Evaluation of Resource-Based App Repackaging Detection in Android

Olga Gadyatskaya[1(✉)], Andra-Lidia Lezza[1], and Yury Zhauniarovich[2]

[1] SnT, University of Luxembourg, Luxembourg, Luxembourg
olga.gadyatskaya@uni.lu, andra.lezza.001@student.uni.lu
[2] Qatar Computing Research Institute, HBKU, Doha, Qatar
yzhauniarovich@qf.org.qa

Abstract. Android app repackaging threatens the health of application markets, as repackaged apps, besides stealing revenue for honest developers, are also a source of malware distribution. Techniques that rely on visual similarity of Android apps recently emerged as a way to tackle the repackaging detection problem, as code-based detection techniques often fail in terms of efficiency, and effectiveness when obfuscation is applied [19,21]. Among such techniques, the resource-based repackaging detection approach that compares sets of files included in apks has arguably the best performance [10,17,20]. Yet, this approach has not been previously validated on a dataset of repackaged apps.

In this paper we report on our evaluation of the approach, and present substantial improvements to it. Our experiments show that the state-of-art tools applying this technique rely on too restrictive thresholds. Indeed, we demonstrate that a very low proportion of identical resource files in two apps is a reliable evidence for repackaging. Furthermore, we have shown that the Overlap similarity score performs better than the Jaccard similarity coefficient used in previous works. By applying machine learning techniques, we give evidence that considering separately the included resource file types significantly improves the detection accuracy of the method. Experimenting with a balanced dataset of more than 2700 app pairs, we show that with our enhancements it is possible to achieve the F-measure of 0.9919.

Keywords: Android security · Repackaging · Resource files

1 Introduction

With more than 1.4 billion active devices and more than 1.6 million of apps only on the official Google Play market, Android is the dominating mobile OS today[1]. Android is an open eco-system, i.e., users can install apps not only from Google Play [6]. This openness led to flourishing third-party markets, e.g., with localized content, or even with stolen apps. Indeed, *application repackaging*, when

The work of Olga Gadyatskaya was supported by the Luxembourg National Research Fund (C15/IS/10404933/COMMA).

[1] According to Gartner http://www.gartner.com/newsroom/id/3169417.

B.B. Brumley and J. Röning (Eds.): NordSec 2016, LNCS 10014, pp. 135–151, 2016.
DOI: 10.1007/978-3-319-47560-8_9

a legitimate app is re-published by adversaries, is polluting Android markets worldwide. It is a known vector of Android malware distribution [26], and not even Google Play is immune to this threat [23]. While the recently spotted Trojans hardly included any useful functionality, users still fell victims to their lure because familiar icons and names were used by the badware[2].

App repackaging detection approaches recently turned to the intuition of visual similarity between original apps and their plagiarized copies [10,15–17,19,20]. Indeed, the users have certain expectations for the "look and feel" of the original apps, and it might be more challenging for malicious repackagers to change the GUI design than to insert, modify or remove some code parts [19]. Among these techniques, arguably the best performance could be achieved by the *resource-based repackaging detection* approach that directly compares the "look and feel" of applications represented by the included images, multimedia, layout and other files. This approach was adopted by, e.g., the FSquaDRA tool [20], the PlayDrone system [17], and the APPraiser framework [10]. These tools compute similarity of two applications based on the number of identical files (resources) included in both packages proportional to the total number of included files (the Jaccard similarity score that ranges in [0,1]).

Although a strong correlation of the resource-based similarity score with the code-based similarity score produced by `Androguard` [5] was previously reported [20], and manual validation exercises were positive [10,17,20], a thorough assessment of the resource-based repackaging detection approach effectiveness has never been done before. Whether it could show reliable results in a practical setting was an open question.

In this paper we close this gap by empirically evaluating resource-based repackaging detection in experiments on a dataset including repackaged and non-repackaged pairs. In particular, we explore the following research questions:

RQ1: Does the resource-based repackaging detection approach work in practice? Can we identify a definitive threshold for the resource-based similarity score that separates classes of repackaged and not-repackaged app pairs with tolerable false positive and false negative rates?

RQ2: Does the effectiveness of the resource-based repackaging detection tool depend on the similarity metric used (i.e., Jaccard similarity used in [10,17,20])? If yes, which similarity metric suits better to the problem of the resource-based repackaging detection?

RQ3: Can we improve the repackaging detection rates with the help of machine learning algorithms?

RQ4: Can the predictive power be improved if different types of resources will be considered separately?

RQ5: What types of resources are more or less susceptible to modifications during the repackaging process?

[2] http://www.welivesecurity.com/2016/02/24/porn-clicker-trojans-google-play-analysis/.

