A Rapid and Scalable Method for Android Application Repackaging Detection

Sibei Jiao, Yao Cheng, Lingyun Ying, Purui Su, and Dengguo Feng

Trusted Computing and Information Assurance Laboratory
Institute of Software, Chinese Academy of Sciences
{jiaosibei,chengyao,yly,supurui,feng}@tca.iscas.ac.cn

Abstract. Nowadays the security issues of Android applications (apps) are more and more serious. One of the main security threats come from repackaged apps. There already are some researches detecting repackaged apps using similarity measurement. However, so far, all the existing techniques for repackaging detection are based on code similarity or feature (e.g., permission set) similarity evaluation. In this paper, we propose a novel approach called ImageStruct that applies image similarity technique to locate and detect the changes coming from repackaging effectively. ImageStruct performs a quick repackaging detection by considering the similarity of images in target apps. The intuition behind our approach is that the repackaged apps still need to maintain the "look and feel" of the original apps by including the original images, even they might have their additional code included or some of the original code removed. To prove the effectiveness and evaluate the reliability of our approach, we carry out the compare experiments between ImageStruct and the code based similarity scores of AndroGuard. The results demonstrate that ImageStruct is not only with good performance and scalability, but also able to resistant to code obfuscation.

Keywords: Android Malware, Repackaged Application Detection, Image Similarity

1 Introduction

Android is privileging in recent years. Data shows that Android has shared 85% of the global smartphone market in Q2 2014 [1]. Normally, users purchase or download apps to their mobile phones from centralized application markets. For Android phones, Google hosts the official Android market Google Play, while all the other markets that provide application downloading is called third party markets, such as Amazon Appstore for Android. Android apps are emerging nowadays. There have been more than 1,400,000 apps in Google Play only [3]. From the huge amount of apps, there come the security threats, despite Android's security system based on permission and sandbox, and Google Play's bouncer mechanism [16]. As a matter of fact, the Android ecosystem suffers from malicious apps seriously, among which repackaging is one of the major security threats [13].

© Springer International Publishing Switzerland 2015
J. Lopez and Y. Wu (Eds.): ISPEC 2015, LNCS 9065, pp. 349–364, 2015.
DOI: 10.1007/978-3-319-17533-1_24

There are mainly two motivations for app repackaging. Firstly, dishonest developers embed the advertisements related to their own benefit accounts to other developers' apps and repackage the apps using their own signatures. After republishing the repackaged apps to the Android market, the apps are able to earn monetary profit when the embedded advertisements displayed or clicked on users' phones. Secondly, malware writers modify popular apps by inserting some malicious payload into the originals. The malicious payload is to take over mobile devices, steal user's private information, send premium SMS messages stealthily, or purchase apps without user's awareness, which all lead to the user privacy leakage or financial loss. Malware writers leverage the popularity of the original apps to increase the propagation of the repackaged malicious ones. The repackaging not only violates the copyright of original app developers, but also brings disorder factors along with severe threats to the Android markets. It has been found that about 5% to 13% of apps in the third party Android markets are the plagiarism of apps in the official Android market with no consideration of code obfuscation [22]. Besides, according to a recent study[13], it shows that 1083 of the analyzed 1260 malware samples (86.0%) are repackaged versions of legitimate apps with malicious payloads, indicating that repackaging is a favorable channel for mobile malware propagation. Moreover, since users can choose to download apps from both official market and third party markets that from different countries (e.g., Anzhi, one of the biggest Android market in China), the repackaging problem appears in both inter- and intra- market, which increases the scale and challenge for repackaging detection.

Currently, the problem of app repackaging is widely explored and several solutions have been proposed to identify plagiarized apps [5][19][8][15]. Most of these solutions are based on features extracted from the apps' code which is often impacted by the repackaging process, as the added functionality, i.e. new advertisement libraries and/or malware code, requires modification of apps' code. Of great importance, code obfuscation techniques are widely adopted to evade detection. However, the existing systems comparing similarity based on bytecode or statistical patterns of control-flow or API are *not* able to deal with the obfuscated repackaged apps [10]. In this paper, we focus on dealing with the code obfuscation problem in repackaging detection.

The intuition behind our work is that when an illegitimate author repackages an app to include malicous functions or create a pirated copy, the repackaged apps still need to maintain the "look and feel" of the original one by including the original images and other resource files, even though they might have additional code included or original code removed. Based on above observations, we explore similarity analysis on the apps' resource files especially image files to detect repackaged apps.

