FSquaDRA: Fast Detection of Repackaged Applications

Yury Zhauniarovich[1], Olga Gadyatskaya[1,2], Bruno Crispo[1], Francesco La Spina[1], and Ermanno Moser[1]

[1] Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
{zhauniarovich,gadyatskaya,crispo,laspina,moser}@disi.unitn.it
[2] Interdisciplinary Center for Security, Reliability and Trust, University of Luxembourg, Luxembourg City, Luxembourg

Abstract. The ease of Android applications repackaging and proliferation of application clones in Google Play and other markets call for new effective techniques to detect repackaged code and combat distribution of cloned applications. Today all existing techniques for repackaging detection are based on code similarity or feature (e.g., permission set) similarity evaluation. We propose a new approach to detect repackaging based on the resource files available in application packages. Our tool called FSquaDRA performs a quick pairwise application comparison (full pairwise comparison for 55,000 applications in just 80 hours on a laptop), as it measures how many identical resources are present inside both packages under analysis. The intuition behind our approach is that malicious repackaged applications still need to maintain the "look and feel" of the originals by including the same images and other resource files, even though they might have additional code included or some of the original code removed.

To evaluate the reliability of our approach we perform a comparison of the FSquaDRA similarity scores with the code-based similarity scores of AndroGuard for a dataset of randomly selected application pairs, and our results demonstrate strong positive correlation of the FSquaDRA resource-based score with the code-based similarity score.

Keywords: Smartphones, Repackaging, Mobile applications.

1 Introduction

Mobile ecosystems today represent a huge and fast growing market. Success stories of such companies as Rovio (with the Angry Birds game) attract to the mobile business vast amounts of developers. Yet, the developers can suffer from monetary and reputation losses when their applications are stolen and appear on the markets *repackaged*.

The problem of application (app for short) stealing on Android stems from the fact that at present it is not very difficult to repackage an Android app.

Applications are usually signed with a self-signed certificate. Thus, an adversary can easily change the code and sign the app with his own certificate. At present, neither the official Google Play market nor alternative markets do not detect if an application has been repackaged. At the same time, there is a strong aspiration from adversaries to steal applications. They can earn monetary profits either by changing the revenue destination of advertisement libraries, or by embedding malware, which can transform phones into controllable “zombies”. Thus, to maintain the healthiness of the ecosystem there is a strong need to detect the repackaged applications and prevent their distribution.

Currently the problem of Android app repackaging is widely explored and several solutions to identify plagiarized applications were proposed, e.g., [9,7,6,10,16,12]. All these solutions are based on features extracted from the app code. However, it is clear that the code itself is often impacted by the repackaging process: the added malicious functionality (new advertisement libraries and/or malware code) modifies the code of the app. Additionally, the usage of obfuscation libraries during repackaging can further modify the code [11]. Moreover, adversaries can simply replicate some initial behaviour of an app (so called app spoofing [14]). Obviously, the detection rates of repackaging for a code similarity-based techniques decrease under the influence of these factors. Notice that the availability of various tools like smali/backsmali [4] or apktool [3] greatly alleviates the task of code changing and application repackaging.

Yet, it is not only the code that defines an app. Nowadays, smartphones have powerful processors, advanced video and audio systems that are able to support screens with very high resolutions and to produce sounds of high quality. These factors lead to the constant demand of attractive apps. Therefore, to become popular an app should not only include the code with interesting functionality, but should also contain attractive layouts, images and other supplementary resources, which become an integral part of the user experience. These resource files (resources, for short) are delivered on the device packaged together with the code, and are now an inseparable part of modern mobile apps.

This paper proposes an approach to detect repackaged apps based on comparison of the content of the resource files forming Android app packages. Our approach relies on the observation that usually Android packages (`apk` files) include a significant number of resources, and that malicious repackagers aim to change the applications in a way they resemble the originals as much as possible. Therefore, the code parts may change but the resource files (including icons, images, music and video files, etc.) often remain the same.

To be practical, the approach of detecting repackaged applications based on resource files comparison needs to be fast enough, considering the vast number of Android apps (currently there are more then 700,000 apps only in the official market). Thus, a simple pairwise comparison of all files inside two compared apps is not quite scalable because the complexity is proportional to the product of the number of files inside two packages multiplied by the average size of a file. Luckily, during the process of app signing a hash of each file inside the `apk` is computed and stored in the package. We leverage this information to compute the similarity of applications. Thus, our approach is fast enough to be used even for comparing applications pairwise.

To our knowledge, we are the first who propose to detect Android repackaged applications based on similarities in resource files, and not on the ones in the code. This paper contains the following contributions:

– We propose a novel technique to detect repackaged Android applications based on files included in the packages.
– Using the peculiarities of Android app signing process we develop a very fast algorithm that can be used for pairwise comparison of apps. FSquaDRA managed to compare on average 6700 app pairs per second on our dataset using a commodity hardware. This number shows that our approach clearly outperforms all available solutions based on pair-wise code comparison.
– Understanding the importance of Android app repackaging problem we release our tool as open-source[1] to drive the research in this direction.
– We evaluate the practicality of our approach by comparing the resource-based similarity score produced by FSquaDRA with the code-based similarity score computed by the open-source AndroGuard tool [8,2]. Our experiments show that the FSquaDRA similarity score is strongly correlated with the AndroGuard code similarity score.
– We evaluate the effectiveness of the FSquaDRA on a dataset with more than 55000 applications crawled on Google Play and 7 alternative markets, and report repackaging rates for this dataset.