Answering these questions, this work makes the following contributions:

– We practically verified that resource-based approaches [10,17,20] can be indeed used for detection of repackaged applications. We have found the threshold value 0.0629, which can be further used directly in tools [10,17,20] to minimize both false positive and false negative errors.
– Our experiments with several similarity scores showed that the Overlap similarity score achieves the best performance (F-measure 0.9847), while prior works [10,17,20] relied on the slightly less efficient Jaccard similarity.
– We experimented with repackaging detection based on individual scores for distinct resource file types. We used 18 file types as a feature vector, evaluated several classifiers with these features and found that effectiveness of the approach is improved by considering separately different types of files. In the best case, with the non-optimized Random Forest classifier, we achieved F-measure of 0.9919 improving the single score-based approach considerably.
– We investigated the susceptibility to modification in repackaging of the individual resource file types. Our results show that multimedia files, libraries, raw resources and images are least frequently changed in repackaging, while the main `dex` code file, the manifest file and the compiled resources (e.g., strings) are the most frequently changed resource file types.

Our findings underline that resource-based repackaging detection is a practical enhancement to an on-market triage. To stimulate further investigations and adoption of the method, we release our system open-source[3].

2 Resource-Based Repackaging Detection

Resource Files. Resource files are an integral part of any Android application package (apk). They include graphics, texts, layouts, and multimedia content that will be presented to the user to provide a unique user experience. Other types of files in the apk are code files (`classes.dex` and library files) and the manifest file. In this paper, we in fact refer to all files composing an apk as resource files. Resource files are typically numerous (an average apk includes more than 300 files [20]), thus they can be considered representative for the apk.

Upon package signing by the developer, SHA1 digests of all included resource files (and other files) are created and stored in the apk within the `MANIFEST.MF` file. Later, on the device, the hashes are used to verify the integrity of the files constituting the package. The Android application signing mechanism, however, does not protect against integrity violation of the package (repackaging). In malicious repackaging, the adversary strips off the signature of the original developer, decompiles the app, introduces the required changes (e.g., changes the ad library identifier to redirect the revenue streams or injects malicious code), rebuilds the app and signs it again with a new certificate [23].

[3] The code is available at https://github.com/zyrikby/FSquaDRA2.

Resource-Based Similarity Score. The basic intuition behind the resource-based similarity score, which is ultimately leveraged for repackaging detection in [10,17,20], is that, in order to maintain the visual similarity of the repackaged app with the original one, the repackager does not change the resource files at all, or only modifies a fraction of them. Thus, resource files can be used to pinpoint visually similar app pairs.

The resource-based similarity score (*ressim* for short) for a pair of apks is computed in [10,17,20] by applying the Jaccard similarity coefficient to sets of resource file hashes. For two apks A and B with file hash sets H_A and H_B, correspondingly, $Jressim(A, B) = |H_A \cap H_B|/|H_A \cup H_B|$, where $Jressim$ stands for Jaccard resource similarity score. With this formula, two apps with completely different sets of files hashes have the $Jressim$ score equal to 0, whereas two apps with completely identical resources have the $Jressim$ score equal to 1.

The tools utilizing resource-based repackaging detection are FSquaDRA [20], PlayDrone [17], and APPraiser [10]. FSquaDRA computes the resource-based similarity score ($Jressim$) and leverages the fact that hashes of all files are already included in the apk [20]. For identifying similar apps, APPraiser utilizes the same $Jressim$ score applied to included files (it computes MD5 hashes of the files and eliminates common libraries), but it is implemented at the market scale and more efficiently than FSquaDRA by leveraging the sparseness of data [10]. PlayDrone also applies the $Jressim$ score for detection of similar apps, and it includes resource file names as features alongside MD5 digests of the files themselves, and excludes common libraries from consideration [17]. PlayDroid operates at the Google Play market scale. Evaluation of the approach conducted with these tools was limited.

Indeed, the reported validation of the resource-based similarity approach is based on manual experiments with a limited number of apps [10,17,20], and the strong correlation discovered between the $Jressim$ score and the similarity score computed by the static analyzer `Androguard`, which measures the apk similarity score based on the included method signatures (i.e., the code) [5]. Thus, the strong correlation of resource-based scores with the code-based ones shown in [17,20] gives justification that the resource-based similarity detection approach is valid. Despite strong suggestions from the literature [10,15–17] and our personal communication with mobile security companies that the approach is applied in practice, the resource-based similarity detection method so far has not been validated on a sufficiently large dataset.

Moreover, without evaluation on the ground truth (a dataset with known repackaged and non-repackaged pairs), it is not possible to estimate a threshold (a value such that all pairs with a higher $Jressim$ score are reliably repackaged, and with a lower score are probably not repackaged) for the $Jressim$ score that can then be used by app markets in their triage. For the FSquaDRA tool the threshold value 0.7 was suggested, but [20] acknowledged that there was no way to confirm the threshold or adjust it without a repackaged dataset. The PlayDroid system applies the threshold value 0.8 and reports experiments with thresholds in range [0.6, 1.0]. (however, it includes resource file names as features in addition to the MD5 hashes of resource files, and excludes common libraries, so we cannot directly compare these threshold values) [17]. The APPraiser tool relies on the threshold value 0.8, and [10] reported that changing it to 0.7 or 0.9 did not affect the experiments significantly. At the same time, the *Jressim* value of 0.7 implies that 70 % of files are the same for two apks. Intuitively, much smaller fraction of identical resource files could already be a sign of repackaging.

