the malicious operations may not statically present in the code of the app and are instead loaded later at runtime. This result confirms the importance of dynamic analysis to ensure the security of mobile devices.

Table 4 Percentage of apps pairs that exhibit or do not exhibit each variety of obfuscation techniques

	Obfuscation type		
	Name changings	Reflection (%)	Dynamic Loading (%)
Percentage of pairs for which obfuscation is present both in the original and in the repackaged version	3.55	99.09	52.65
Percentage of pairs for which obfuscation is present neither in the original nor in the repackaged version	95.94	0.60	30.56
Percentage of pairs for which the original app exhibits obfuscation but the repackaged version does not	0.34	0.04	0.33
Percentage of pairs for which the original app does not exhibit obfuscation, but the repackaged app does	0.16	0.27	16.46

Analysis, discussion and implication: We studied the popularity of apps based on two metrics including aggregate star-rating and number of downloads. Based on the star-rating presented in F7, we cannot completely support the idea that popular apps are selected for repackaging. But, according to finding F8, the number of downloads in original apps which are selected for repackaging exhibits a statistically significant difference with that for all free apps. We conducted a correlation test (Pearson and Spearman) regarding these two metrics but found a very low correlation between them. This shows that having a high star rating is not necessarily accompanied with having a high download rate. Moreover, among the repackaged apps there are still a considerable number of apps that do not have a high star rating or a high number of downloads. These results have a clear implication for the effort to detect repackaging: such efforts simply cannot be focused on highly popular apps. Furthermore, there is also the question of which available popular apps are more likely to be selected for repackaging. Result F9 show that apps that do not exhibit name changes obfuscation are most likely to be repackaged. Therefore, while the popularity of an app

tempts attackers to repackage it, they still prefer apps whose code is not obfuscated. This result reconfirms the importance of obfuscation and of research in improving obfuscation techniques as it relates to protecting the revenue of apps. We believe that using the alphabet for name changing can still reveal information about the target program since there is a defined ordering in which the classes are processed.

F10 presents a supplementary result, not related to the above research question, which confirms that static analysis alone is not enough to detect the presence of malware in repackaged apps, since the malicious elements may be loaded at runtime.

4.4 RQ4: Why users download the repackaged apps when the original version is freely available?

Motivation: To propose approaches to defeat and prevent repackaging, it is useful to know the reasons why users download the fake version of an app even though the original version is available for free. Clearly, users do not know that they are downloading a fake version, especially since they are using trusted stores like Google Play Store.

Approach: In understanding user behavior for deciding to download an app, we must consider a variety of metrics. A complete review of all factors affecting user behavior is beyond the scope of this paper, which focuses on a statistical analysis of the apps' manipulations. Nonetheless, there are still some findings that can help us to partially answer this research question.

Findings:

F11: Repackaged apps originate from a variety of markets including Google Play Store. We found that most of the original apps that have been repackaged were originally published in the Google Play Store. Fig. 13 shows that 58% of apps were published on this platform. Anzhi, 1mobile and Appchina are the other favorite markets. While the Google Play Store is a popular store for Android apps, repackaging exploiters seem to use it as the main place for browsing and finding target apps to spread malware.

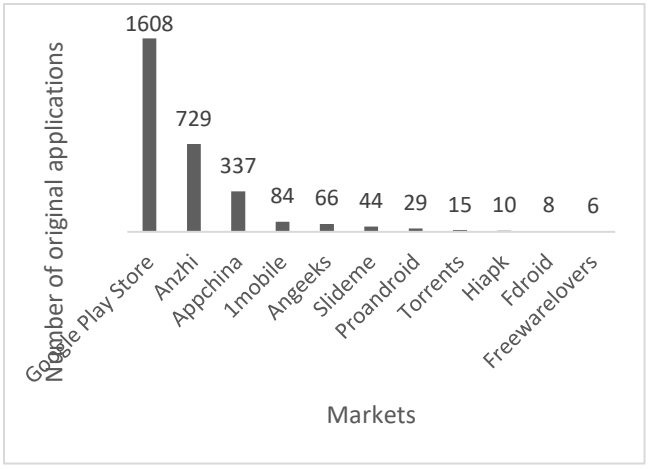

Fig. 13 Number of original apps according to the market where they are published

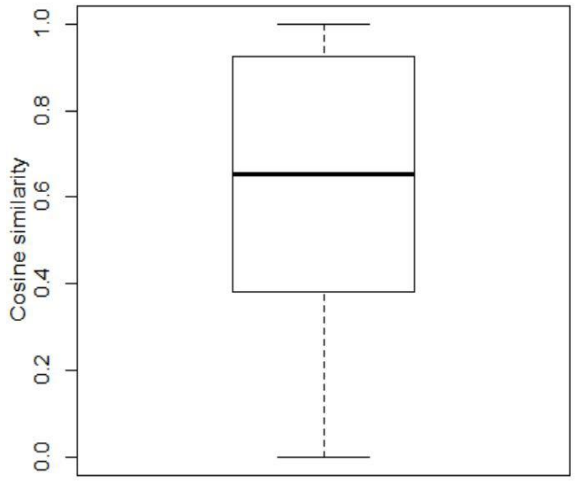

Fig. 14 Boxplot showing the similarity score between the name of the original app and the name of its repackaged version

F12: A considerable number of repackaged apps are published again in the same store as the original ones. We also find that 13,881 out of 15,296 (90.75%) of repackaged apps were republished again in the same store, and that the number published in other stores is 1709. Note that 294 of these were published in both the same and other stores. Therefore, those apps are repeatedly counted in the number of repackaged apps stored in the same store and the number of apps published in other stores.

F13: Repackaged apps' names are usually very similar to the name of the original app. We extracted the app's name by using Androguard. Fig. 14 shows a boxplot that represents the cosine similarity measure (Singhal 2001) between the name of repackaged apps and the name of their original app. The figure shows that more than 50% of the original and repackaged pairs have a name similarity greater than 65.30%, which suggests that developers of repackaged apps deliberately maintain similar

names to the original apps, perhaps to dupe users. For example, an app whose name was “Duck Shooter” in the original version was repackaged and renamed “Shoot My Duck”. The cosine similarity between these names is 84.27%.

There remains a considerable number of app pairs with different names. In fact, 35.40% of the app pairs exhibit a name similarity of less than 0.5 based on the cosine distance metric. We used Google language detection library in Python[4] to detect the languages in which the name of apps is written and then compared the languages for pairs of repackaged and original apps. In total, there were 796 original apps and 6733 repackaged apps that were in languages other than English. There are 1,164 out of 15,296 pairs (7.60%) where the language of the app’s name in the original and repackaged version is different. This result suggests that the repackaged app’s language may be another motivation for users to download the repackaged version rather than the original. More research is needed in order to clarify the user’s motivation in downloading apps in languages other than English. We could examine, for example, whether the layout of the app catalog or user reviews favor these alternate language apps over their English counterparts.

F14: Repackaged apps in the Google Play Store have a high star rating and a high number of downloads. In the Kaggle dataset, we found the data for 2035 repackaged apps that exclusively come from the Google Play Store. Fig. 15 and Fig. 16, respectively, present the aggregate star rating and number of downloads for those apps. Fig. 15 depicts that 75% of repackaged apps have an aggregate rating greater than 3.66. Also, 50% have a rating greater than 3.96, and 25% have a rating greater than 4.31. Moreover, using the Wilcoxon-Mann-Whitney test doesn’t support the hypothesis of a statistically significant difference between the star ratings of original apps and that of repackaged apps. Furthermore, as can be seen in Fig. 16, 58% of apps have more than 1000 downloads. These are considerably high download number and high star rating for repackaged apps and suggests that repackaged apps are popular and as a consequence, that they are successful in spreading malware.