We propose ImageStruct which takes advantage of our novel approach and can be served as a complementary to existing repackaging systems. ImageStruct measures the similarity between two apps, based on which repackaged apps can be further detected. Specifically, giving each app from a third party Android marketplace, we measure its similarity with those apps from the official Google

Play. In order to handle the large number of apps in the marketplaces, we extract distinguishing features from apps and generate app-specific fingerprints which are based on apps' images to locate and detect the similar apps. When the similarity of two apps with differen signatures exceeds certain threshold, one of the app is considered repackaged. We construct large scale real-world experiments and run ImageStruct on apps from four third-party Android marketplaces (two from China and the other two from America) and one malware set against 39,492 apps from the official Google Play. In summary, we make four main contributions in this paper:

- We propose a novel technique to detect repackaged Android applications based on apps' image resources, which is able to resilient to code obfuscation.
- A fast algorithm to detect repackaged apps is proposed based on image fingerprint generation. It manages to compare on average 1 app every 10 second on our dataset using a commodity hardware. The data shows that our approach runs faster than existing system, e.g. DNADroid, AndroGuard.
- We implement ImageStruct that carries the image similarity measurement and evaluate the practicality of our approach. By comparing the resource-based similarity score produced by ImageStruct with the code-based similarity score computed by the open-source AndroGuard [2], the results show that the ImageStruct similarity score is strongly correlated with the AndroGuard code similarity score.
- We evaluate the effectiveness of the ImageStruct on a dataset with more than 48,000 apps crawled from Google play and 4 alternative markets. The experiments shows that repackaging rates of alternative markets ranging from 6.7% to 14.5%, which is relatively high for a healthy mobile ecosystem.

In the following section, we describe the background and related work. In section 3, we detail our methodology and its implementation. Then in section 4, we present empirical results from two experiments. In section 5, we discuss the problem of false positives and evasion. The last section includes our conclusion and future work.

2 Background and Related Work

In this section, we introduce the structure of Android application and the existing work for repackaging detection.

2.1 Background

Android apps on the devices exist in the form of Android packages (apk files). The package contains executable code, manifest file, libraries and resource files, all compressed in a zip archive with signature from developer (Fig.1 is an unzipped FruitNinja apk [18]). Nowadays, smartphones have powerful processors, advanced video and audio systems, meaning that smartphones are able to support screens with high resolutions and to produce sounds of good quality.

```
Archive:  com.halfbrick.fruitninjafree_19199200_0.apk
  Length      Date    Time    Name
---------  ---------- -----   ----
   187151  08-20-2012 10:25   META-INF/MANIFEST.MF
   187272  08-20-2012 10:25   META-INF/IDREAMSK.SF
      949  08-20-2012 10:25   META-INF/IDREAMSK.RSA

     7975  04-19-2012 13:21   assets/sound/bonus-banana-x2.ogg
     4035  04-19-2012 13:21   assets/sound/bonus-count-up.ogg
    16062  04-19-2012 13:21   assets/sound/bonus-drum-roll.ogg
     5976  04-19-2012 13:21   assets/sound/bonus-explosion-1.ogg
     6032  04-19-2012 13:21   assets/sound/bonus-explosion-3.ogg
     6253  04-19-2012 13:21   assets/sound/bonus-explosion-5.ogg

   194807  08-15-2012 10:15   res/drawable/wood2.png
     3578  08-15-2012 10:15   res/drawable/yourall.png
     3422  08-15-2012 10:15   res/drawable/yourfriend.png
     2592  08-15-2012 10:15   res/layout/achivementdesc.xml
     1168  08-15-2012 10:15   res/layout/admob.xml
```

Fig. 1. Files inside FruitNinja

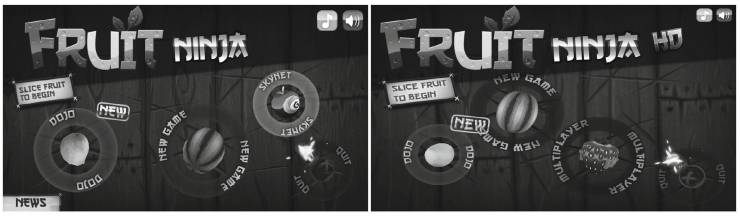

Fig. 2. User interfaces of official (left) and repackaged (right) FruitNinja

The apps are not only the composition of code any more. The multimedia factors are essential for apps to fully utilizing the power of smartphones and attracting users. These resource files become an inseparable part of modern mobile apps. The large portion of resource files in Android packages is also confirmed by our dataset that consists of 49,000 apps. The rescouses are delivered to the device packaged together with logical code. To deceive or mislead users, the repackaged apps usually maintain the original functionality and appearance. Fig.2 shows the user interfaces of an official FruitNinja and a repackaged FruitNinja which are hard to distinguish. The observation that apps with different functionality typically have different images, and similar apps (e.g. different versions of the same app) have similar images is the basis our approach relies on.