Other Repackaging Detection Methods. State-of-art approaches in repackaging detection on Android have a strong focus on code similarity (e.g., [1,3–5,8,9,11,18,25]). To achieve scalability, tools leverage a combination of lightweight app fingerprints (e.g., certificates, package names, method signatures, n-grams of code) for identifying similar apps (e.g., [7,12]).

Recently, techniques that look at visual application similarity emerged. Differently from the resource-based repackaging detection approach evaluated in this paper, these techniques investigate layout files (e.g., [16]) and activity transition graphs (e.g., [15,19]) as means to represent the UI behaviour that is difficult to modify without a good understanding of the code. Among these techniques, DroidEagle [16] follows the same intuition as resource-based repackaging detection, and applies perceptual hashing to image files in order to detect similar pictures. It focuses on representing layout files as tree layout hashes and searching for similar layout structures.

ResDroid [15] utilizes resource files as features for detecting repackaged applications (e.g., it computes the average number of png files per folder in `res/drawable`). The MassVet system [2] follows a hybrid approach, as it relies on both similarity of UI structures and code similarity.

3 Dataset

We use a dataset of repackaged app pairs received from a fellow research group [11][4]. The dataset contains 2754 apps originally mapped into 1497 repackaged pairs. This dataset is representative of the piggybacking case: all app pairs in it include the original benign app and a repackaged version piggybacking malware (confirmed by VirusTotal[5]) [11]. Notice that for each repackaged app pair, its member apps are signed with different certificates.

As a first step to explore the obtained dataset, we applied the FSquaDRA tool [20] to perform pair-wise comparison of all files. In this experiment, we found that for 38 apps information about file hashes cannot be extracted by the tool. Among these 38 apps, 26 apps could not be installed, because they were not correctly signed: the whole `META-INF` folder was absent, or this folder was located in a wrong place (e.g., in the`assets` folder), or the signature file was missing). We received the error message `Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES]` during installation of these apps.

[4] https://github.com/serval-snt-uni-lu/Piggybacking.

[5] https://www.virustotal.com/.

As a security-aware app market would have discarded these apps anyway, we excluded these apks from consideration.

We were able to install the remaining 12 apps on a device and emulator. 10 of those were functional only on the device; and 2 failed to run on both device and emulator. The 12 apps are all malicious applications that install other apps in the background and show the same ads at startup. The FSquaDRA tool was unable to process them because they were misconfigured, so we excluded these apps from consideration. Overall, 126 repackaged app pairs were excluded from our dataset (for all 126 pairs at least one app was among the 38 non-processable ones). The remaining 1371 app pairs constitute for us the *truly repackaged pairs* dataset. To evaluate the false negative error rate of the approach, we have computed the $Jressim$ scores for app pairs in this dataset.

Moreover, in order to evaluate the false positive error rate of the approach, we created a set of *truly non-repackaged pairs* by randomly selecting 1371 app pairs from the dataset, excluding the broken 38 apps. We matched two apps together only if 3 conditions were satisfied: (1) their pair was not already considered as truly repackaged, (2) they did not belong to a connected component of repackaged apps (if apps a and b are a repackaged pair, and apps b and c are a repackaged pair, then apps a, b and c belong to the same connected component of repackaged apps), (3) they were signed with different certificates. We computed the $Jressim$ scores for the selected non-repackaged pairs. Thus, we obtained a balanced labelled dataset for further experiments consisting of 2742 app pairs of two kinds: repackaged and non-repackaged. Notice that similarly to the designers of the original dataset [11], in our experiments we focus on detecting plagiarism (malicious repackaging), not rebranding (repackaging by the same developer).

Table 1. Summary statistics for evaluation of resource-based repackaging results (the $Jressim$ scores) on the ground truth

Dataset	Statistics	Value
Truly repackaged app pairs (1371 pairs)	Min	0.0000
	1st Quartile	0.5050
	Median	0.7442
	3rd Quartile	0.9167
	Max	1.0000
	Mean	0.6893
Truly non-repackaged app (1371 pairs)	Min	0.0000
	1st Quartile	0.0000
	Median	0.0000
	3rd Quartile	0.0000
	Max	0.4218
	Mean	0.0022

4 Resource Similarity Evaluation

In this section we address the **RQ1** and empirically evaluate the baseline resource-based similarity detection approach (the *Jressim* score) on the ground truth.