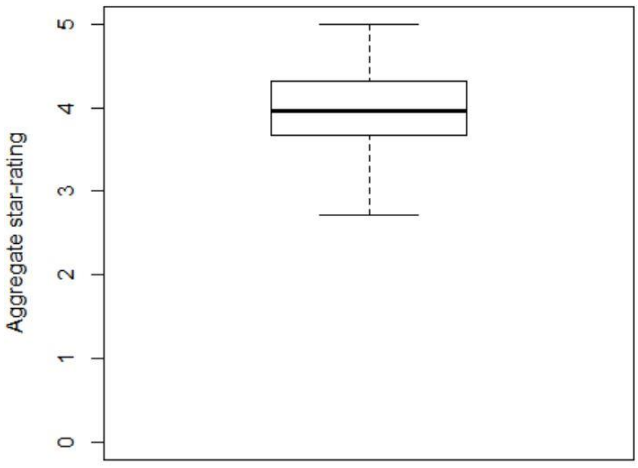

Fig. 15 Boxplot showing the star rating for the repackaged apps

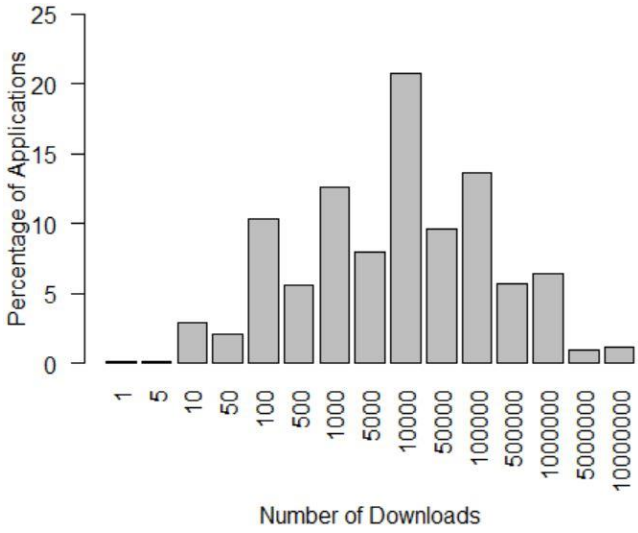

Fig. 16 Distribution of repackaged apps based on the number of downloads

Analysis, discussion, and implication: Result F11 shows that most repackaged apps were originally downloaded from Google Play Store. This suggests that the repackaging exploiters use the most popular store to obtain a list of popular apps and thus exploit the users’ tastes. The interesting point here is that based on F12, the repackaging exploiters target the users of the same store in distributing the repackaged versions. Moreover, result F14 indicates that they are often successful in deceiving users, getting them to download their repackaged apps. On the other hand, according to result F13, there is high similarity between the app name in original and repackaged versions. Similar names increase the chance for the repackaged app to be listed by search engines when a user is looking for the original app. Moreover, the date of the publication of the repackaged version is,

[4] https://github.com/Mimino666/langdetect

obviously, later than its original pair in the same store. These statistics suggest that users may be fooled, assuming that the repackaged app is the latest version or new release of the original app.

For the repackaged apps published in other stores, other than the original version's store, more study is needed before conclusions could be reached. For example, it would be interesting to study the language of the layouts in the app or the language in which reviews are written.

4.5 RQ5. How are an app's attributes modified in the repackaged version?

Motivation: Knowing about the changes in the attributes of an app after repackaging can guide researchers in proposing approaches for detecting repackaging. While investigating this research question, we focus on the following attributes: the APK name, the size of the APK file, the app components, and the list of permissions.

Approach: We extracted an app's attributes from the manifest.xml file of each app and compared the attributes of the original apps to those of the repackaged versions. We compared the number of components in the original and repackaged apps. We also compared the name of each component, using the cosine similarity distance (Singhal 2001) to measure the extent by which the two names are deemed similar. We proceeded in like manner for permissions, i.e., we compared the number of permissions as well as the name of permissions.

We used Androguard (Desnos 2015) to extract the following attributes from each app: app size, app name, the number of components and their names, and the apps' permissions. This data is extracted from the Androidmanifest.xml file that accompanies each Android app and contains metadata about the app. It notably includes the app components' names and permissions. Table 5 shows an example of the attributes of an app that are extracted by Androguard from the Androidmanifest.xml file.

Table 5 An example of the components of an app

Attribute Name	Attribute Value
Activities	com.ansca.corona.CoronaActivity|com.ansca.corona.CameraActivity,com.ansca.corona.VideoActivity|com.openfeint.internal.ui.IntroFlow,com.openfeint.api.ui.Dashboard,com.openfeint.internal.ui.Settings|com.openfeint.internal.ui.NativeBrowser|com.adknowledge.superrewards.ui.activities.SRPaymentMethodsActivity,com.adknowledge.superrewards.ui.activities.SRDirectPaymentActivity,com.adknowledge.superrewards.ui.activities.SROfferPaymentActivity,com.adknowledge.superrewards.ui.activities.SRWebViewActivity,com.zong.android.engine.web.ZongWebView
Services	com.zong.android.engine.process.ZongServiceProcess
Receivers	com.ansca.corona.purchasing.GoogleStoreBroadcastReceiver
Content Providers	com.ansca.corona.FileContentProvider
Permissions	android.permission.INTERNET|android.permission.READ_PHONE_STATE|android.permission.ACCESS_NETWORK_STATE

Findings:

F15: The size of the APK file may or may not increase after repackaging. Fig. 17 shows a plot of the size of the original app' files in comparison with the file size of the repackaged version. The figure shows that the size may or may not increase after repackaging, suggesting that the APK file size criterion alone is not an indicator of repackageability. Note that a strong similarity is obtained when the (x, y) points of the graph are positioned on the blue line.

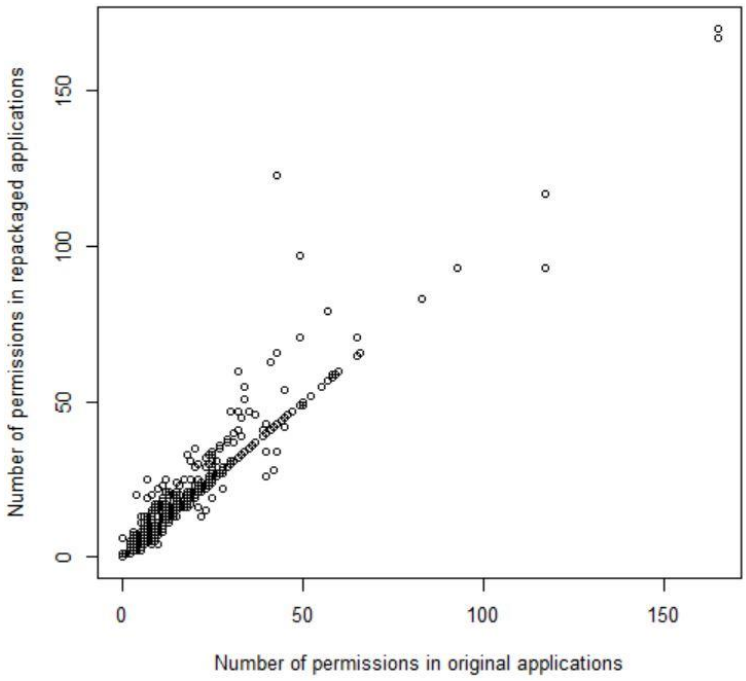

Fig. 17 Comparison of the APK file size of the original apps with the file size of the repackaged versions

F16: Repackaged apps have a similar number of components in comparison to their original pair. Fig. 18 shows the number of app components in the original apps compared with that in the repackaged apps. We extracted the number of components from the apps' Androidmanifest.xml file. Note that app developers should write the names of activities, services, and receivers in Androidmanifest.xml file. However, content providers can be created at runtime. We found that the number of activities, services, receivers, and content providers did not change much in the repackaged apps. The number of activities is the same in 14,366 out of 15,296 app pairs (93.92%). The number of services is the same in 15,016 out of 15,296 app pairs (98.17%). The number of receivers is also the same in 15,043 out of 15,296 pairs (98.35%). This also applies to the content providers where the number of content providers is the same in 15,241 out of 15,296 app pairs (99.64%). These results demonstrate that the number of components of an app is not an indicator of repackageability. This information can however be useful in finding similar apps.

F17: Components' name does not change in most of the repackaged apps in comparison to their original pair. We also examined the changes in component names. We found that among 14,366 original-repackaged app pairs with the same number of activities, 11,527 of these (80%) have identical names in the original and repackaged versions.

There are 7,466 original and repackaged app pairs that have at least one service, and that also have the exact number of services in the original and repackaged versions. Furthermore, we found that in 7,428 app pairs (99.49%) the names of all of the services are identical. There are 9,914 original and repackaged app pairs that have at least one receiver component and that also have the same number of receivers in the original and repackaged versions. Likewise, we found that in 9,857 app pairs (99.43%) the names of all of the receivers are identical. Similarly, we found that 481 app pairs out of 492 apps containing content providers (97.96%) have identical content provider names. These results are illustrated in Fig. 18.