2.2 Related Work

There have been much work related to repackaging apps. The analysis performed in [22] shows that 5-13% apps hosted in alternative markets are repackeged. Jiang et. al. [22] search repackaged applications in third-party markets using Google Play as baseline. Another tool called DroidMOSS uses fuzzy hashing of code to calculate fingerprint of the app and then computes the edit distance between two fingerprints to score the similarity. Grace et. al. [21] further investigate the problem of repackaged apps and concentrate on detection of piggybacked apps (repackaged apps that carry a malicious payload). They decouple code into primary and non-primary modules, and then generate fingerprint for each primary module, which contains the main functionality. While iterating over the

fingerprints, the linearithmic algorithm detects similar primary modules that are considered as piggybacked candidates. Finally, piggybacked apps are detected by comparing the sets of non-primary modules of these similar apps. The experiments show the presence of 1.3% piggybacked apps in the dataset. DNADroid [5] detects cloned (plagiarized) apps using semantic similarity. It extracts PFG (Program Flow Graph) of each method in target apps, based on which the subgraph isomorphism problem serves as final criteria. DNADroid managed to detect 191 cloned pairs (0% false positives was reported). The authors also compared their tool with AndroGuard [2]. In the total of 191 pairs, AndroGuard failed for 24 pairs and with very low similarity score for 10 pairs, meaning that it missed 18% of the pairs found by DNADroid. Based on the work on DNADroid, Crussell et al. developed AnDarwin which extracts features from app code and compares in non-pairwise way, allowing performing large-scale analysis apps. On a dataset of 265,359 third-party apps collected from 17 markets DNADroid detected 4,295 cloned and 36,106 rebranded apps. Juxtapp [8] presents another approach to detect code reuse among Android apps. To discover the similarity between the code, they use k-grams of Dalvik opcode sequences as features. To obtain app representation, they apply hashing to the extracted features. The Juxtapp can detect (a) buggy and vulnerable code reuse, (b) known malware instances and (c) pirated apps. To assess the Juxtapp efficiency, the authors ran the experiment of pairwise comparison on a set of 95,000 Android apps (an Amazon EC2 cluster with 25 slave nodes was used), which costed about 200 minutes. As for effectiveness, 174 and 239 samples containing vulnerable patterns were identified in the in-app billing code and the code using Licence Verification Library respectively. Moreover, they identified 34 new instances of known malware in the alternative Anzhi market. MIGDroid [9] is able to detect repackaged malwares based on method invocation graph. The method invocation graph reflects the "interaction" connections between different methods. MIGDroid constructs method invocation graph on the smali code level, which will be divided into weakly connected sub-graphs further. The thread score of each sub-graph is calculated based on the invoked sensitive APIs. The sub-graphs with higher scores will be more likely to be malicious. Experiment results based on 1,260 Android malware samples demonstrate the specialty of MIGDroid in detecting repackaged malwares.

Recently, a framework for evaluating Android app repackaging detection algorithms has been proposed [10]. The paper classifies currently available approaches for detection of repackaged apps and presents a framework that can assess the effectiveness of such algorithms. The framework translates Dalvik bytecode into Java code, applies obfuscation techniques and packs back the code back to the Dalvik. The effectiveness assessment is to run over real and converted apps modified by the framework. It assesses repackaging detection algorithms by broadness (i.e., how an algorithm can stand to obfuscation techniques applied separately) and by depth (i.e., if an algorithm is resilient to techniques applied sequentially). As the case study, the authors applied the framework to AndroGuard - the only publicly available tool for repackaging detection. The results

show that AndroGuard can successfully combat with different obfuscation techniques and, thus, can be widely used to detect repackaged apps. It is worth noticed that ImageStruct will successfully pass the tests as it does not rely on code similarity.

Much of the existing work on similarity analysis is focused on program code, which faces a number of domain specific considerations when performing code similarity analysis: the source code is not available for most apps so a preprocessing step is needed; each app includes common libraries that it uses, which leads to significant code copying that can undermine code similarity scores; Android developers often obfuscate code to make the reverse engineering more difficult at the same time misbehaving authors obfuscate code to evade detection. These problems motivate an alternative approach for determining the similarity between apps. Li et al. [15] proposed a system called DStruct , which detects similar apps based on their directory structures. The files inside the APPs are typically organized into directories. They construct a tree to represent the directory structure, and compute the edit distances among them. However, DStruct is very time consuming and Android app's directory structures can be easy obfuscate.