Baseline Results. To answer **RQ1** we started by applying the resource-based repackaging detection method implemented by the open-source tool FSquaDRA [20] to our dataset. Table 1 reports the summary statistics for both repackaged and non-repackaged pairs and it reveals the shape of data. We can see that for the truly repackaged pairs the *Jressim* scores are quite high (with mean value 0.6893 and median 0.7442). At the same time, there are still repackaged app pairs that have 0.0 similarity score. For the truly non-repackaged app pairs the summary statistics are different. We see that more than 75 % of the *Jressim* scores in this case are equal to 0. At the same time, some app pairs in this dataset expose non-zero *Jressim* scores while being non-repackaged.

We can now compute the value (*threshold*) that minimizes the false positive error rate (the number of non-repackaged pairs that will be above the threshold) and false negative error rate (the number of repackaged pairs that will fall below the threshold) using the standard 10-fold cross-validation scheme. For Jaccard similarity, the average threshold in 10-fold cross validation on the dataset at hand is **0.0629**. This threshold might be further used with tools like FSquaDRA and APPraiser [10,20]. We report the accuracy, precision, recall and the F-measure for this threshold in Table 2(a).

At the same time, figures reported in Table 1 show that the baseline resource-based repackaging detection approach produces a number of outliers: some repackaged app pairs have low *Jressim* scores, while some non-repackaged app pairs have relatively high *Jressim* scores (significantly higher than the threshold 0.0629). We have looked at these outliers in order to understand the practical reasons for errors in the approach.

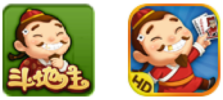
Table 2. Accuracy statistics

(a) *Jressim* accuracy metrics overview with the threshold 0.0629

Metrics	Value
Accuracy	0.9847
Precision	0.9912
Recall	0.9781
F-measure	0.9845

(b) `Androguard` metrics overview with the threshold 0.4330

Metrics	Value
Accuracy	0.9581
Precision	0.9764
Recall	0.9441
F-measure	0.9600



(a) (b)

Fig. 1. Icons of apps in a repackaged pair with the *Jressim* score 0.0

False Negatives. Repackaged app pairs with the $Jressim$ score less than 0.0629 are false negatives. There are 29 such pairs (out of 1371); all of them were present in the original list of repackaged pairs that came with the dataset [11].

Among the 29 false negatives, 7 pairs have the $Jressim$ score equal to 0.0. We have manually reviewed these apps. 4 pairs are visually similar, while 3 pairs are visually different (for 2 pairs even different functionality). 4 visually similar pairs have different hashes of the resource files, and the resource files have been substantially changed in the repackaged apps (new folders were introduced; images were substituted). Figure 1 gives an example of icons of a repackaged pair with $Jressim = 0.0$. As seen from the figure, the repackagers have produced a completely new icon that is still recognizable to a user.

For the other 22 pairs of repackaged apps with $Jressim$ greater than 0 but less than 0.0629, the apps in these pairs are visually similar. The large proportion of these are plagiarized apps translated into a different language. Repackagers of these apps changed substantially resource files, thus, the approach failed to classify them correctly.

False Positives. 11 non-repackaged pairs are false positives as they have the $Jressim$ score greater than the threshold 0.0629. We manually inspected these apps and checked which resources were shared between the apps in these pairs. We found that the false positives appeared in our results due to usage of the same libraries for app development. E.g., 3 app pairs were developed using the Facebook SDK[6]. These findings show that a prior filtering of resources is useful, as it will allow to considerably reduce the amount of false positives. Such pruning can be done automatically, e.g., by removal of the most popular file hashes in the whole dataset [10,17].

Comparison with Androguard. We applied `Androguard` [5][7] to our dataset to measure code-based similarity for both repackaged and non-repackaged pairs. The threshold that yields the lowest cumulative error for `Androguard` on our dataset is 0.4330. Table 2(b) summarizes the accuracy metrics for `Androguard` achieved with this threshold. Notice that the threshold value 0.4330 minimizes the cumulative error on the whole dataset. In the 10-fold cross-validation scheme the accuracy metrics will be lower.

Tables 2(a) and (b) indicate that the resource-based repackaging detection approach has better effectiveness than the code-based approach. Moreover, resource-based repackaging detection has a much better efficiency. Using FSquaDRA [20], we ran full pairwise comparison (comparing all app pairs for apps in the original dataset [11]) in 165 s (on a laptop with 2.8 GHz processor and 16 GB of RAM). `Androguard` required more than 10 h only for the truly repackaged and non-repackaged pairs. `Androguard` is inherently slow on non-repackaged pairs [5], which in practice constitute the vast majority [10].

[6] Facebook SDK for Android https://developers.facebook.com/docs/android.
[7] https://github.com/androguard/androguard.

5 Fine-Tuning the Basic Approach

We now analyze how to improve the predictive power of the basic resource-based repackaging detection approach. In particular, we explore the questions **RQ2** and **RQ3** regarding the most suitable similarity metrics and classifier with the best discriminative power.