We used cosine similarity to compute the similarity measure. For each pair of original and repackaged apps, we compared the component names. For example, we compared the names of activities in the original app with the name of the activities in the repackaged app. Based on the definition of cosine similarity, we calculate the similarity by measuring the cosine of the angle between two vectors. For example, for activities, term frequency of the activity's name is the vector. Note that the permutation of activity names for each app will not alter the result. For example, the following two sets of activities' name have a cosine similarity equal to one:

"com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.google.ads.AdActivity|com.appbrain.AppBrainActivity", and
"com.revmob.ads.fullscreen.FullscreenActivity|com.google.ads.AdActivity|com.appbrain.AppBrainActivity|com.abarakat.webview.WebViewActivity"

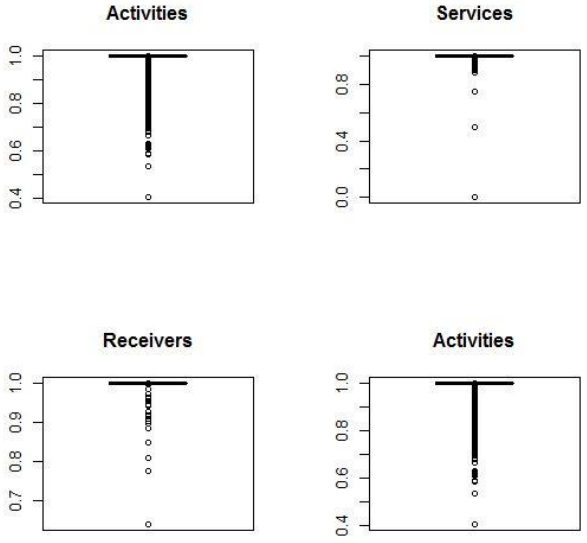

Fig. 18 Boxplot showing the cosine similarity measure between the name of components in the original apps and the names of the components of their corresponding repackaged apps

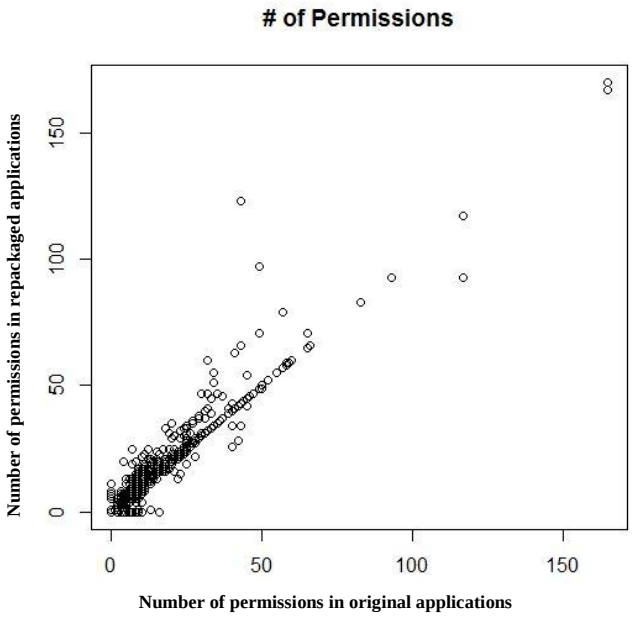

Fig. 19 Number of permissions in original apps compared to the repackaged apps

F18: The number of permissions in repackaged apps is the same as that in the original apps in 93.34% of the cases. Fig. 19 compares the number of permissions in repackaged apps with the number of permissions in their original pair. The figure shows that most of them have a similar number of permissions. Indeed, there are 14,109 pairs that have at least one permission and that also have the same number of permissions in the original and repackaged versions. In this group of 14,109 pairs, we find 14,045 pairs (99.55%) that have the same permissions in the original and in the corresponding repackaged version. This result is different from the result presented by Li et al. (2017b) where they found out that more permissions are added in most of repackaged apps. However, they also mentioned that there are some apps that some permissions added in repackaged one while such permissions exist in original one too. Note that the result we provide here is based on the distinct permissions exist in an app.

In the remaining 64 app pairs out of 14,109 pairs, however, some permissions were altered. We investigated the changes in the permissions requested in these cases. Fig. 20 shows the permissions deleted from the original apps and Fig. 21 shows the permissions added in repackaged versions. Two important conclusions must be stressed. First, the figures show that the deleted permissions are mostly specific to the app and start with the app's package name such as "AppPackageName.permission.C2D_Message". This permission relates to Android cloud device messaging (C2DM service) and allows the developer to push data to the app installed in the user's device from a server. Note that C2DM service was discontinued for existing apps and shut down completely in October 2015 (Google Inc. 2015) and that this permission was replaced with newer C2DM permission. Clearly, it connects the app to a new server. Second, in 5 samples, permission JPush_Message and CHECK_LICENSE was deleted. The permissions are needed if a remote server needs to connect an app and check the license of the app. This deletion suggests that malware developer blocked the connection of the original developer in the repackaged malware.

There are 663 pairs where the repackaged version has more permissions than in the original version. Fig. 21 shows the frequency at which certain permissions were added. Since the permissions were added by malware developers, we can be certain that they are used to perform malicious operations. We provide this list of permissions to the public and to interested researchers working in malware detection. The top ten permissions added, in order of frequency, are the following:

android.permission.GET_TASKS,
android.permission.SYSTEM_ALERT_WINDOW,
android.permission.WAKE_LOCK,
android.permission.VIBRATE,
android.permission.READ_PHONE_STATE,
android.permission.ACCESS_NETWORK_STATE,
android.permission.WRITE_EXTERNAL_STORAGE,
android.permission.ACCESS_COARSE_LOCATION,
android.permission.INTERNET,
android.permission.ACCESS_WIFI_STATE.

Analysis, discussion and implication: In summary, our findings show that repackaged apps maintain almost the same number of components with the same names after repackaging. This feature can be used to detect repackaged apps. Traditional approaches for detecting repackaged apps rely on pairwise comparisons of apps to identify similar apps (e.g., (Zhou et al 2012a), (Shahriar and Clincy 2014), (Crussell et al. 2012), and (Crussell et al. 2015)). The main drawback of these approaches is their high time complexity since each app must be compared with all other apps. Taking advantage of the findings in RQ4, in the next subsection, we propose a novel app indexing scheme that relies upon the name of activities to cluster apps with similar activity names.