2 Our Approach

Android applications are spread across the devices in the form of Android packages (`apk` files) that contain code, manifest, libraries and resource files compressed in a `zip` archive. Thus, each app includes not only the code, but also a large set of supplementary files being an integral part of the Android package. This is confirmed by our dataset that consists of 55000 apps. For this dataset on average there are 315.56 files inside an Android package with maximum value of 11099 files and minimum of 4 (we present the details of our dataset later on).

Previously, to detect repackaged applications researchers considered predominantly the code (`classes.dex`) and the manifest `AndroidManifest.xml` files. We propose to use the full set of files inside apks to detect repackaging.

Our intuitions are as follows. An adversary, who clones an application, seeks to resemble the original one as much as possible, thus, increasing the probability of the clone installation. In Android apps code is loosely coupled with resources giving the adversary a possibility to easily change the code. For example, the legitimate Opera Mini application and its repackaged version containing malware [13] coincide in 230 out of 234 files inside those packages.

For the scope of this paper we consider two cases of repackaging: (malicious) *plagiarism*, when two application packages include the same files but are signed

[1] `https://github.com/zyrikby/FSquaDRA`

by different developers (with different certificates), and (benign) *rebranding*, when two application packages include the same files and are signed by the same certificate.

Using binary comparison of files, which constitute two Android applications, it is possible to understand to what extent these two apps are similar. Unfortunately, binary comparison is not a cheap operation. Moreover, a file in the first app should be compared against each file in the second package. These overheads may be considerably reduced using comparison of the file digests (hashes). Our tool uses this technique to calculate the similarity between two applications. At the same time, digest computation against the content of a file requires considerable resources consumption and, thus, directly cannot be used in a tool that has to process significant amount of apks. To overcome this limitation we use the hashes calculated during the application signing process. Thus, the overhead for hash computations does not affect our tool. To facilitate the understanding of our algorithm we first describe the code signing process used in Android.

Android application signing background. An unsigned `apk` file contains a compiled code and a set of resources. In Android, all Java code is compiled into one file called `classes.dex`. Moreover, some of the xml files can also be compiled into a binary format. Besides compiled files, an Android package usually contains non-compiled resource such as icons, drawables, text files, different binary files, etc. This archive is then signed with the standard Java signing tool called *jarsigner*. This tool creates a special directory inside the archive called `META-INF`, where it stores the information related to the code signing process.

We are only interested in the first step of the signing process, which produces the main manifest file (`MANIFEST.MF`). During this step the *jarsigner* tool calculates a digest of each file inside the unsigned `apk` and writes it into the `MANIFEST.MF` file. On Android, the *SHA1* algorithm is used to compute the digest of file content.

The manifest file consists of the main attributes section and and a set of per-entry attributes, one entry for each file contained in the unsigned apk file. These per-entry attributes store information about the file name (relative path) and the digest encoded using the `base64` format. Therefore, after the first step of application signing process a SHA1 digest of the content of each file is available in the manifest file. These hash values are later used in our tool.

The algorithm and implementation details. Protocol 1 describes the algorithm implemented in FSquaDRA for pairwise comparison of apps. In Line 2 of Protocol 1 we select all apk files located under the directory, the path to which is specified by the variable *path* provided as an argument. After that, in Lines 6-10 FSquaDRA extracts the required information from the apk files. At first, our tool gets the name of the file. Then it extracts the attributes of the apk using the `getApkAttributesToMemory` method. In particular, it iterates over the entries in the `MANIFEST.MF` file and writes the results into a map, which key corresponds to the relative path of a file inside the package and value is equal to the SHA1 hash of the file. Additionally, during this step FSquaDRA extracts

Protocol 1. The algorithm of application comparison

```
1: $ApkAttr_{list} \leftarrow [\ ]$
2: $Apk_{list} \leftarrow getApkFileList(path)$
3: \\ Get application attributes
4: for all $A_i \in Apk_{list}$ do
5:     $ApkName_i \leftarrow getApkName(A_i)$
6:     $Attr_i \leftarrow getApkAttributesToMemory(A_i)$
7:     Add $(fileName_i, Attr_i)$ to $ApkAttr_{list}$
8: end forall
9: $size \leftarrow length(ApkAttr_{list})$
10: \\ Pairwise comparison of applications
11: for $(k = 0; k < size; k++)$ do
12:     $hashes_k \leftarrow getFileHashesSet(Attr_k)$
13:     $certs_k \leftarrow getCertHashes(Attr_k)$
14:     for $(l = k + 1; l < size; l++)$ do
15:         $hashes_l \leftarrow getFileHashesSet(Attr_l)$
16:         $certs_l \leftarrow getCertHashes(Attr_l)$
17:         $jSim \leftarrow getJaccardIndex(hashes_k, hashes_l)$
18:         $sameCert \leftarrow certsTheSame(certs_k, certs_l)$
19:         OUT: $ApkName_k, ApkName_l, sameCert, jSim$
20:     end for
21: end for
```

the developer certificates, which have been used for the application signing, and stores into *Attr* object the digests computed over these certificates. This allows us to reduce the memory consumption of FSquaDRA and speed up the certificate comparison process. The name of the app file along with the object $Attr_i$ containing all required application attributes are stored into the $ApkAttr_{list}$ list.