To answer these questions we used our dataset and machine learning approaches. For machine learning tasks we used the scikit-learn library, version 0.17.1 [13]. Additionally to the provided algorithms, we also developed a basic classifier separating two classes using a threshold obtained by minimizing the cumulative error (as reported in Sect. 4); to avoid over-fitting this classifier was further applied only in the 10-fold cross-validation setting.

Exploring Similarity Metrics. In [10,20] the Jaccard similarity metric was used for the full sets of hashes of resource files in a given pair of apps. In general, any similarity score applied to sets (multisets) of resource file hashes shows to which extent one application is similar to another. In this section we explore if usage of another metrics can improve the discriminative power of the method. To achieve this goal, we extended the open-source FSquaDRA tool [20] with the possibility to calculate similarity scores using different metrics. In particular, we took as a reference the `SimMetrics` Java library[8] and implemented in Python the metrics that compare lists of objects (sets/multisets). In particular, we implemented the following metrics:

Block (Manhattan) Similarity: $similarity(a,b) = 1 - distance(a,b)/(|a| + |b|)$, where $distance(a,b) = ||a - b||_1$ (distance from point a to point b is the sum of of absolute differences of their Cartesian coordinates).

Cosine Similarity: $similarity(a,b) = a \cdot b \ / \ (||a|| \times ||b||)$, i.e., it measures $\cos \phi$ of the angle ϕ between vectors a and b. Cosine similarity considers cardinality of elements (number of occurrences).

Sørensen-Dice Similarity: $similarity(a,b) = 2 \times |a \cap b|/(|a| + |b|)$.

Euclidean Similarity: $similarity(a,b) = 1 - distance(a,b)/\sqrt{|a|^2 + |b|^2})$, where $distance(a,b) = ||a - b||$ (distance from point a to point b is Euclidean norm of the vector $a - b$).

Jaccard Similarity: $similarity(a,b) = |a \cap b|/|a \cup b|$. This is the *Jressim* score used in [10,20].

Generalized Jaccard Similarity: follows the same formula as Jaccard similarity, but works over multisets.

Overlap Similarity: $similarity(a,b) = |a \cap b|/min(|a|,|b|)$.

Generalized Overlap Similarity: same as the overlap similarity, but works over multisets.

[8] https://github.com/Simmetrics.

Table 3. Predictive power comparison of similarity metrics

Metric	Accuracy	Precision	Recall	F-measure
Block	0.9832	0.9891	0.9774	0.9831
Cosine	0.9832	0.9883	0.9781	0.9831
Dice	0.9836	0.9898	0.9774	0.9835
Euclidian	0.7400	0.9020	0.5383	0.6733
Jaccard	0.9847	**0.9912**	0.9781	0.9845
Generalized Jaccard	0.9836	0.9898	0.9774	0.9835
Generalized overlap	0.9840	0.9855	0.9825	0.9840
Overlap	**0.9847**	0.9856	**0.9840**	**0.9847**
SimonWhite	0.9836	0.9898	0.9774	0.9835
Tanimoto	0.9829	0.9891	0.9767	0.9827

Simon White Similarity: the generalized (quantitative) Sørensen-Dice similarity, else called Simon-White coefficient, works over multisets and considers cardinality of elements.

Tanimoto Similarity: is expressed using the cosine similarity formula, but multiple occurrences of elements are not considered (as it works over sets).

We calculated these metrics for all app pairs in our dataset. For each metric, the average results in 10-fold cross-validation for the basic classifier that discriminates based on the similarity score is reported in Table 3 (same folds partition was applied for all metrics). The result demonstrate that the Overlap similarity metric has better accuracy, recall and the F-measure, while the Jaccard similarity metric shows better precision. The F-measure, which harmonically combines both precision and recall, indicates that generally Overlap similarity is preferable for the repackaging classification task with our dataset. Both previous studies on resource-based similarity detection [10,20] relied on the Jaccard similarity score, however, experiments show that the Overlap metric can achieve better results. Therefore, in the rest of this paper we rely on this similarity metric.

For the Overlap similarity score (*Oressim* for short) the average *threshold* in 10-fold cross-validation is **0.1188**. This threshold should be further used in resource-based repackaging detection approach with the Overlap similarity score.

Applying Classifiers. In order to answer **RQ3** and assess which classifiers have the best discriminative power in our task, we experimented with 5 general-purpose classifiers: Logistic Regression, Support Vector Machines with a linear kernel, Decision Tree, Random Forest and Gradient Boosting. In this work we used the default values of the algorithm parameters, i.e., we skipped the parameter tuning step. Moreover, these classifiers were instantiated with the same initial state to ensure experiment replicability.