We propose ImageStruct which is a scalable and rapid tool to determine the repackaged apps based on the images in apps' archives. This approach avoids all the problems that are associated with the code and provides another metric for determining app similarity. Even though, we do not consider ImageStruct as a replacement for existing system, since the latter performs quite effectively and has the capability to detect vulnerable code reuse. Instead, we develop ImageStruct so that we could use it in conjunction with existing systems to eliminate false positives and negatives and to reaffirm results of each other.

3 Design

There are three design goals for repackaging detection: accuracy, rapid speed, and scalability. Accuracy is a natural requirement to effectively identify app-repackaging behavior in current marketplaces. However, challenges arise from the fact that the repackaging process might dramatically use code obfuscation technique in the repackaged app, which renders app bytecode similar schemes ineffective. Meanwhile, due to the large number of apps in various marketplaces, our approach needs to be rapid and scalable. Our current data set for app similarity measurement has 49,000 apps, which makes the expensive semantic-aware full app analysis not feasible. In our design, we use images in app packages for app fingerprint generation. The generated fingerprints need to be robust in order to accommodate possible changes from app-repackaging behavior.

In this paper, we aim to uncover repackaged apps and understand the overall repackaging situation in current marketplaces. We focus on images inside Android apps instead of bytecode. Observation shows repackaged app always has similar "look and feel" with original app. And at meantime, our dataset shows that all the apps contain images with no exception. There are two assumptions

our approach based on. First, we assume that the apps from the Google Play are more authentic and assumed to be trusted and not repackaged. In real-world situation, there may be exceptions when bouncer fails (which has happened before), but still ImageStruct is helpful to distinguish app pairs with repackaging relationship. Second, we assume that the signing keys of app developers are not leaked (which is the identical certificate that developers should have enough security awareness of) so that there is no possibility for a repackaged app shares the same signature with the original one.

3.1 Overview

As illustrated in previous sections, repackaged apps share two common characteristics. First, the one who clones the application, seeks to resemble the original one as much as possible to increase the installation probability of cloned one. Second, the original app and the repackaged app are signed by different developer keys. By leverages these two insights, ImageStruct extracts related features from apps and then discerns whether one app is repackaged from the other one.

Fig.3 illustrates the overview of our approach. In essence, ImageStruct has three key steps. The first step is to extract two kinds of features from each app, i.e., image fingerprints and author information. These two features are used to uniquely identify each app. After that, the second step is to store these features efficiently. As there are tens of millions of apps in the market, full pairwise comparison is not a good choice. We design a better way to store app features for similarity measurement. Finally, based on app features, the third step discerns the source of apps, i.e., either from the Google play or from the third-party marketplaces, and measures their similarity scores to detect repackaged apps. In the following subsections, we demonstrate each step in detail.

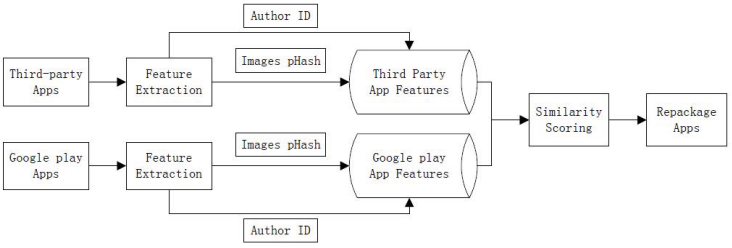

Fig. 3. The ImageStruct work flow

3.2 Feature Extraction

An Android app is essentially a compressed archive file including image files, META-INF subdirectory and other necessary files (executables, libraries and multimedia resource files). The images determine the app's appearance while the META-INF subdirectory contains the author information.

There are many image feature calculation methods, such as color histogram [20], SIFT [14], pHash [7] and etc. PHash using discrete cosine transform to remove the useless high frequency characteristics, and retain the low-frequency characteristics of the image. It is able to detect similarity within 25% changes to original image which often happens when repackage (e.g. image cropping and resolution adjusting). This characteristic of PHash distinctly differs from the other calculation methods. Based on efficiency and accuracy consideration, pHash is used as ImageStruct's feature extraction algorithms.

For the author information, the META-INF subdirectory includes the full developer certificate. The certificate provides information of developer name, contact, organization, as well as the public key fingerprints. In ImageStruct, we map each developer certificate into a unique 32-bit identifier (authorID). This unique identifier is then integrated into the feature for comparison.