Lines 15-25 show how the comparison of applications is performed. The similarity score (the FSquaDRA similarity, or the fss score for short) corresponds to the Jaccard similarity coefficient (expressed by Formula 1) computed over the sets of file hashes extracted in Line 8.

$$jSim(H_k, H_l) = \frac{|H_k \cap H_l|}{|H_k \cup H_l|} \tag{1}$$

We implemented our algorithm in Java. We did not parallelize it intentionally (i.e., our tool runs in a single-thread program). This allows us to calculate the net time required to run our comparisons and predict the execution time and memory consumption. An increase of a dataset results in the linear growth of the execution time for attributes extraction, while the pairwise comparison operation cumulative time rises quadratically (in the number of apks under consideration). In the current implementation the memory consumption grows linearly with the number of applications. The code is availabe under the Apache-2.0 license[2].

3 Evaluation

Our dataset consists of 55,779 Android applications. The dataset collection was performed during June-July of 2013. During this period we explored 8 different markets: the official *Google Play*[3] market (13,223 apps; including 500 top

[2] `https://github.com/zyrikby/FSquaDRA`
[3] `https://play.google.com/store/apps`

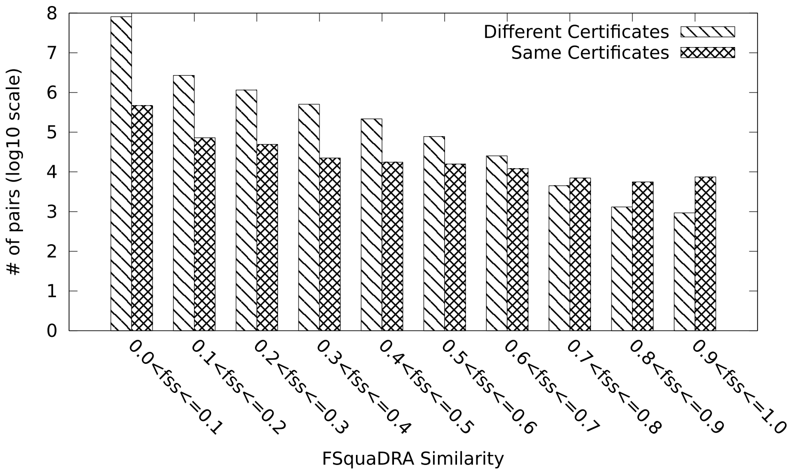


Fig. 1. Histogram of app repackaging rates detected with FSquaDRA (logarithmic scale)

free apps for each category) and 7 third-party stores: *androidbest*[4] (1662 apps), *androiddrawer*[5] (2857 apps), *androidlife*[6] (1678 apps), *anruan*[7] (4232 apps), *appsapk*[8] (2679 apps), *pandaapp*[9] (14,143 apps), and *SlideME*[10] (15,305). Our dataset occupies 317.4 GB of disk space.

We have run FSquaDRA on the collected app dataset on a Mac Book Pro laptop with 2.9 GHz Intel Core i7 Processor with 2 cores, and 8GB 1600 Mhz DDR3 memory. FSquaDRA required 15.10 hours to load all apk attributes in memory for our complete dataset, and 64.41 hours to compute the similarity scores for all apk pairs ($>10^9$) in our dataset consuming less than 6GB of RAM. On the dataset FSquaDRA performs on average 6700 app pair comparisons per second. We consider these results quite encouraging, as pairwise app comparison for code-based similarity metrics cannot be executed in comparable time.

Figure 1 presents a histogram of positive fss scores distribution for our dataset of 55779 applications (in logarithmic scale). Notice that the app pairs with fss>0 constitute approximately 5.41% of the total app pairs number for our dataset. To simplify presentation we break down the fss values into 10 bins in the range (0, 1]. In Fig. 1 we can see that the vast majority of the application pairs with detected resource similarity have the fss score in the range (0, 0.1], and that for the fss score in the range (0.7, 1] there are more app pairs

[4] `http://androidbest.ru/`
[5] `http://www.androiddrawer.com/`
[6] `http://androidlife.ru/`
[7] `http://www.anruan.com/`
[8] `http://www.appsapk.com/`
[9] `http://android.pandaapp.com/`
[10] `http://slideme.org/`

with the same certificate detected by FSquaDRA than app pairs with different certificates. We provide more insight why this is the case in the sequel.

To evaluate the quality of our approach we would like to compare our results with some state-of-art code similarity-based repackaging detection technique, e.g., [16,15,7,10]. Unfortunately, these tools were not released publicly, and we were not able to obtain them. Similar problem was also reported in [11], where the authors used AndroGuard as a freely available tool for comparison of code similarity in apks. Following this approach, we use AndroGuard to provide us a metrics of code similarity for app pairs.