We applied the selected classifiers, including our own, to the dataset using only one feature – the Overlap similarity score *Oressim* calculated on all file

Table 4. Predictive power comparison of classifiers

Classifier	Accuracy	Precision	Recall	F-measure
OurClassifier	**0.9847**	0.9855	**0.9840**	**0.9847**
Logistic Regression	0.9799	0.**9941**	0.9657	0.9796
Linear SVM	0.9814	0.9904	0.9723	0.9812
Decision Tree	0.9756	0.9776	0.9737	0.9755
Random Forest	0.9763	0.9784	0.9745	0.9762
Gradient Boosting	0.9799	0.9840	0.9759	0.9798

hashes. Every classifier was validated in 10-fold cross-validation. Table 4 reports the average results for all classifiers. We can see that the cumulative error minimization classifier (OurClassifier) performs well and shows the best scores for accuracy, recall and the F-measure. In terms of precision, the Logistic Regression classifier has shown better results. Thus, in the task of resource-based repackaging detection with a single feature (the Overlap similarity score for all files), the algorithm classifying app pairs based on the score threshold separating classes of repackaged and non-repackaged app pairs can be used. At the same time, this algorithm cannot be generalized when several features are used for classification.

6 Resource Files Analysis and Improved Classification

We now address the research questions **RQ4** and **RQ5** concerning the exploration of resource file types for fine-tuning repackaging detection.

Exploring Resource File Types. An app package includes different types of files. It is still an open question whether repackagers generally modify all types of files, or only some specific types. Therefore, while previously a cumulative similarity score on all files was used, it might be beneficial to explore different types of resources separately. We now start to explore **RQ4** by dissecting the files constituting Android packages into different types.

Android documentation specifies resource types that can be included in a package[9]. However, during compilation some of them are compiled and placed into the compiled resource file `resources.arsc`. As the `MANIFEST.MF` file only stores paths to files and the hashes of their contents, it is impossible to compute separate similarity scores for these compiled resources. Operating with the `MANIFEST.MF` information, we can only divide files into types based on the common path prefixes and suffixes. In particular, we divide files based on their purpose (e.g., audio-video files) and their location (for instance, under the `res/` folder). The following resource types (features) are identified:

[9] http://developer.android.com/guide/topics/resources/available-resources.html.

Location-based file division:

1. `manifest`: the manifest file (`AndroidManifest.xml`)
2. `main_code`: main files with the compiled Android code (`classesN.dex`, where N is either empty or integer number)
3. `resources_arsc`: compiled resources file (`resources.arsc`)
4. `libs`: files located under `lib/` and `libs` folders
5. `assets`: files under `assets/` directory
6. `res_all`: all files located under `res/` folder
7. `res_raw`: files under `res/raw/` directory
8. `res_xml`: files located under `res/xml/` directory
9. `res_drawable`: files under `res/drawables/` folder
10. `res_menu`: files located under `res/menu/` directory
11. `res_layout`: layout files under `res/layout/` directory
12. `res_anim`: files under `res/anim/` folder
13. `res_color`: files in `res/color/` directory

Purpose-based file division:

14. `native_libs`: all files with `.so` extension
15. `code_general`: all files with `.so`, `.bin`, `.dex` and `.jar` extensions
16. `audio_video`: supported audio and video files[10]
17. `image`: all supported image files
18. `all_xml`: all XML files

We extracted the hashes corresponding to the considered types from both packages in an app pair under consideration, and calculated the Overlap similarity score *Oressim* separately for every file type. Thus, for each pair we obtained a feature vector with 18 values. The features are not independent: e.g., `code_general` includes `native_libs` and `main_code`. However, since the classifiers we chose do not assume feature independence, in contrast to e.g., Naïve Bayes, we did not put any restrictions on them. After extracting features, we applied the selected classifiers, excluding our own as it could not be generalized for multiple features, to the dataset using 10-fold cross-validation.

To discover the optimal set of features that drives better results, we applied the sequential forward selection (SFS) algorithm [14]. This algorithm starts with a single feature and sequentially adds the variables with which the classifier shows the highest score, until the size of the feature space specified by the user is reached. Once a feature is added, it cannot be removed from the search space. In our case, we selected the total number of features (18) as the limit. This approach reports not only if the classification can be improved by considering separately different types of files, but also if we can reduce the set of features to be extracted from files, thus, saving time for the feature extraction.

Some apps may lack files of a particular type. The conventional approach in this case is to assign the similarity score equal to 0 (e.g., this is how the

[10] http://developer.android.com/guide/appendix/media-formats.html.

SimMetrics library works). However, we also experimented with assigning to such cases a value that is out of the range (−1) to distinguish them.

Table 5 reports the F-measure for all classifiers (the accuracy score shows similar behavior, thus, we do not provide it here). Several important conclusions can be drawn from the results. First, it is now evident that if file types are considered as features, the effectiveness of repackaging detection can be improved. The Random Forest classifier with a combination of 12 features achieves the F-measure score of 0.9919, which is considerably better than our classifier operating on the cumulative similarity score (the F-measure 0.9847). Considering only 2 types of resources (when N/A's are substituted to −1), the Random Forest classifier already outperforms our classifier that minimizes the cumulative error.