3.3 Feature Storage

After feature extraction, we need to apply a appropriate way to store feature information. The easiest way is store in the form of {application, features}. It is a simple and intuitive way, which however increase the time consuming of similarity detection. Suppose there are n apps already in the original database, then there come m new apps for similarity detection, the time complexity is $n * m$. Nowadays, there are thousands of apps are newly published every day. Accordingly, the time complexity of similarity detection will grow rapidly. To deal with the problem, we utilize a new storage method in the form of {features, applications}. In practice, the number of images in one Android app is ranging from 10 to 10000, and the average number is around 2000. With the improvement of storage structure, the time complexity is $2000 * m$ which is of constant and much better to satisfy the purpose of rapid and scalable.

As shown in Fig.4, apps' feature is stored in two databases, redis[17] and MySQL[12]. Redis is an advanced key-value cache and store database. It is often referred to as data structure server, since keys can contain strings, lists and sets. The redis database stores app hash indexed by image fingerprint. Each fingerprint corresponds to a set of apps' hash values. MySQL is the world's most popular open source database. The MySQL database stores apps' information, including app name, signature and source indexed by app's hash value.

3.4 Similarity Scoring

After features extraction, we got two databases which store the necessary information to calculate app's similarity. The above first two steps are applied to each app regardless of its source. In the third step, we treat apps differently according to where it comes from, Google Play or alternative marketplaces. The similarity scores based on the derived fingerprints are calculated against the Google Play apps. Our image features is deterministic, as the same fingerprints will be generated if two images are identical. In addition, PHash, the algorithm we adopt, can also effectively locate the image changes possibly made in repackaged apps.

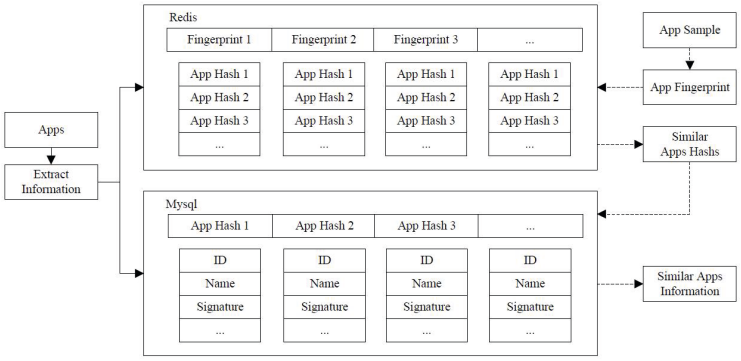

Fig. 4. Implementation framework

We also perform containment analysis on the fingerprint sets to determine the similarity percentage of images in both apps. Containment is defined as the percentage of features in app A that also exist within app B. This value $C(A|B)$ is computed by dividing the number of features existing in both apps by the number of images in A.

$$C(A|B) = \frac{|A \cap B|}{A}$$

The app similar formula is as follows.

$$similarityScore(app_1, app_2) = \frac{|app_1 images \cap app_2 images|}{app_1 images}$$

If the similarity score between two apps exceeds certain threshold and the two apps are signed with different developer keys, ImageStruct reports the one that is not from the Google Play as repackaged. The threshold is very important as it affects both false positives and false negatives of our system. Specifically, a high threshold would likely lead to low false positives but high false negatives, while a low threshold would introduce high false positives but low false negatives. In our experiments, we empirically find the threshold 0.6 is a good balance between the two metrics.

Once the similarity percentage computation is completed, we sort (e.g. quick sort) the results so that the pairs of more significant similarity appear first. More information of these apps can be retrieved from the MySQL database by apps' hash value.

4 Evaluation Experiments and Results Analysis

To conduct an overall evaluation of our approach, we implement ImageStruct on Linux system. Specifically, we use pHash [7] which is a image fingerprint generation tool to extract image feature in application package. For the apps' author

information, we adopt keytool[11] which is already a part of Android SDK. After the features are extracted, they are stored separately in redis and mysql database as we explained in previous sections. Based on these features, similarity scoring is carried out. The similarity scoring takes both image features and author information into account. Apps that share the same author information are excluded from repackaging detection process. Otherwise, ImageStruct calculates the edit distance and hence derives the similarity score. The larger the score is, the more similar the pair of apps are to each other.

Images are pervasive existence in Android apps. We analyze top 100 popular free apps of every category in Google Play. The data shows that the image number in each app ranges from 1 to 5000. The average value of image number is 422. This number gives ImageStruct confidence that using image as a universal feature can be applicable to almost all apps.

To demontrate the repackaging detection ability, we crawl apps from four popular third-party Android marketplaces, two of which are from US and two of which are from China. Our study is based on the apps collected in the November 2014. Meanwhile, we also collect nearly 40,000 apps from the Google Play as benchmark database between July 2014 and November 2014. Besides, we also analyse the malware set from Android Malware Genome Project [23]. The dataset details in Table 1.