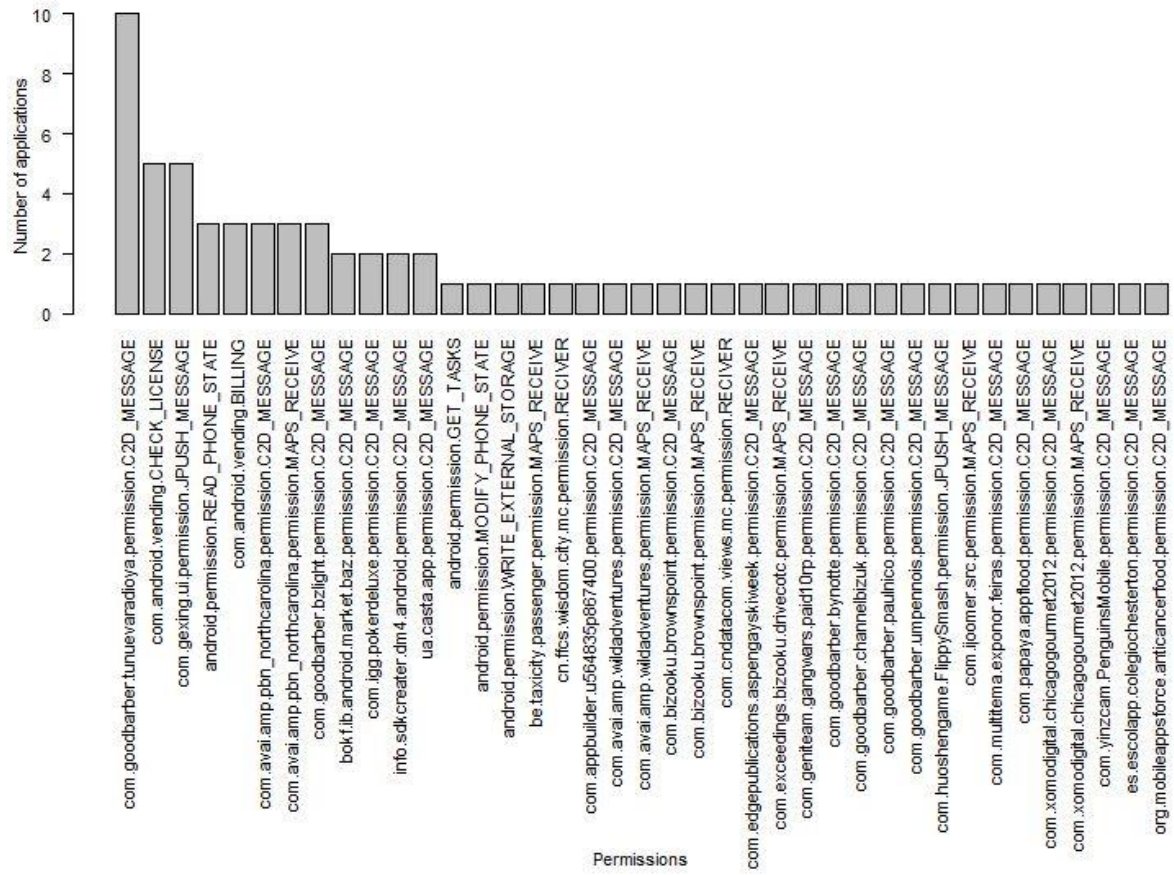

Fig. 20 Permissions deleted in repackaged apps

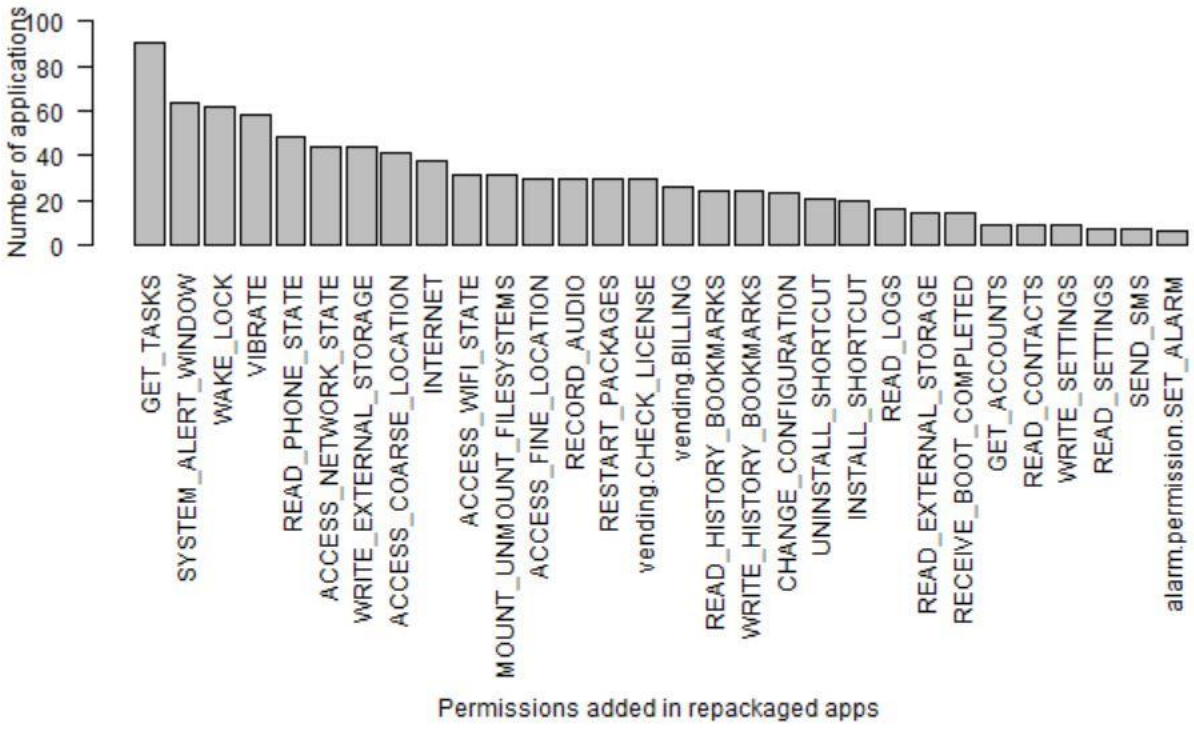

Fig. 21 Permissions added in repackaged apps

Indexing Scheme

The aim of this scheme is to decrease the number of comparisons needed to find repackaged apps when using the indexing system. We used a piece-wise hashing strategy in which apps that have activities with almost similar names are assigned almost similar hash values. We used Ssdeep (Kornblum 2006) for hashing the name of activities. Then, following a method proposed by Winter et al. (2013), we used the n-gram of the hash as the look-up key referring to a row in our indexing table. We thus obtain a table where each row refers to an array that contains the ID numbers of apps and the row number is an n-gram generated by hashing the name of activities. Here, we used 7-gram because Winter et al. mentioned that, in Ssdeep similar hashes must have common 7-grams (Winter et al. 2013). Note that we have removed the activities coming from ad libraries since they are similar in a considerable number of apps and thus increase the number of comparisons though not providing much differentiation between apps. These activities are written in the ad packages. We obtained a list of ad packages from a study by Book et al. (2013). Table 6 shows the names of activities of a repackaged app (repackaged app1) and its original pair (original app1) and the way that they are used for indexing. As it is shown in Table 6, repackaged app1 with SHA 89EF7AC73CD35E588DAFC6646436BE586299EBDB246452E8D5E20B7233BBD1B4 is the repackaged version of the original app1 with SHA AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2. Both of them have activities with the same name. Therefore, even after removing activities related to ad libraries, they have a similar hash, and they will be recorded in the same row of the index.

Another original app has SHA D7546FB04BC9ABB901017848BC49251AF038DC7E17E013523E5567AB6FFE0BC5. In what follows we refer to it as original app2, and it is not the original version of repackaged app1. Some of the activities of this app share a similar name to the activities in repackaged app1. Therefore, by using n-gram of a hash of activities name for indexing, repackaged app1 and original app2 will be also indexed in the same row as well.

Table 6 An example of the name of activities used for indexing

	Original app1 with SHA: AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2 *	Original app2 with SHA: D7546FB04BC9ABB901017848BC49251AF038DC7E17E013523E5567AB6FFE0BC5
Activities' name separated by delimiter "|"	com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.google.ads.AdActivity|com.appbrain.AppBrainActivity	com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.google.android.gms.ads.AdActivity|com.appbrain.AppBrainActivity|com.bqquqi.cdueey177143.MainActivity|com.bqquqi.cdueey177143.BrowserActivity|com.bqquqi.cdueey177143.VDActivity
Activities after removing ads	com.abarakat.webview.WebViewActivity	com.abarakat.webview.WebViewActivity|com.bqquqi.cdueey177143.MainActivity|com.bqquqi.cdueey177143.BrowserActivity|com.bqquqi.cdueey177143.VDActivity
Hash of activities	QuEXnfA/AHzMAX3Qcn	QuEXnfA/AHzMAX3QcS6LXOSIMyQGRZcS6LXOSIMZH/qARcS6LXOSIMfRcn
n-grams	QuEXnfA, uEXnfA/,EXnfA/A, XnfA/AH,…	QuEXnfA, uEXnfA/,EXnfA/A, XnfA/AH,…

*Repackaged app1 with SHA 89EF7AC73CD35E588DAFC6646436BE586299EBDB246452E8D5E20B7233BBD1B4 has activities with the same name as Original app1 with SHA AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2

We computed the Ssdeep hash for all original and repackaged apps in our dataset. The hash is in Base64 encoding and each digit in Base64. Identifying the best piece-wise hash functions or finding the best indexing approach and evaluating the performance of the scheme is beyond the scope of this paper. Here, we seek to show that using the names of activities for indexing can decrease the number of comparisons needed for pairwise comparison of apps. In traditional pairwise comparisons each repackaged app (15,976 apps in our dataset) must be compared with all original apps (2,776 apps in our dataset). We calculated the number of pairwise app comparisons needed when using our new indexing system. The result is given in Fig.

22. The median is 12; which means that on average, each app must be compared with 12 other apps. It may also happen that two apps are indexed together in more than one row. When this occurs, we compare the two apps only once.

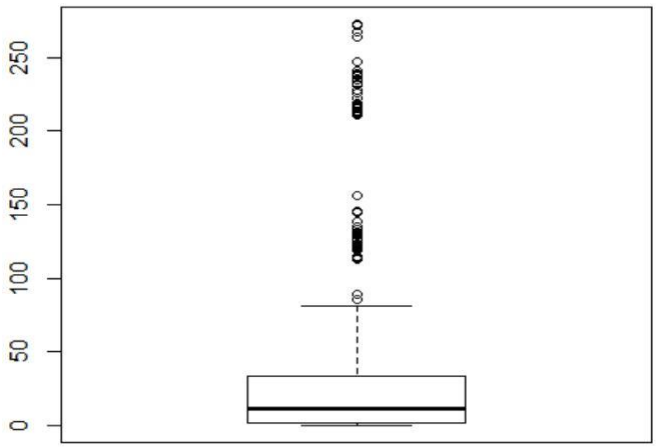

Fig. 22 Number of comparisons for each repackaged app needed in the proposed indexing system

5 Related Work

Repackaged apps. The literature relating to Android app repackaging can be categorized into three groups. In the first group are studies that focus on detecting repackaged apps. The second group consists of studies on preventing apps from being repackaged. The final group includes literature describing statistical studies on the state of repackaged apps.