Secondly, the classifiers behave differently depending on how the N/A values are treated. When we substitute N/A values to 0, Linear SVM and Logistic Regression classifiers show almost similar scores, outperforming the score of our custom classifier. At the same time, when N/A are equal to −1, even using a number of features, those classifiers cannot beat our classifier trained only on a single feature. At the same time, the Decision Tree algorithm shows the opposite behavior improving its score when N/A values are substituted to −1. This shows that classifiers used in resource-based repackaging detection systems should be selected considering, among all factors, how the N/A values are treated.

Last but not least, Table 5 demonstrates that generally the scores achieved by the Random Forest and Gradient Boosting algorithms in case when N/A values are substituted to −1, are better than using the conventional approach

Table 5. F-measure dependency on the number of features for the classifiers. Values highlighted with **bold** shows best result within a column, with *red* – best result in a row.

Features number	Logistic regression		Linear SVM		Decision tree		Random forest		Gradient boosting	
	n/a = 0	n/a = −1	n/a = 0	n/a = −1	n/a = 0	n/a = −1	n/a = 0	n/a = −1	n/a = 0	n/a = −1
1	0.9733	0.9733	0.9734	0.9734	0.9700	0.9700	0.9712	0.9712	0.9745	0.9745
2	0.9842	0.9834	0.9842	0.9820	0.9803	0.9879	0.9792	0.9872	0.9839	0.9868
3	0.9857	0.9838	0.9857	0.9824	0.9813	0.9883	0.9821	0.9886	0.9857	0.9883
4	0.9860	0.9842	0.9868	0.9831	0.9832	**0.9883**	0.9850	0.9897	0.9861	0.9901
5	0.9868	0.9842	0.9872	0.9835	0.9832	0.9883	0.9853	0.9908	0.9861	0.9901
6	0.9871	0.9842	0.9872	0.9838	**0.9835**	0.9883	0.9868	0.9904	0.9857	**0.9901**
7	0.9871	0.9842	0.9872	0.9842	0.9832	0.9879	0.9882	0.9919	0.9861	0.9901
8	0.9871	0.9842	0.9872	0.9846	0.9831	0.9875	0.9886	0.9908	0.9868	0.9901
9	0.9871	0.9842	0.9872	0.9846	0.9835	0.9872	0.9872	0.9901	0.9868	0.9897
10	0.9871	**0.9842**	**0.9872**	**0.9846**	0.9832	0.9875	0.9879	0.9905	**0.9868**	0.9897
11	0.9875	0.9842	0.9872	0.9846	0.9832	0.9872	0.9875	0.9908	0.9868	0.9897
12	0.9875	0.9842	0.9872	0.9846	0.9824	0.9872	0.9875	**0.9919**	0.9865	0.9894
13	**0.9875**	0.9842	0.9872	0.9842	0.9824	0.9875	**0.9890**	0.9919	0.9861	0.9897
14	0.9875	0.9842	0.9868	0.9846	0.9824	0.9861	0.9875	0.9919	0.9868	0.9894
15	0.9875	0.9842	0.9868	0.9846	0.9824	0.9857	0.9879	0.9912	0.9858	0.9894
16	0.9875	0.9834	0.9868	0.9842	0.9828	0.9850	0.9872	0.9908	0.9850	0.9890
17	0.9864	0.9831	0.9864	0.9839	0.9820	0.9846	0.9872	0.9908	0.9857	0.9886
18	0.9868	0.9827	0.9864	0.9835	0.9817	0.9858	0.9868	0.9901	0.9846	0.9883

(substitution to 0). Obviously, the absence of files of a particular type is also a feature and thus, it can improve the predictive power of a classifier, as we observed in our experiments.

Susceptibility of File Types to Modification in Repackaging. One of the main questions we would like to answer in this paper is **RQ5**. Obviously, to further improve the method for app plagiarism detection based on resource files, it is important to know which resources are more frequently modified during the repackaging process. To answer this question we performed the following experiment. For every type of files (present in both packages from an app pair) we calculated the average similarity score using only repackaged pairs from our dataset. File types with higher such scores are *less frequently modified* in repackaging. Obviously, if in a repackaged app pair the similarity score for some file type is high, then such files were mostly not modified. The results of this experiment are presented in Table 6(a).

Table 6(a) suggests that multimedia, raw, images, libraries in general, and native libraries are less frequently changed in the repackaging process. Several important conclusions can be drawn from this fact. First, despite the fact that in the recent years several methods were proposed to detect repackaged apps using resource similarity, it seems that adversaries still do not consider them as a threat to their business. Thus, the resource files that are not required to be changed in repackaging are mostly left untouched. Secondly, the mentioned file types are more difficult to change. Clearly, without special tools it is quite difficult to edit multimedia files or native libraries. That can also explain why these file types are mostly left in the original state. These considerations can improve the resiliency of approaches based on resources similarity comparison.