Table 1. App number in experiment datasets from Google Play, third-party markets and Genome

App source	# of Apps
Android Drawer (US)	985
Freeware Lovers (US)	1,438
eoeMarket (CN)	3,347
Anzhi (CN)	3,009
Malware Genome	1,246
Google Play	39,492

4.1 Repackaged Applications in Alternative Marketplaces and Malware Set

To perform a concrete study on the repackaged apps in markets and measure the effectiveness of ImageStruct, we detect the apps from four third-party marketplaces and Android Malware Genome Project against Google Play dataset to find out whether there is any repackaged app. Specifically, for each app, ImageStruct scores the similarity score between the app and each of the 39,492 Google Play apps. In practice, we apply 0.6 as the threshold due to the balance of false positive and false negative. Thus, if the similar score of a pair of apps is higher than 0.6, we consider they are in a repackaging relationship.

The results are shown in Table 2. The second column indicates the number of repackaged apps in each market dataset detected by ImageStruct. We can see

that ImageStruct reports 6% to 14% of apps came from 4 alternative markets are repackaged, which is relatively high for a healthy app market and further can seriously affect the security of entire smartphone ecosystem. We manually verify 20 apps randomly out of the repackaged apps in each market reported by ImageStruct. For each marketplace, only one or two false positives are found, which validate the effectiveness of ImageStruct. As for the false negative cases, the main contributing factor is the completeness of original app set. If the original app is not in the benchmark dataset (Google Play dataset), the similarity could never be detected. ImageStruct can be improved as the accumulation of original data from Google Play.

Table 2. Repackaging detection results for 5 dataset

App source	Repackaged #	Repackaging %
Android Drawer (US)	122	12.4%
Freeware Lovers (US)	121	8.4%
eoeMarket (CN)	225	6.7%
Anzhi (CN)	438	14.5%
Malware Genome	178	14.3%

4.2 Comparison with AndroGuard

AndroGuard have the capability to analyze similarity of Dalvik code [6]. The difference between two apps, such as newly-added or deleted methods could be detected. AndroGuard performs it in 3 steps; a) generates signature for each Dalvik method in apps, b) identifies the identical methods in both apps, c) discovers all methods that are similar. The signature of Dalvik method is generated based on the control flow information, API calls and exceptions inside the method. Methods are considered identical if they have the same signature hashes. Normalized Compression Distance (NCD) [4] is used to measure the similarity between methods.

A random selection of 250 pairs with the same certificate and 250 pairs with different certificates are used to perform AndroGuard similarity metrics, which averages out the similarity score with no predominance of score range. This two groups are different by nature which is reinforced by our experiment data. Fig.5 presents a scatterplot of the similarity values from AndroGuard (Y-axis) and ImageStruct (X-axis). It shows that the difference between AndroGuard and ImageStruct is within a certain range highly correlated with each other, which confirms that ImageStruct can effectively detect repackaged applications. However, there are some points with high AndroGuard similarity scores, but with ImageStruct similarity scores varying. After manually inspecting these pairs, we manage to find the reason causing the problem. One of the most common observed cases is that the same logical code is used for displaying different content. For instance, we find several reading apps, which can load digital books. For every different book, a individual application is been developed. All these

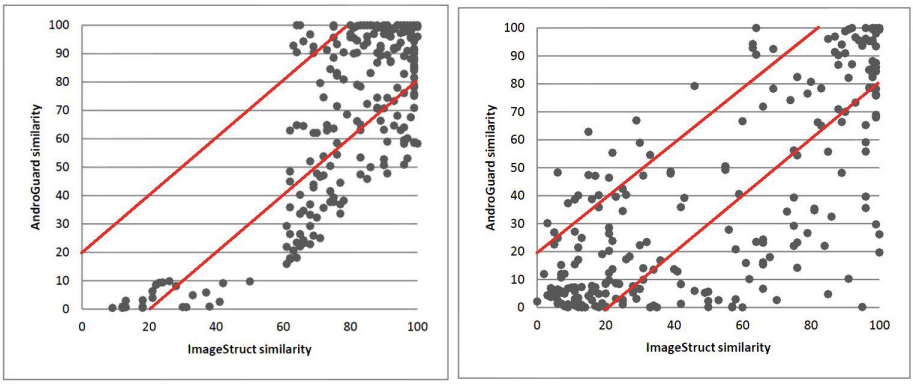

Fig. 5. Scatterplots of similarity scores from ImageStruct (X-axis) vs. AndroGuard (Y-axis) for app pairs signed with different certificates (left) and same certificates (right)

applications use the same code, but the images (e.g. the book cover or chapter cover) are different. We also inspect some pairs with low AndroGuard similarity scores and high ImageStruct scores. One of the case is that the same images are used in different apps, e.g. the apps developed by same author share many same images. Besides, for the random pairs expected to have low similarity.