The main question we would like to investigate is whether the FSquaDRA similarity metrics is correlated with the AndroGuard code similarity metrics. This can be interpreted twofold:

- *Problem of false positives.* For apps that FSquaDRA classifies as similar, are they similar also according to the AndroGuard classification (and vice-versa)? If our tool classifies an app pair as similar, but there is no actual code similarity, this pair can be interpreted as false positive. It is obvious that it is impossible to completely avoid false positives for FSquaDRA because common resources, such as, e.g., open source sound and image files, can increase the FSquaDRA metrics, while the code would be different. So here we are interested in strong correlation of the similarity metrics values.
- *Problem of false negatives.* For apps that FSquaDRA classifies as completely different, are there many app pairs sharing code similarities according to AndroGuard? Again, it is not possible to completely avoid false negatives due to the different nature of code similarity and resource similarity, but we would like to assert that the false negatives rate is not too high.

Notice that in this section we interpret the AndroGuard code similarity score as the ground truth. We have performed manual inspection of some application pairs to confirm the findings of FSquaDRA (reported further), but it is impossible to inspect manually a substantial subset of our dataset. Therefore we have to rely on the code similarity metrics as the ground for evaluating FSquaDRA reliability.

The AndroGuard algorithm which computes the similarity score (*ags* for short) of two apps is presented in [8]. The similarity score is based on the analysis of Dalvik code of an app pair and detection of identical, similar and different (new or deleted) methods in the apps. To perform this, the algorithm a) generates a signature for each method of each application, b) identifies all methods that are identical in both apps, c) discovers all methods that are similar. A signature is generated based on the method control flow information, used API calls and exceptions inside the method. If two signature hashes are identical then the methods are considered identical. To compute the similarity between methods Normalized Compression Distance (NCD) [5] is used.

AndroGuard however was found to be not very reliable, as its similarity metrics was discovered to be not commutative. That is, for two apks A and B, it could be that $ags^*(A, B) \neq ags^*(B, A)$, where ags^* is the value computed by the AndroGuard tool directly. We have decided to still use the existing Andro-

Guard implementation, but to adjust the AndroGuard score. We have experimented with a series of app pairs, and have established that the metrics ags= $(ags^*(A,B) + ags^*(B,A))/2$ is more faithful than the original ags^* similarity score, and we have used this metrics for comparison with FSquaDRA results.

To compute a similarity value for two applications AndroGuard takes significantly more time than FSquaDRA, and it was not possible to compute the similarity metrics for the whole app corpus we have crawled. E.g., it takes approximately 65 seconds on average to compare one pair of apps using AndroGuard (the actual time of comparison depends a lot on the similarity of apps in the pair; it takes significantly less time to compare very similar apps than completely different ones). We cannot also rely on a straightforward random selection of app pairs, because it is clear from Fig. 1 that, e.g., the share of app pairs with fss similarity in the range (0, 0.2] is a lot larger than the share of app pairs in (0.8, 1.0], which is as interesting. Therefore, we have performed a random selection of 100 app pairs with same certificate and 100 app pairs with different certificates from each bin with non-null fss metrics, and we have computed the AndroGuard similarity metrics for these pairs (2000 pairs total). This selection enables the best selection of an app pairs corpus with different fss metrics, and without strong predominance of some fss value range. To evaluate the false negative rates we have randomly selected 100 apk pairs with same certificate and 100 apk pairs with different certificates from the dataset with fss=0.

Table 1 presents summary statistics computed for the randomly selected app pairs. Notice that for non-null fss values we compare separately app pairs with same certificate and with the different ones, as these two groups are different by nature. This observation is indeed reinforced by the data we have. Fig. 2(a) presents a scatterplot of the fss and ags similarity metrics values for the selected app pairs with different certificates (potentially plagiarised). We can see the strong correlation of the values from the figure. This is confirmed by the data: the standard Pearson's product-moment correlation computed for data in this figure is 0.791. Notice that any value $\geq$0.5 is commonly considered as strong correlation. Testing for the null-hypothesis (that true correlation is non existent) for this dataset gives that the 95% confidence interval is [0.767, 0.813]; and the p-value$\approx 10^{-16}$, so we can safely reject the null-hypothesis. The sample mean of the difference (fss-ags) for each selected app pair with different certificates is approximately equal to -0.047, with standard t-test rejecting the null-hypothesis (the p-value$\approx 10^{-12}$), and the 95% confidence interval for true mean [-0.052, -0.029]. The standard deviation for the difference (fss-ags) is 0.186. We also present a boxplot for this difference in Fig. 3(a).

These data confirm that FSquaDRA can be an effective tool to detect repackaged applications, as the fss similarity values for app pairs with different certificates are highly correlated with code-based similarity metrics of AndroGuard; and the average difference in the similarity metrics produced by FSquaDRA and by AndroGuard is not significant.