The early work on detecting repackaged apps relied upon illuminating (pair-wise) comparisons between two apps to see if they showed similar behavior and attributes. Researchers proposed using a variety of factors to identity repackaged apps including: instruction sequences (Zhou et al. 2012a), specific opcodes (Shahriar and Clincy 2014), static data dependency (Crussell et al. 2012) (Crussell et al. 2015), Abstract Syntax Tree which contains information such as the number of arguments and the methods evoked by each method (Potharaju et al. 2012) and k-grams of binary opcode sequences (Hanna et al. 2012).

All of these approaches require performing a static analysis of the app code and suffer from obfuscation techniques such as reflection and encryption. To combat the limitations of static analysis, a dynamic analysis approach was proposed. Guan et al. (2016) use input-output states of core functions in the app and then compare function and app similarity. Wu et al. (2015) modeled the app's behavior from the HTTP traffic. Aldini et al. (2015) detect similar apps by processing the system call traces logged through the execution of an app.

Some researchers proposed detection techniques that rely upon a visual inspection of the appearance of the apps. Their work is based on the fact that manipulation of apps in repackaging is done in such a way as to maintain the same sense of "see" and "feel" of the original app in order to encourage users to download the repackaged apps. Zhang et al. (2014) used an approach called ViewDroid to detect similar apps by comparing the graph of apps' activity components. Sun et al. (2015) proposed an approach to check UI layout similarity. Soh et al. (2015) detect Android app clones based on the analysis of UI information collected at runtime. Chen at al. (2015a) found repackaged apps by detecting similar UI structure within apps. Zhauniarovich et al. (2014) proposed an approach to compare the resource on the app package (.apk file). Kywe et al. (2014) proposed detecting similar apps by applying text similarity and image similarity measurements. Yue at al. (2016) introduce RepDroid, a tool that relies upon traces of UI to detect repackaging. Their method benefits from the fact that it does not necessitate the apps' source code, and is thus resilient to obfuscation.

To overcome the extensive computation time of pair-wise comparisons, some researchers proposed approaches that require less computation time (Shao et al. 2014; Jiao et al. 2015; Wang et al. 2015; Chen et al. 2014). For example, Jiao et al. (2015) used an image processing approach to find the similarity between images in two apps. They utilized a new storage method in the form of features-application pairs, which stores apps with the same features in the same data structure. This approach decreased the number of comparisons needed. Zhou et al. (2013b) propose an alternate two-step method, based upon a static

analysis of the code. First, since the payload of malware is not an integral part of the app's functionality, they develop a module decoupling technique that can separate such payload from the code that implements the main functionalities of the program. Second, they extract a vector of semantic features from the module implementing the main functionalities of the program, which allows rapid detection of similar code.

Gonzalez (2014) proposed the idea of detecting repackaged apps based on the tell-tale signs left by tools that perform reverse engineering. There is an ordered list of string identifiers in an Android app's resources that refer to all strings in the app's source code. The order of strings in this list changes after repackaging, yielding evidence of repackaging.

Several methods have been proposed to detect repackaged apps and prevent the spread of the malware they may contain (Zhou and Jiang 2012; Khanmohammadi et al 2015). These include static as well as dynamic analysis approaches that extract some features from apps and classify malware samples. Several features are used to detect malicious behaviors, including API calls (Alazab et al. 2010; Islam and Altas 2012; Wu et al. 2012; Chen et al. 2015a; Zhou et al. 2012a; Enck et al. 2014; Grace et al 2012 ; Mariconti et al. 2017; Aafer et al. 2013; Yang et al. 2014; Khanmohammadi et al. 2017), strings contained in the apps (Sanz et al. 2012; Gonzalez et al. 2014), instruction code sequences (Shahriar and Clincy 2014), code chunks (Suarez-Tangil et al. 2014; Tian et al. 2016), permissions (Sahs and Khan 2012), ICC patterns (Xu et al. 2016), method call graphs (Hu et al. 2014; Gascon et al. 2013; Chen et al. 2015b; Sun et al. 2014) and system calls (Lin et al. 2013).

The second group of studies proposes approaches to protect against repackaging. Zhou et al. (2013a) provide a tool called AppInk, which uses watermarking to protect the app against repackaging. In general, watermarking is an approach for embedding a secret, known only to the developer, into the code in a way that doesn't change the semantics or the functionality of the app. At runtime, the presence of this secret is detected to provide evidence that the code has not been tampered with by a third party. AppInk adds code to the original app that uses a set of events for watermarking. This piece of code extracts the APIs called in part of the code and is a maintained secret. If the code is modified, its functionality will also be altered, and this change will be detected. The main limitation of their approach is that a manipulated Dalvik VM is required to perform the file checking. Ren et al. (2014) also used watermarking. They used the algorithm Self Decrypting Code (SDC) (Sharif et al. 2008) to change some conditions in the code, which checks equations with a constant. At runtime, the conditions are checked based on user input. In case the code has been changed by a third party, these conditions may be manipulated, and in such a case, the conditions added to the code will not be property evaluated. Therefore, the repackaging of the original code will be detected by the user at runtime. Zhou et al. in another work (Zhou et al. 2014) presented a new encoding for the app's Dalvik code; they modified the Dalvik virtual machine to decode the instructions before executing the code. Changing the Dalvik's code precludes the disassembler from decompiling the .apk of the app and reaching the source code. As this approach needs to make changes in Dalvik VM, which is part of the Android system, users must agree to carry out an update of the Android framework on their device before it can be used. Lee et al. (2014) also proposed an integrity check of the source code in the Dalvik VM by comparing the hash of the dex file with data provided by developers. Luo et al. (2016) propose a method to prevent application repackaging by injecting a large number of detection nodes in the code. The detection node is a segment of code that checks the public key of the app's certificate at runtime, and compares it to a hard-coded value. If repackaging is detected, the node will cause the application to crash. Interestingly, the authors suggest that the app be obfuscated, and include other methods, such as delayed malfunction in case repackaging is detected, in order to ensure that the scheme is not detected and foiled by an astute adversary. Zang et al. (2018) propose a similar approach and discuss multiple strategies to hide the presence of the detection code (termed logic bombs) from the adversary.

The last group of studies focuses on empirical studies of apps downloaded from markets to glean insights on the state of the art in repackaging. Mojica et al. (2012) put forth an empirical study on software reuse over apps downloaded from variety of categories in Google Play Store. They studied and identified similar classes in apps. Their findings show that app developers perform substantial software reuse. Mojica et al. (2014) also performed studies on class reuse in apps. They showed the percentage classes in apps inherited from a base class in the Android API, inherited from domain-specific base class, and computed the percentage of classes reused in each category.

Linares-Vásquez et al. (2014) studied the impact of third-party libraries and code obfuscation practices on estimating the amount of reuse by class cloning in Android apps. They showed that by excluding third-party libraries from the analysis, the amount of class cloning significantly decreased. They also found that obfuscation increases the number of false positives when detecting class clones. Viennot et al. (2014) proposed an approach for crawling the Google Play Store and downloading a large number of apps. In a recent study, Li et al. (2017) gathered large number of apps and proposed a code similarity metric in

order to identify repackaged apps and provided a dataset of original and repackaged apps. They also carried out an empirical study over the dataset and provided findings to answer some research questions. More particularly, they have provided information on the way the piggybacked apps are changed. In comparison to their work, we aimed to answer research questions by considering a set of findings related to such research questions.

Adware detection. Several studies have proposed approaches for detecting third-party libraries, including advertisement libraries, embedded in apps (e.g. (Ma et al. 2016) and (Backes et al. 2016)). Often, the motivation for this detection effort is that apps with embedded third-party libraries might also contain embedded malicious code. However, none of the studies we surveyed specifically detect malicious ad libraries versus benign ones.

Some studies have focused specifically on detecting malicious operations in Android adware. Liu et al. (2014) proposed a system called DECAF for detecting the placement of fraud of advertisements in apps. DECAF detects if an ad is shown in a manner that contravenes the relevant policies provided by ad networks such as AdMob (2019) or Microsoft Advertising (2019). For example, such policies may forbid an ad from being displayed too close from an app's UI button, in order to avoid a situation where the user clicks it unintentionally. Another fraud in adware is called on-click fraud. It occurs when adware fetches ads without displaying them to the user, or "clicks" on ads automatically. Crussel et al. (2014) focus on detecting on-click fraud by analyzing network traces. In a recent study, Dong et al. (2018a) proposed an approach that, apart from detecting on-click and placement fraud, also detects if the adware implement procedures for tricking users into unintentionally clicking ads while they are interacting with the UI elements. They present eight rules for identifying fraudulent behavior in adware. These rules were derived from a study of known adware operations as well as from the policies enforced by ad networks. In their proposed approach, snapshots of UI and network traces are recorded while the app is executing. They studied the recorded data to identify violations of these eight rules. Another malicious operation that adware may perform is to redirect users to web sites that contain malware of phishing. Rastogi et al. (2016) studied android apps that lead users the websites that host malicious operations through web links embedded directly in applications or via pages of advertisements that originate from ad networks. They developed an approach that detects these web sites and analyzed their content. Their study resulted in a number of findings that characterize such malicious web sites. For example, they find that a large number of malicious web sites deceive users by claiming to give away free products.