ImageStruct does not deviate from AndroGuard a lot on average. We can see that for the app pairs not marked as similar by ImageStruct, AndroGuard does not output significant code similarity score either. 60% of the times, the difference between results of ImageStruct and AndroGuard is within 20% (within red line in Fig.5). It means that if developers include little images in apps similar with another app, they are also likely not to reuse the code, which is often the case for apps intensively produced by companies.

During the experiment we find that AndroGuard takes significantly more time than ImageStruct. The time of comparison depends on the complexity and similarity of target app pair. It takes much less time to compare more similar apps than totally different ones. Average time for one pair is approximately 85 seconds using AndroGuard. Hence, measurement of similarity metrics for the whole app corpus we have crawled is not feasible.

4.3 Performance Evaluation

ImageStruct runs on Ubuntu 12.04 with Intel Core CPU (4 cores) and 32GB of RAM, which is a regular or a little weaker hardware configuration for most of the repackaging detection systems. The time consuming ranges from 1 seconds to 2 minutes for ImageStruct to compare one app against all the other apps in the benchmark dataset. The time depends on the number of images included in the target app. The lower boundary of time occurs when target app has less than 10 images, and the upper boundary of time occurs when the app has over 2000 images. Fig.6 shows the feature extraction time cost changing with the number of images per-app. The time cost is proportional to the number of

images. ImageStruct takes about 1 minute to deal with 1409 images, which is efficient enough for large dataset and could be accelerated in distributed way.

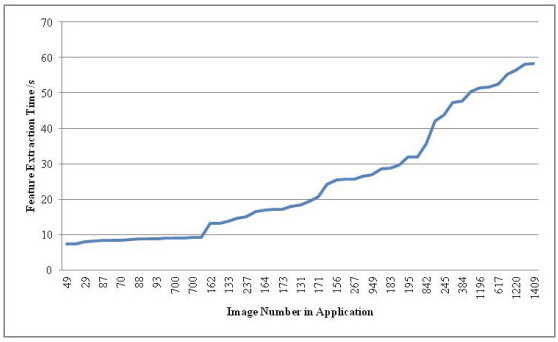

Fig. 6. Feature extraction time cost (Y-axis) is proportional to the number of images inside the Android app (X-axis)

We use parallel program to divide the tasks up to improve performance greatly when dealing with large dataset. Each processor works on the fingerprints extraction and distance calculation in parallel. Table 3 shows the time cost for ImageStruct running different datasets under 8 threads, which indicates the good scalability of ImageStruct.

Table 3. Performance of repackaging detection under 8 threads

Source	App#	Feature extraction(s)	Repackaging detection(s)
Android Drawer (US)	985	10,976	60
Freeware Lovers (US)	1,438	12,064	48
eoeMarket (CN)	3,347	34,233	113
Anzhi (CN)	3,009	29,431	101
Malware Genome	1,246	1,623	28
Total	10,025	88,327	350

4.4 Discussion

False positives can arise due to multiple facts as we use feature extracted from images. Different apps may appear similar if their image resources are similar. For example, the image similarity of different apps published by same company are higher than 80%. In fact, this case will not affect the repackaging judgement as the apps are with the same signature. False positives mainly result from the apps that differ in functionality but reuse images. Usually, developers download image materials from Internet and then use them in their own apps. This pattern leads to the situation that the images among different apps appear similar, which further results in ImageStruct's false positives.

Currently, the false positives and false negatives are both significant, as authors mainly focus on modifying code to add their components or advertisements instead of the resources of apps when repackaging. As there are already many techniques to detect code reuse, repackagers attempt to obfuscate the code to avoid detection as much as possible. On the contrary, they do not pay much attention to the resource files, another reason of which is they have to maintain the appearance of the original app. We test ImageStruct on different versions of one app and find high similarity between them. This means our approach could effectively detect repackaged apps that introduce the same level of modifications to the images as that occurs when evolving to different versions. Since our approach depends on the image similarity, if repackagers understand the detail mechanism of our approach, they may significantly alter images when repackaging as countermeasure, such as adding futile images or modifying the contents, which will lower the image similarity score reported by ImageStruct. But still, these apps will appear in the list of similarity apps with a low percentage similarity, as the repackaged apps still need some similarities with the original app, such as interfaces and icons. Besides, we could adopt other valid image similarity detection algorithm that are more complex and powerful, such as SIFT which considers approximate match to improve the performance.