Fig. 2(b) presents a scatterplot of the fss and ags similarity metrics for the randomly selected apk pairs signed with the same certificate (potentially rebranded). The standard Pearson's product-moment correlation for this dataset is approximately 0.58 (the null-hypothesis on correlation is rejected, with 95% confidence interval for correlation [0.538, 0.62] , and p-value $\approx 10^{-16}$). This can be still interpreted as a strong correlation, but it is less strong than for the apk pairs with different certificate. The sample mean for the difference (fss-ags) in this dataset is approximately equal to -0.27 (standard t-test reports 95% confidence interval for true mean [-0.292, -0.259], and the null-hypothesis for sample difference mean being zero is rejected with p-value$\approx 10^{-16}$). This means that on average for apks signed with the same certificate FSquaDRA tends to estimate their similarity score noticeably lower than the code-based similarity score computed by AndroGuard. These findings can be intuitively explained by the fact that developers tend to reuse the code patterns across their products. For app pairs signed with the same certificate it is clear that they can contain similar code snippets with high probability. Therefore higher code similarity score is expectable.

We can also see from Fig. 2(b) that there is a lot of app pairs with very high AndroGuard similarity score, but varying FSquaDRA similarity score, which are most probably the pairs impacting the correlation coefficient for this dataset. We have manually inspected some of these pairs and have managed to find several patterns, when such situations occur. One of the most common observed case is when the same code is used for displaying different content. For instance, in our dataset we found several applications, which were developed to display books. For every book a single application has been developed. All these applications use the same code but the resources (the book chapters) are different. Thus, our tool shows low similarity score (because still some files, e.g., `classes.dex`, are the same), while according to the code similarity score the applications in the pair are the same. Similar behaviour we also witnessed with other categories of applications, which display the same type of content, e.g., for wallpaper apps and widgets. Another interesting example, which falls into this category, is when the apps in the pair provide a UI customization functionality for the third application. In this case, AndroGuard produces high similarity score for such pairs of apps, while because of the difference of the UI components FSquaDRA reports low similarity.

The lower correlation of the metrics can be also attributed to the usage of the same ad libraries. This happens when the fraction of the code produced by a developer significantly smaller than the ones brought by ad libraries. In this case AndroGuard falsely detects applications as repackaged, while FSquaDRA produces more credible results (because the applications are different).

Fig. 3(b) presents a boxplot for the sample difference (fss-ags). In comparison with Fig. 3(a), we can notice that for apk pairs with the same signature the range of the similarity scores difference is larger. Our data suggests that FSquaDRA may not be as efficient for detecting repackaging in apps signed with the same certificate (rebranded), as it is for the apps signed with different certificates (plagiarized). Nevertheless, correlation of the FSquaDRA score with the code-based similarity score of AndroGuard is still strong (>0.5).

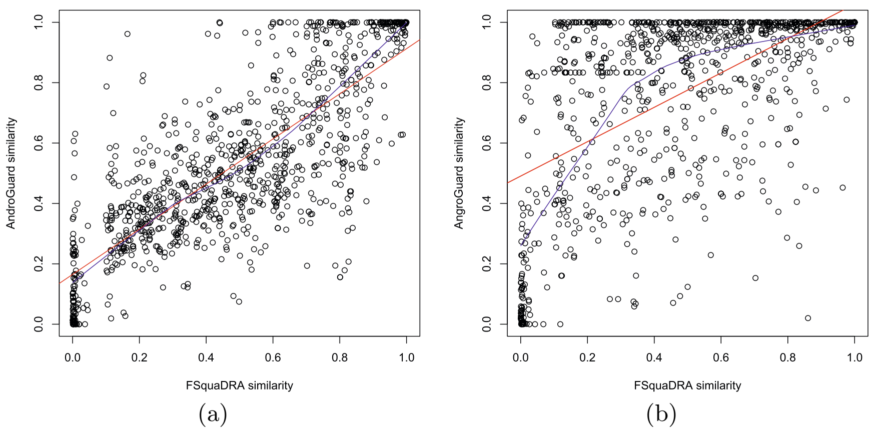


Fig. 2. Scatterplots of FSquaDRA similarity vs. AndroGuard similarity for the pairs: a) signed with different certificates; b) signed with the same certificate. The red line is the line of best fit, the blue curve is the LOWESS (locally weighted scatterplot smoothing line).

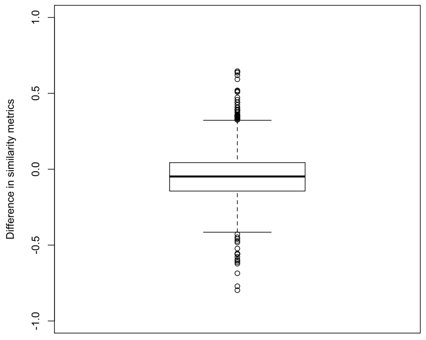
Finally, let us consider the difference (fss-ags) for the randomly selected app pairs with fss=0. The sample mean of (fss-ags), or, simply, of the ags similarity score taken with the negative sign, for these app pairs is approximately -0.041, with the 95% confidence interval for the true mean [-0.051, -0.0309], and the standard deviation for this dataset is approximately equal to 0.0737. Thus, FSquaDRA does not error a lot on average. From these statistics we can see that for apk pairs not marked as similar by FSquaDRA AndroGuard does not see significant code similarity either, even for applications signed with the same certificate. Therefore we can conclude that if developers do not include any similar resouces in apps, they also mostly do not reuse code (this is often the case of apps produced by companies). We do not report the correlation coefficient for this type of dataset, as the fss score equals to 0.