6 Threats to Validity

The selection of the dataset is one of the most common threats to validity for empirical studies. It is possible that the selected apps share common properties that we are not aware of and therefore, invalidate our results. We mitigated this threat by using AndroZoo, which was developed by researchers to advance the field of mobile security. The dataset was carefully built to contain Android apps from various categories. We also used Kaggle, which is a very rich and diverse source of Android apps.

Another threat to the generality of our study relates to the kind of repackaged apps present in the AndroZoo dataset. All of the repackaged apps in AndroZoo are detected as malware by at least one anti-virus. Therefore, our study is limited to repackaged apps containing malware. However, there may be repackaged apps where no malware is detected. Li et al. (2017b) categorized repackaged apps and named the ones having malware as *piggybacked apps*, and this is the only type of app examined as part of this study. In fact, our study includes only piggybacked apps. Obtaining a dataset of repackaged apps which do not contain malware and study them is left for future work.

Another threat is the drawback of tools used in our study such as APKtool (Wisniewski 2012) and Dex2jar which may vitiate the accuracy of results. We mitigated this threat by using findings based on statistics over a substantial number of apps.

To identify files in a repackaged app that are manipulations of files in its corresponding original app, files that are added in a repackaged app, and those of the original app that were deleted in the repackaged apps, we used two thresholds t1=70% and t2=98%. We identified these values through experimentation. Different values of t1 and t2 may impact the results. To mitigate this threat, we checked the files of a large number of repackaged-original app pairs to validate the results.

There is also the threat of time inconsistency where the time duration between the date of recorded meta data and release of apps are different for all apps. To mitigate this threat, we grouped the apps according to their release date in finding F7 and finding F8. However, the release date of the apps in the AndroZoo dataset may not be accurate, as was shown in a study done by Li et al. (2018) about the release time inconsistency. Indeed, they found that in 48% of app pairs in AndroZoo, the repackaged apps release time is anterior to that of the original apps.

Another threat to validity concerns our findings related to the API calls added and deleted during the repackaging process. As explained in RQ2, when the name changing obfuscation is present in a repackaged app, the API called identified as deleted or added may be the same. Nonetheless, we may detect these APIs as added\deleted because the obfuscated files have a low similarity with the original app. This fact does introduce a small possibility of error. In our dataset, this scenario is actually rather uncommon since less than 4% of the repackaged apps exhibit obfuscation of a type "name changing" that would introduce this error.

A final threat to validity concerns the comparison between the apps in our dataset and other apps in the stores that are not repackaged (see finding F7, F8, and F9). We obtained information about these apps from Kaggle dataset and from the study performed by Dong et al. (2018b). The threat to validity is that we cannot be sure if those apps are repackaged version of any other app or not. We can only say that we are comparing the repackaged app with the rest of the apps in the world. This threat would exist even if we downloaded apps randomly from stores as well.

7 Discussion and Conclusion

Summary. We have performed an empirical study comparing repackaged apps and their original versions. We used the dataset available on AndroZoo (Li et al. 2017a) containing 2,776 original apps and 15,296 repackaged app. Each original app is associated with a multiplicity of repackaged apps. In total, our dataset contained 15,296 pairs of original and repackaged apps. Our study sought to answer five research questions. The first question is: what are the main motivations for repackaging. We found that statistically, the main motivation for repacking is the distribution of adware. The second question examines how the apps' code is manipulated during the repackaging process. We found out that in repackaged apps, adware use API calls that are similar to those that occur in legitimate apps containing advertisement. In the third research question, we examined which types of apps are most frequently targeted for repackaging. Our finding showed that apart from the popularity of apps, the absence of obfuscation is a significant factor in determining if an app will be repackaged. In the fourth question, we asked why users download the repackaged apps even though the original one is freely available. We first showed that repackaged apps often have a high popularity and are located in trusted app stores. Then, an analysis of the name of apps suggested that users might erroneously believe that they are downloading a newer release of the original app or the local version of it in a foreign language. The last question studied the attributes of repackaged apps such as apps' components and permissions. Our findings show that in repackaged apps, these attributes remain similar to those of the original apps, suggesting that those attributes cannot be alone serve as the identifying features for detecting malware. However, they can form the basis of an indexing system used to find similar (and possibly repackaged) apps. Note that the detection of repackaged apps is different from malware detection. In the first case the object is to look for similarities between apps while in the latter case, the goal is to detect malicious behaviors.

Suggestion to protect apps against repackaging. Based on the findings in RQ3, most of the apps that were repackaged do not exhibit obfuscation. This suggests that malware developers need to understand the code of the app that they manipulate. Therefore, to protect apps against being repackaged we suggest applying obfuscation the app, which can hinder comprehension of the code.

Suggestion for malware detection. As discussed in RQ2, the frequency of API calls does not differ much between original apps and repackaged apps containing adware. Therefore, a detection mechanism for adware based exclusively on API calls is unlikely to be successful. Since, as it is shown in RQ1, a large proportion of repackaged app contains adware, as opposed to other types of malware, we suggest that all detection approaches investigate the accuracy of their approach using a dataset that contains apps repackaged with adware as well as the corresponding original versions of the same app.

Validity of common assumptions. Previous studies, e.g. (Li et al. 2017b), have shown that more permissions are requested in repackaged apps in comparison to the original apps. However, our study shows that 93.34% of repackaged apps have similar permission as their original counterpart. Therefore, using permission as a feature for detecting malware is not practical. Li et al. (2017b) mentioned that they found that repackaged apps sometimes requested the same permission multiple times, when it was already requested by the original app. In this paper, we did not distinguish between individual and duplicate requests for permissions.

In investigating RQ4, we found that a considerable number of repackaged apps do not have a name that is very similar to the name of the original app. For example, an app named “Pregnancy Exercise” was repackaged with names such as “Dance Waltz” and “Cha Cha Cha”. We also found out that many repackaged apps are republished in a different language than the original app. Some repackaged app detection methods, such as the ones proposed by Sun at al. (2015) or Soh et al. (2015), are premised on the fact that malware developers endeavor to maintain the “feel” and “see” of the original app, to better dupe users into downloading their repackaged apps. We have found this assumption to be erroneous in a considerable number of apps. If there are similarities in the appearance of apps, it is simply because the malware developers do not put much effort alter it. In other word, keeping the same “feel” and “see” is not necessary. However, further study is needed to determine if the apps’ resources, such as images, are kept similar in the repackaged apps. Still, based on our findings, detection approaches which are based on this fact should be adopted only with caution. Overall, we can also say that attackers seem prefer popular apps.

Future work. Studying the resource manipulations performed in repackaged app could yield a promising strategy to detect repackaged apps. Furthermore, since one of the motivation of users to download repackaged apps seem to be to obtain a legitimate apps in a language in which it is unavailable, the choice of the language used in repackaged app’s user interface is important aspect of future investigation.

The introduction of adware usually leads to the insertion of new code in the repackaged version, as well as to the deletion of other segments of the original code. Based on our findings in RQ2, the APIs used in the added parts are very similar to the one present in deleted code; often serve to connect to a server. This suggests that any approach that can make it difficult for adware developers to identify those parts of code such as code obfuscation techniques could be a promising defense mechanism against repackaging.

Finally, in this paper, we provide a characterization of the differences between repackaged and benign apps. The next step is to dig deeper and examine other properties such as code quality metrics, the use of libraries, process metrics (developer’s experience, etc.) to further explore the problem by answering the question of why the repackaged payloads are introduced.