5 Conclusion

In this paper, we propose an novel approach to detect Android application repackaging based on the image resources and implement ImageStruct which is capable of effective and fast repackage detection. ImageStruct scores the similarity for target apps and classifies them as similar if substantial number of images in packages are with same feature. We have evaluated the practicality of ImageStruct in two aspects to determine whether it shows correlated results with the code based repackaging detection techniques, and whether it is scalable and fast enough to handle significant large number of apps. Our results are encouraging. The ImageStruct similarity score is strongly correlated with the AndroGuard code-based similarity score, especially for the apps signed with different certificates, i.e. potentially plagiarized. ImageStruct also has good performance. It is able to extract features for a dataset of more than 39,000 apps in less than 48 hours, and only takes 24 hours to check 10,000 apps if they are repackaged or not. At the meantime, the approach can be easily parallelized using different parallelization algorithms to further enhance the performance.

The methodology in ImageStruct inspires new feature in application plagiarism detection algorithms not only for Android but also for other systems, such as iOS and Windows Phone. Useful patterns and meaningful findings can be inferred from the results output by ImageStruct to find out more repackaging characteristics and further improve the feature extraction and threshold configuration, which could be an interesting future work. Moreover, ImageStruct can be used to improve the on-market plagiarism detection algorithms by complementing the code similarity-based approaches to provide a healthy mobile ecosystem.

Acknowledgments. This work is supported by National Basic Research Program (Grant No.2012CB315804), National Natural Science Foundation of China (Grant No.91118006), and Beijing Natural Science Foundation (Grant No.4154089).

References

1. Strategy Analytics. Strategy analytics: 85% of phones shipped last quarter run android (2014), http://bgr.com/2014/07/31/android-vs-ios-vs-windows-phone-vs-blackberry/
2. anthony.desnos@gmail.com. Androguard: Reverse engineering, malware and goodware analysis of android applications (2013), https://code.google.com/p/androguard/
3. AppBrain. Number of android applications (2014), http://www.appbrain.com/stats/number-of-android-apps
4. Cilibrasi, R., Vitanyi, P.M.B.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
5. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33167-1_3
6. Desnos, A.: Android: Static analysis using similarity distance. In: 2012 45th Hawaii International Conference on System Science (HICSS), pp. 5394–5403 (January 2012)
7. Evan, K., David, S.: Phash (2014), http://www.phash.org/
8. Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: A scalable system for detecting code reuse among android applications. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 62–81. Springer, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-37300-8_4
9. Hu, W., Tao, J., Ma, X., Zhou, W., Zhao, S., Han, T.: Migdroid: Detecting app-repackaging android malware via method invocation graph. In: 2014 23rd International Conference on Computer Communication and Networks (ICCCN), pp. 1–7 (August 2014)
10. Huang, H., Zhu, S., Liu, P., Wu, D.: A framework for evaluating mobile app repackaging detection algorithms. In: Huth, M., Asokan, N., Čapkun, S., Flechais, I., Coles-Kemp, L. (eds.) TRUST 2013. LNCS, vol. 7904, pp. 169–186. Springer, Heidelberg (2013)
11. Google Inc. Android dvelopment guide: Signing your applications (2014), http://developer.android.com/guide/publishing/app-signing.html
12. Oracle Inc. Mysql (2014), http://www.mysql.com/
13. Symantec Inc. Android threats getting steamy (May 7 (2011), http://www.symantec.com/connect/blogs/android-threats-getting-steamy
14. J. Craig Venter Institute. Sift (2014), http://sift.jcvi.org/
15. Li, S.: Juxtapp and DStruct: Detection of Similarity Among Android Applications. PhD thesis, EECS Department, University of California, Berkeley (2012)
16. Oberheide, J.: Dissecting the android bouncer (2012), https://jon.oberheide.org/files/summercon12-bouncer.pdf
17. Sanfilippo, S.: Redis (2014), http://redis.io/topics/sponsors

18. Studios, H.: Fruit ninja (2013), http://halfbrick.com/
19. Ulrich, B., Paolo, M.C., Clemens, H., Christopher, K., Engin, K.: Scalable, behavior-based malware clustering. In: Proceedings of Network and Distributed System Security Symposium 2009. Citeseer (2009)
20. Wikipedia. Color histogram (2014), http://en.wikipedia.org/wiki/Color_histogram
21. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of "piggybacked" mobile applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 185–196. ACM (2013)
22. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy, pp. 317–326. ACM (2012)
23. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109 (May 2012)