4 Cross-Market Repackaging

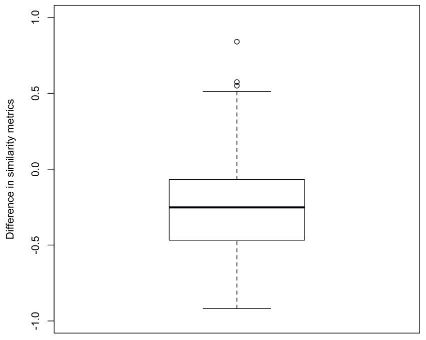
After asserting that FSquaDRA produces similarity metrics that is valuable for detecting repackaged applications, being strongly correlated with the code similarity metrics, we look into repackaging rates corresponding to the markets under consideration, and investigate clusters of repackaged applications. Notice that clearly any FSquaDRA score greater than 0 for a pair of apks can be an indication that these apks are clones. However, to increase the certainty of detecting clones we have chosen the fss value of 0.7 to be a reliable threshold for repackaging. Based on our observations, we consider it a good starting point for resource similarity score sufficient to reliably detect clones, and we leave the task of identifying the threshold precisely for future work.

Table 1. Summary statistics for comparison of the fss and ags metrics

Sample	Statistics	Value	Details
App pairs with non-null fss with different certificates in comparison with ags; 1000 app pairs	Mean of difference fss - ags	-0.04122781	Standard one sample t-test 95% confidence interval: [-0.05278174, -0.02967388] p-value = 4.62e-12
	Standard deviation for difference fss - ags	0.1861895	
	Median	-0.04799	
	Correlation coefficient of fss and ags values	0.7919082	Pearson's product-moment correlation 95% confidence interval [0.7675988, 0.8139426] p-value $\leq$ 2.2e-16
App pairs with non-null fss with same certificates in comparison with ags; 1000 app pairs	Mean of difference fss - ags	-0.276119	Standard one sample t-test 95% confidence interval: [-0.2928976, -0.2593405] p-value = 2.2e-16
	Standard deviation for difference fss - ags	0.2703832	
	Median	-0.25180	
	Correlation coefficient of fss and ags values	0.580733	Pearson's product-moment correlation 95% confidence interval [0.5381128, 0.6203911] p-value $\leq$ 2.2e-16
App pairs with null fss with mixed certificates in comparison with ags; 200 app pairs	Mean of difference fss - ags	-0.04124	Standard one sample t-test 95% confidence interval: [-0.05152188, -0.03095351] p-value = 1.777e-13
	Standard deviation for difference fss - ags	0.07375432	
	Median	-0.01304	
2200 app pairs, fss including app pairs with the same ags; and different certificates, and with fss=0 and fss>0	Mean of difference fss - ags	-0.14800	Standard one sample t-test 95% confidence interval: [-0.1585031, -0.1374917] p-value = 2.2e-16
	Standard deviation for difference fss - ags	0.2512748	
	Median	-0.09894	
	1^{st} quartile	-0.27380	
	3^{rd} quartile	0.00000	
	Correlation coefficient of fss and ags values	**0.7149053**	Pearson's product-moment correlation 99% confidence interval [0.6869681, 0.7407324] p-value ¡ 2.2e-16



(a)



(b)

Fig. 3. Boxplot of the difference of FSquaDRA similarity with AndroGuard similarity for app pairs with fss>0: a) signed with different certificates; b) sighed with the same certificate

Cross-market comparison. Table 2 presents the repackaging rates of Google Play applications cloned in other markets. Under the assumption that the Google Play market is the source of original applications, this table reports how many cloned pairs were detected with the fss score greater than 0.7, and the total number of apk pairs with fss>0 for all markets of our study compared with Google Play (the corresponding subset of our dataset). In this experiment we have compared each crawled apk in Google Play with each apk crawled in the considered third party markets. We also provide the processing time required for each market comparison with Google Play. Notice that for all markets the number of app pairs with the fss score greater than 0.7 is not very significant. To understand better how the big is the subset of potentially repackaged applications we also provide the total number of app pairs with $fss>0$ detected, and the number of pairs with $fss>0$ and signed with different certificates.

From Table 2 we can observe that the markets with the highest repackaging rates are androiddrawer (16.16% of app pairs have similarity of resources $fss>0$) and Google Play (10.31% of app pairs have $fss>0$). We suspect that this is the case because these markets are more popular sources of apps, in comparison with others; and malicious repackagers that seek acquiring significant ad revenues or big user base for their botnets may target more popular markets. Yet, this intuition needs to be confirmed with more data, and there can be other plausible explanations.