References

Aafer Y, Du W, Yin H (2013) DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: International conference on security and privacy in communication systems. Springer, Cham, pp 86–103

AdMob and AdSense policies. https://support.google.com/admob/answer/6128543?hl=en. Accessed 25 Feb 2019a

Alazab M, Venkataraman S, Watters P (2010) Towards Understanding Malware Behaviour by the Extraction of API Calls. In: 2010 Second Cybercrime and Trustworthy Computing Workshop. Ieee, pp 52–59

Aldini A, Martinelli F, Saracino A, Sgandurra D (2015) Detection of repackaged mobile applications through a collaborative approach. Wiley Concurr Comput Pract Exp 27(11):2818–2838 . doi: 10.1002/cpe.3447

Android Developer Documentation (2018). https://developer.android.com/reference/dalvik/system/package-summary Accessed 3 Mar 2018

Arp D, Spreitzenbarth M, Malte H, Gascon H, Rieck K (2014) DREBIN : Effective and Explainable Detection of Android Malware in Your Pocket. In: NDSS (Vol. 14). pp 23–26

Bartel A, Klein J, Monperrus M, Traon Y Le (2012) Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In: ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. ACM, pp 27–38

Backes M, Bugiel S, Derr E (2016) Reliable Third-Party Library Detection in Android and its Security Applications. In: the 2016 ACM SIGSAC Conference on Computer and 2Communications Security. ACM, pp 356–367

Book T, Pridgen A, Wallach DS (2013) Longitudinal Analysis of Android Ad Library Permissions. IEEE Mob Secur Technol, ArXiv:1303.0857

Canfora G, Mercaldo F, Visaggio CA (2013) A classifier of Malicious Android Applications. In: Eighth International Conference on Availability, Reliability and Security (ARES). IEEE, pp 607–614

Chawla N V., Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: Synthetic minority over-sampling technique. J Artif Intell Res 16:321–357 . doi: 10.1613/jair.953

Chen J, Alalfi MH, Dean TR, Zou Y (2015a) Detecting Android Malware Using Clone Detection. J Comput Sci Technol 30(5):942–956 . doi: 10.1007/s11390-015-1573-7

Chen K, Liu P, Zhang Y (2014) Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: 36th International Conference on Software Engineering - ICSE 2014. pp 175–186

Chen K, Wang P, Lee Y, Wang X, Zhang N, Huang H, Zou W, Liu P (2015b) Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In: 24th USENIX Security Symposium (USENIX Security 15). pp 659–674

Chien E (2005) Techniques of Adware and Spyware. In: the Proceedings of the Fifteenth Virus Bulletin Conference (Vol. 47). Dublin Ireland

Crussell J, Gibler C, Chen H (2015) AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. IEEE Trans Mob Comput 14(10):2007–2019 . doi: 10.1109/TMC.2014.2381212

Crussell J, Gibler C, Chen H (2012) Attack of the clones: Detecting cloned applications on Android markets. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 37–54

Crussell J, Stevens R, Chen H (2014) MadFraud : Investigating Ad Fraud in Android Applications. In: the 12th annual international conference on Mobile systems, applications, and services. ACM, pp 123–134

Desnos A (2015) Androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !). https://github.com/androguard/androguard. Accessed 19 Jul 2018

Dong F, Wang H, Li L, Guo Y, Bissyande TF, Liu T, Xu G, Klein J (2018a) FraudDroid : Automated Ad Fraud Detection for Android Apps. ArXiv: 1709.01213v4

Dong S, Li M, Diao W, Liu X, Liu J, Li Z, Xu F, Chen K, Wang X, Zhang K (2018b) Understanding Android Obfuscation Techniques : A Large-Scale Investigation in the Wild. ArXiv: 801.01633v1

Enck W, Cox LP, Gilbert P, Mcdaniel P (2014) TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Trans Comput Syst 32(2):5

Erturk E (2012) A Case Study in Open Source Software Security and Privacy : Android Adware. In: In 2012 World Congress on Internet Security (WorldCIS). IEEE, pp 189–191

Gao J, Li L, Tegawend PK (2019) Should You Consider Adware as Malware in Your Study ? In: 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp 604–608

Gascon H, Yamaguchi F, Rieck K, Arp D (2013) Structural Detection of Android Malware using Embedded Call Graphs Categories and Subject Descriptors. In: ACM workshop on Artificial intelligence and Security. ACM, pp 45–54

Gonzalez H, Kadir AA, Stakhanova N, Alzahrani AJ, Ghorbani AA (2014) Exploring Reverse Engineering Symptoms in Android apps. In: The Eighth European Workshop on System Security. ACM, p 7

Google Inc. (2012) Cloud to Device Messaging (Deprecated). https://developers.google.com/android/c2dm/. Accessed 23 Jul 2018

Grace M, Zhou Y, Zhang Q, Zou S, Jiang X (2012) RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In: 10th International Conference on Mobile Systems, Applications, and Services. pp 281–294

Guan Q, Huang H, Luo W, Zhu S (2016) Semantics-based repackaging detection for mobile apps. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 89–105

Gupta S. (2013), Types of Malware and its Analysis, International Journal of Scientific and Engineering Research 4(1)

Hanna S, Huang L, Wu E, Li S, Chen C, Song D (2013) Juxtapp: A scalable system for detecting code reuse among android applications. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) 7591 LNCS:62–81

Huang A (2008) Similarity Measures for Text Document Clustering. In: the sixth new zealand computer science research student conference. pp 49‑56.

Hu W, Tao J, Ma X, Zhou W, Zhao S, Han T (2014) MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph. In: Proceedings - International Conference on Computer Communications and Networks, ICCCN. pp 1–7

Hurier M, Suarez-Tangil G, Dash SK, Bissyande TF, Le Traon Y, Klein J, Cavallaro L (2017) Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware. In: IEEE International Working Conference on Mining Software Repositories. pp 425–435

Islam R, Altas I (2012) A Comparative Study of Malware Family Classification. pp 488–496

Jiao S, Cheng Y, Ying L, Su P, Feng D (2015) A rapid and scalable method for android application repackaging detection. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 349–364

Khanmohammadi K, Hamou-Lhadj A (2017) HyDroid: A Hybrid Approach for Generating API Call Traces from Obfuscated Android Applications for Mobile Security. In: the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, p. 168-175

Khanmohammadi K, Rejali M, Hamou-Lhadj A (2015) Understanding the Service Life Cycle of Android Apps: An Exploratory Study. In: the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Denver, US

Kornblum J (2006) Identifying almost identical files using context triggered piecewise hashing. In: Digital Investigation. pp 91–97

Kumar M (2017) Beware! New Android Malware Infected 2 Million Google Play Store Users. https://thehackernews.com/2017/04/android-malware-playstore.html. Accessed 19 Jul 2018

Kywe SM, Li Y, Deng RH, Hong J (2014) Detecting camouflaged applications on mobile application markets. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 241–254

Lee YK, Lim JD, Jeon YS, Kim JN (2014) Protection method from APP repackaging attack on mobile device with separated domain. In: International Conference on ICT Convergence. pp 667–668

Leka O (2016) Database of Android Apps | Kaggle. https://www.kaggle.com/orgesleka/android-apps/data. Accessed 19 Jul 2018.

Li Y, Sundaramurthy SC, Bardas AG, et al (2015) Experimental Study of Fuzzy Hashing in Malware Clustering Analysis. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 15).

Li L, Bissyandé TF, Klein J (2018) MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In: The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018). IEEE

Li L, Gao J, Hurier M, Kong P, Bissyandé TF, Bartel A, Klein J, Traon Y Le (2017a) AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. doi: 10.1145/2901739.2903508

Li L, Li D, Bissyande TF, Klein J, Le Traon Y, Lo D, Cavallaro L (2017b) Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. In: IEEE Transactions on Information Forensics and Security. IEEE, pp 359–361

Li Li, Tegawendé Bissyandé, Jacques Klein(2018) Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark, arXiv preprint arXiv:1811.08520

Lin YD, Lai YC, Chen CH, Tsai HC (2013) Identifying android malicious repackaged applications by thread-grained system call sequences. Comput Secur 39(PART B):340–350 . doi: 10.1016/j.cose.2013.08.010

Linares-Vásquez M, Holtzhauer A, Bernal-Cárdenas C, Poshyvanyk D (2014) Revisiting Android reuse studies in the context of code obfuscation and library usages. In: 11th Working Conference on Mining Software Repositories - MSR 2014. pp 242–251

Liu B, California S, Nath S, Nsdi I (2014) DECAF : Detecting and Characterizing Ad Fraud in Mobile Apps This paper is included in the Proceedings of the. In: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). pp 57–70

Luo L, Fu Y, Wu D, Zhu S, Liu P (2016) Repackage-proofing Android Apps. In: In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp 550–561

Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G (2017) MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In: 24th Network and Distributed System Security Symposium

Ma Z, Wang H, Guo Y, Chen X (2016) LibRadar : Fast and Accurate Detection of Third-party Libraries in Android Apps. In: the 38th international conference on software engineering companion. ACM, pp 653–656

Maly F, Kriz P (2015) An Ad Hoc mobile cloud and its dynamic loading of modules into a mobile device running Google android. In: New Trends in Intelligent Information and Database Systems. Springer, Cham, pp 191–198