Table 2. Results of experiments, each market in comparison with Google Play

Market	**Repackaging Rates**				**Time**	
	Same signature # pairs ($fss>0.7$)	Different signature # pairs ($fss>0.7$)	Total $fss>0$ (% of total pair #)	Total $fss>0$ with diff. cert. (% of total pair #)	Loading apk attributes in memory	Processing
androidbest	27	10	714258 (3.25%)	713194 (3.24%)	14.16 min	12.274 min
androiddrawer	528	14	6108547 (16.16%)	6097437 (16.14%)	15.46 min	56.02 min
androidlife	41	44	1145396 (5.16%)	1143400 (5.15%)	14.24 min	15.67 min
anruan	106	97	3349271(5.985%)	3347895 (5.982%)	15.26 min	36.11 min
appsapk	422	86	2105334 (5.94%)	2094716 (5.91%)	15.66 min	22.52 min
Google Play	1897	1301	9019858 (10.31%)	8985401 (10.27%)	13.28 min	59.97 min
pandaapp	755	381	10741872 (5.74%)	10726743 (5.73%)	28.52 min	136.65 min
SlideME	475	579	9496874 (4.69%)	9481029 (4.68%)	25.96 min	97.07 min

Application clusters. Repackaged applications can form clusters (a set of repackaged apps stemming from some original application). We tried to elicit and analyze strongly connected clusters containing applications with very similar resources. The results produced by FSquaDRA can be interpreted as an undirected labelled graph, where nodes correspond to the applications in our dataset and edges represent similarity relationship between two applications, labelled with the fss similarity score. Thus, to find the clusters of applications we used the following algorithm. At first, we selected all pairs, which had shown the FSquaDRAsimilarity value more than 0.7. After that in the resulting graph we searched for connected components (i.e., set of connected nodes), which corresponded to application clusters. We looked for clusters that have 3 and more nodes. Using this approach we discovered 71 cluster, the largest of which included 9 applications.

We have investigated manually some of the clusters, and we report on the largest two of them (smaller clusters are not reported for the lack of space). The largest cluster with 9 nodes contains applications from 3 different markets (4 from Google Play, 4 from SlideME and 1 from appsapk), all signed with different certificates. The nodes are connected with 8 edges; similarity scores for app pairs not connected by an edge vary in the range [0.61, 0.7). The cluster with 8 applications contains packages distributed on 5 different markets (2 come from Google Play, 3 from SlideME, 1 from anruan, and 2 from pandapp). These 8 applications are connected by 7 nodes, and the fss scores for the app pairs not connected by an edge vary in [0.4, 0.6). In this cluster 3 applications (from anruan and pandapp) were signed by the same certificate, and others were signed with different certificates.

After we manually inspected all applications in these clusters, we discovered that these apps were legitimate applications and not maliciously repackaged. These "false positives" appeared because all apps in the cluster used the same popular library ActionBarSherlock [1], which is supplied with lots of files. Additionally, the applications contained a very limited number of their own unique files, and thus FSquaDRA falsely detected them as repackaged applications. We performed also an analysis using AndroGuard and found out that the code files were also very poisoned with this library. AndroGuard similarity scores for these clusters were in the range [0.46, 0.96]. Therefore, in the shadow of the methodology selected for our analysis this is still a good result for our tool. However, this example clearly shows that it is desirable to implement techniques for automatic library resources detection and exclusion, similarly as it is done for code in [7,15]. We leave this problem for the future work.

5 Related Work

Existing works in repackaging detection on Android mostly focus on code similarity and do not consider the resource similarity, in contrast to FSquaDRA. Unfortunately, it is impossible to compare our tool with others because existing research tools, excluding AndroGuard, are not publicly available.

In [16] the authors search repackaged applications in third-party markets using Google Play as a baseline. A tool called DroidMOSS uses fuzzy hashing of code to calculate a fingerprint of the app and then computes the edit distance between two fingerprints to compute the similarity score. The analysis performed in [16] shows that 5-13% applications hosted in alternative markets are repackeged. These conclusions agree with our findindings reported in Sec. 4.

In the paper [15] the authors further investigate the problem of repackaged apps and concentrate on detection of piggybacked applications (repackaged apps that carry a malicious payload). To find these apps the authors perform code decoupling into primary and non-primary modules, and compute a fingerprint for each primary module, which contains the main functionality. After that while iterating over the fingerprints the linearithmic algorithm detects apps with similar primary modules, which are considered as piggybacked candidates. Finally, piggybacked apps are detected comparing the sets of non-primary modules of these similar apps. The experiments show the presence of 1.3% piggybacked apps in the dataset.

Paper [6] presents the DNADroid tool detecting cloned (plagiarized) applications. Using the semantic similarity of apps the tool detects potential clone candidates. At the second step, the tool extracts Program Flow Graph of each method in compared applications, and, based on the subgraph isomorphism problem as a final criteria of method similarity, computes similarity score of the apps. DNADroid managed to detect 191 cloned pairs (0% false positives was reported). The authors also compared their tool with AndroGuard [2]. On 191 pairs AndroGuard failed for 24 pairs and produced very low similarity score for 10 pairs meaning that it missed 18% of the pairs found by DNADroid. Continuing the work on DNADroid [6] Crussell et al. developed a new tool AnDarwin [7], which extracts features from app code and compares them, instead of pairwise comparisons of code, allowing to perform large-scale analysis of Android applications. On a dataset of 265,359 third-party apps collected from 17 markets DNADroid detected 4,295 cloned and 36,106 rebranded applications (cloned apps with the same signature).

The authors in the work [12] concentrate on investigation which applications are likely to suffer from being plagiarised, and how to detect plagiarised applications uploaded to a market. The authors analysed the meta-information of 158,000 applications. They detected that 29.4% of applciations are more likely to be plagiarised, based on the assumption that it was more likely that a malicious developer would use for plagiarising the applications, which alredy contained the permissions needed to perform malicios actions.