Microsoft Advertising. https://advertising.microsoft.com/home. Accessed 25 Feb 2019b

Mojica IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2014) A Large Scale Empirical Study on Software Reuse in Mobile Apps. Software, IEEE 31(2):78–86 . doi: 10.1109/MS.2013.142Mojica IJ, Nagappan M, Adams B, Hassan AE (2012)

Understanding Reuse in the Android Market. In: Program Comprehension (ICPC), 2012 IEEE 20th International Conference on. pp 113–122

Mulliner C, Robertson W, Kirda E (2014) VirtualSwindle : An Automated Attack Against In-App Billing on Android. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM, pp 459–470

OWASP (2016) Mobile Top 10 2016-Top 10 - OWASP. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10. Accessed 19 Jul 2018

Potharaju R, Newell A, Nita-rotaru C, Zhang X (2012) Plagiarizing Smartphone Applications : Attack Strategies and Defense Techniques. In: In International Symposium on Engineering Secure Software and Systems. Springer, Berlin, Heidelberg, pp 106–120Ren C, Chen K, Liu P (2014)

Droidmarking: Resilient SoftwareWatermarking for Impeding Android Application Repackaging. In: 29th ACM/IEEE international conference on Automated software engineering. pp 635–646

Rastogi V, Shao R, Chen Y, et al (2016) Are these Ads Safe : Detecting Hidden Attacks through the Mobile App-Web Interfaces. In: NDSS

Sahs J, Khan L (2012) A Machine Learning Approach to Android Malware Detection. In: European Intelligence and Security Informatics Conference. pp 141–147

Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Bringas PG (2012) On the automatic categorisation of android applications. In: IEEE Consumer Communications and Networking Conference, CCNC'2012. pp 149–153

Shahriar H, Clincy V (2014) Detection of repackaged Android Malware. In: 9th International Conference for Internet Technology and Secured Transactions, ICITST 2014. pp 349–354

Shao Y, Luo X, Qian C, Zhu P, Zhang L (2014) Towards a scalable resource-driven approach for detecting repackaged Android applications. In: ACSAC '14 (30th Annual Computer Security Applications Conference). pp 56–65

Sharif M, Lanzi A, Giffin J, Lee W (2008) Impeding Malware Analysis Using Conditional Code Obfuscation. In: Network and Distributed System Security Symposium, NDSS 2008

Singhal A (2001) Modern Information Retrieval: A Brief Overview. IEEE Comput Soc Tech Comm Data Eng. doi: 10.1.1.117.7676

Soh C, Tan HBK, Arnatovich YL, Wang L (2015) Detecting Clones in Android Applications through Analyzing User Interfaces. In: 23rd IEEE International Conference on Program Comprehension. IEEE, pp 163–173

Statista (2018) Rating of apps on Google Play as of May 2018. https://www.statista.com/statistics/266217/customer-ratings-of-android-applications. Accessed 26 Jul 2018

Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Blasco J (2014) Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Syst Appl 41(4):1104–1117 . doi: 10.1016/j.eswa.2013.07.106

Sun M, Li M, Lui JCS (2015) DroidEagle: seamless detection of visually similar Android apps. In: 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. ACM, p 9

Sun X, Zhongyang Y, Xin Z, Mao B, Xie L (2014) Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. In: International Information Security and Privacy Conference. pp 142–155

Symantec (2014) Android.Appenda. https://www.symantec.com/security-center/writeup/2012-062812-0516-99. Accessed 4 Mar 2019

Takabi H, Joshi JBD, Ahn GJ (2010) SecureCloud: Towards a comprehensive security framework for cloud computing environments. Proc - Int Comput Softw Appl Conf :393–398 . doi: 10.1109/COMPSACW.2010.74

Tian K, Yao D, Ryder BG, Tan G (2016) Analysis of Code Heterogeneity for High-Precision Classification of Repackaged Malware. In: IEEE Symposium on Security and Privacy Workshops, SPW 2016. pp 262–271

Viennot N, Garcia E, Nieh J (2014) A measurement study of google play. Meas Model Comput Syst - SIGMETRICS 42(1):221–233 . doi: 10.1145/2591971.2592003

Vigna G, Kruegel C, Bianchi A, Poeplau S, Fratantonio Y (2014) Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: NDSS (Vol. 14). pp 23–49

VirusTotal (2018) Free Online Virus Malware and URL Scanner. In: Google Inc. https://www.virustotal.com/#/home/upload. Accessed 19 Jul 2018

Au YWK, Zhou YF, Huang Z, Lie D (2012) PScout : Analyzing the Android Permission Specification. In: CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security. pp 217–228

Wang H, Guo Y, Ma Z, Chen X (2015) WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: International Symposium on Software Testing and Analysis - ISSTA 2015. pp 71–82

Winter C, Schneider M, Yannikos Y (2013) F2S2: Fast forensic similarity search through indexing piecewise hash signatures. Digit Investig 10(4):361–371 . doi: 10.1016/j.diin.2013.08.003

Wiśniewski R (2012) Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. Accessed 19 Jul 2018

Wu DJ, Mao CH, Wei TE, Lee HM, Wu KP (2012) DroidMat: Android malware detection through manifest and API calls tracing. In: Proceedings of the 2012 7th Asia Joint Conference on Information Security, AsiaJCIS 2012. pp 62–69

Wu X, Zhang D, Su X, Li W (2015) Detect repackaged Android application based on HTTP traffic similarity. Secur Commun

Networks 8(13):2257–2266 . doi: 10.1002/sec.1170

Xue Y, Meng G, Liu Y, Tan TH, Chen H, Sun J, Zhang J (2017) Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. IEEE Trans Inf Forensics Secur 12(7):1529–1544 . doi: 10.1109/TIFS.2017.2661723

Xu K, Li Y, Deng RH (2016) ICCDetector: ICC-Based Malware Detection on Android. IEEE Trans Inf Forensics Secur 11(6):1252–1264 . doi: 10.1109/TIFS.2016.2523912

Yang C, Xu Z, Gu G, Yegneswaran V, Porras P (2014) DroidMiner : Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android. In: European Symposium on Research in Computer Security. pp 163–182

Yue S, Feng W, Jiang Y, Tao X, Xu C, Lu J (2017) RepDroid: an automated tool for Android application repackaging detection. In: In (ICPC), 2017 IEEE/ACM 25th International Conference on Program Comprehension. IEEE, pp 132–142

Zeng Q, Luo L, Qian Z, Du X, Li Z (2018) Resilient Decentralized Android Application Repackaging Detection Using Logic Bombs. In: In Proceedings of the 2018 International Symposium on Code Generation and Optimization. ACM, pp 50–61

Zhang F, Huang H, Zhu S, Wu D, Liu P (2014) ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. WiSec 2014 - Proc 7th ACM Conf Secur Priv Wirel Mob Networks :25–36 . doi: 10.1145/2627393.2627395

Zhang L, Niu Y, Wu X, Wang Z, Xue Y, Science C, College T (2013) A3 : Automatic Analysis of Android Malware. In: 1st International Workshop on Cloud Computing and Information Security. Atlantis Press. pp 89–93

Zhauniarovich Y, Gadyatskaya O, Crispo B, La Spina F, Moser E (2014) FSquaDRA: Fast detection of repackaged applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 130–145

Zhao Y, Qian Q (2018) Android Malware Identification Through Visual Exploration of Disassembly Files. Int J Netw Secur 20(6):1005–1015 . doi: 10.6633/IJNS.201811

Zhou W, Wang Z, Zhou Y, Jiang X (2014) DIVILAR: Diversifying Intermediate Language for Anti-repackaging on Android Platform. In: CODASPY ’14 (4rd ACM conference on Data and Application Security and Privac). pp 199–210

Zhou W, Zhang X, Jiang X (2013a) AppInk : Watermarking Android Apps for Repackaging Deterrence. In: The 8th ACM SIGSAC symposium on Information, computer and communications security. pp 1–12

Zhou W, Zhou Y, Grace M, Jiang X, Zou S (2013b) Fast , Scalable Detection of “ Piggybacked ” Mobile Applications. In: In Proceedings of the third ACM conference on Data and application security and privacy. ACM, pp 185–196

Zhou W, Zhou Y, Jiang X, Ning P, Drive O (2012a) Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In: In Proceedings of the second ACM conference on Data and Application Security and Privacy. ACM, pp 317–326

Zhou Y, Jiang X (2012) Dissecting Android Malware: Characterization and Evolution. In: 2012 IEEE Symposium on Security and Privacy. Ieee, pp 95–109

Zhou Y, Wang Z, Zhou W, Jiang X (2012b) Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. 19th Annu Netw Distrib Syst Secur Symp 25(4):50–52