The paper [10] presents another approach to detect code reuse among Android apps. To discover the similarities 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 tool can detect (a) buggy and vulnerable code reuse (b) known malware instances and (c) pirated applications. 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) that lasted about 200 minutes. As for effectiveness, among the apps from Android Market the authors identified 174 samples containing vulnerable patterns in the in-app billing code and 239 apps containing those in the code using Licence Verification Library. Moreover, they identified 34 new instances of known malware in the alternative Anzhi market.

Recently, a framework for evaluating Android application repackaging detection algorithms has been proposed [11]. In the paper the authors classify currently available approaches for detection of repackaged applications and present a framework that can be used to assess the effectiveness of this kind of algorithms. The framework translates Dalvik bytecode into Java code, applies obfuscation techniques and packs back the code into the Dalvik representation. To assess the effectiveness of a tool is run over real and modified by the framework app. The authors proposed to assess 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 [2] – 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 applications. Notice that FSquaDRA will successfully pass the tests of [11], because it does not rely on code similarity.

6 Conclusions

In this paper we present an approach to detect Android application repackaging based on the apk resource files, and an implementation of this approach in the FSquaDRA tool. Leveraging hash files of resources already present in apks, FSquaDRA is capable of fast pairwise apk comparison. It computes the Jaccard similarity score for compared apks and classifies them as similar if substantial number of resource files are the same in both packages.

We have evaluated practicality of FSquaDRA in two aspects: whether it gives results similar to the code-based app repackaging detection techniques, and whether it is fast enough to handle significant number of apks. Our results are encouraging. The FSquaDRA resource similarity score is strongly correlated with the AndroGuard code similarity score, especially for the apks signed with different certificates, and thus, potentially, plagiarized. FSquaDRA is also has good performance, as it was able to process a dataset of more than 55000 apks on a laptop in less than 80 hours. Notice that our implementation was not optimized for better performance, as it is single-threaded. Yet, the approach can be easily parallelized using different parallelization algorithms for pairwise comparison.

The obvious limitation of the current tool is that an adversary who is familiar with the approach can easily change all resource files in the package to make his plagiarized application virtually undetectable by FSquaDRA. Resource similarity metrics can be hardened against this by looking into files themselves rather than just comparing the digests, but it will lead to performance losses (which can become comparable with those of the code-based repackaging detection techniques if implemented reasonably). The most promising, to our point of view, is a hybrid approach, when repackaged applications are detected using both approaches, code and resource comparison. We believe this is a very interesting research direction.

Another interesting future work direction is to look into the data produced by FSquaDRA looking for patterns and interesting findings, such as the fact that on average applications signed with the same certificate have higher code similarity score than resource similarity score, while this difference is not so evident in the apps signed with different certificates.

FSquaDRA opens an avenue of enhancement for app plagiarism detection algorithms, and not only for Android. For other ecosystems, such as iOS or Windows Phone, that request the developers to submit the full source code and resources before publishing apps on the market our technique can be used to improve the on-market plagiarism detection algorithms by complementing the code similarity-based approaches.

Acknowledgements. This work has been partially supported by the FP7-ICT SecCord Project 316622 funded by the EU, and the TENACE PRIN Project (grant no. 20103P34XC) funded by the Italian MIUR.

References

1. ActionBarSherlock, `http://actionbarsherlock.com/`
2. AndroGuard: Reverse engineering, Malware and goodware analysis of Android applications, `https://code.google.com/p/androguard/`
3. Android-apktool: A tool for reverse engineering Android apk files, `https://code.google.com/p/android-apktool/`
4. Smali: An assembler/disassembler for Android's dex format, `https://code.google.com/p/smali/`
5. Cilibrasi, R., Vitányi, P.M.B.: Clustering by compression. IEEE Transactions on Information Theory 51, 1523–1545 (2005)
6. 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)
7. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar android applications. In: Proc. of Esorics 2013 (2013)
8. Desnos, A.: Android: Static analysis using similarity distance. In: Proc. of HICSS 2012, pp. 5394–5403 (2012)
9. Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: Adrob: examining the landscape and impact of android application plagiarism. In: Proc. of MobiSys 2013, pp. 431–444 (2013)
10. 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)
11. 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)
12. Potharaju, R., Newell, A., Nita-Rotaru, C., Zhang, X.: Plagiarizing smartphone applications: attack strategies and defense techniques. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS 2012. LNCS, vol. 7159, pp. 106–120. Springer, Heidelberg (2012)
13. Protalinski, E.: Warning: New Android malware tricks users with real Opera Mini (July 2012), `http://www.zdnet.com/warning-new-android-malware-tricks-users-with-real-opera-mini-7000001586/`
14. Vidas, T., Christin, N.: Sweetening android lemon markets: measuring and combating malware in application marketplaces. In: Proc. of CODASPY 2013, pp. 197–208 (2013)
15. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of "piggybacked" mobile applications. In: Proc. of CODASPY 2013, pp. 185–196 (2013)
16. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proc. of CODASPY 2012, pp. 317–326 (2012)