Table 6. Results for different file types in repackaging (on average)

(a) Files modified in repackaging (1 – files never modified; 0 – always modified)

Average Score	**Feature**
0.9788	audio_video
0.9574	res_raw
0.9269	images
0.9229	libs
0.9199	native_libs
0.9177	assets
0.8202	res_drawable
0.7648	res_all
0.4840	code_general
0.3679	res_xml
0.3503	res_anim
0.3273	res_menu
0.3077	all_xml
0.2802	res_color
0.2557	res_layout
0.1524	resources_arsc
0.0934	manifest
0.0773	main_code

(b) Same files in non-repackaged pairs (1 – always the same; 0 – never the same)

Average Score	**Feature**
0.0159	res_drawable
0.0130	images
0.0118	res_all
0.0094	libs
0.0087	native_libs
0.0057	res_color
0.0045	code_general
0.0036	res_raw
0.0032	assets
0.0030	all_xml
0.0018	res_layout
0.0	audio_video
0.0	manifest
0.0	res_anim
0.0	resources_arsc
0.0	res_xml
0.0	main_code
0.0	res_menu

According to our analysis, dex files, the Android manifest and the compiled resource files are changed quite often. This observation perfectly agrees with the repackaging process logic for the piggybacking scenario, which is the case for our original dataset [11]. If adversaries want to add some malicious functionality, they change the dex and Android manifest files to add necessary permissions [22]. There are many tools that can do this automatically for malicious and benign purposes (e.g., for instrumentation in testing [24]). The file containing compiled resources is also often modified in repackaging. This file incorporates information about string resources included in an apk. Evidently, if adversaries repackage an apk with the purpose of publishing it in other national markets, they translate the application. Therefore, an additional locale should be added to string resources, resulting in the compiled resources file change. Secondly, the adversaries often change ads IDs, which are usually defined within the string resources.

Table 6(b) reports similarity scores for the different resource files types for the non-repackaged dataset. This information is also instructive, as it allows to analyze better the false positives discussed in Sect. 4. Indeed, we see that developers reuse images (e.g., "like" buttons) and libraries across different apps.

We should mention that besides modification of the files, the score changes if files of that type are added or removed. Currently, we do not isolate these cases.

7 Discussion

Threats to Validity. We evaluated the resource-based repackaging detection approach on one dataset [11], which was originally collected under assumption of a lazy adversary who does not change a lot during repackaging. On more diverse datasets effectiveness of resource-based similarity detection approaches can be different, and it needs to be further investigated.

The resource-based similarity detection approach is currently not robust against an attacker who takes care to slightly change all files included in the original apk. This is not a challenging task, and minor edits can be automated. However, as we have mentioned in Sect. 4, most of repackaged pairs we failed to detect were either not visually similar, or visually similar to a user, but not to a machine. As Fig. 1 shows, repackagers can be very creative in modifying apps so that the main theme is recognizable, while the included images are very different. Advance AI techniques can be applied to detect such apps, but these techniques are currently not scalable to the on-market setting.

Yet, though the approach is very accurate on our dataset, which is quite recent, its robustness against a casual attacker (who just changes the files a little to modify the hashes) still can be improved, by, e.g., working with perceptual hashes or fuzzy hashes of the files (not the hashes included in the original package). Better understanding which file types are generally not modified in repackaging (e.g., multimedia, libraries and images) suggests that we can apply fuzzy hashing techniques only to those types of files.

Efficiency. We have operated in our experiments in a setting with a full pairwise comparison. This is not actually a scenario applicable to day-to-day app markets operations. Instead, in a practical set-up, a market expects to compare a relatively small set of recently submitted apps to a very large set of already known apps [10]. For a set-up like this, an app market can maintain a database with sorted hashes of already known files (e.g., in a tree-structure) that is efficiently searchable for hashes from the new apps.

8 Conclusions

In this paper we have practically evaluated the resource-based repackaging detection approach. Our experiments show that this technique is very effective with outstanding results for accuracy, precision, recall and the F-measure. Furthermore, we improve the existing tools [10,17,20] by suggesting the Overlap similarity metric to be used with the Random Forest classifier on 12 features. We have also reported which resource file types are less prone to modification in repackaging, and which resource files can coincide in non-repackaged pairs. Our results may be instructive for researchers and practitioners looking into adding the resource-based repackaging detection approach to their app triage schemes.

References

1. Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proceedings of ICSE. IEEE/ACM (2014)
2. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., Liu, P.: Finding unknown malice in 10 s: mass vetting for new threats at the Google-Play scale. In: Proceedings of USENIX Security Symposium (2015)
3. 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). doi:10.1007/978-3-642-33167-1_3
4. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar Android applications. In: Proceedings of ESORICS (2013)
5. Desnos, A.: Android: static analysis using similarity distance. In: Proceedings of HICSS 2012, pp. 5394–5403 (2012)
6. Gadyatskaya, O., Massacci, F., Zhauniarovich, Y.: Security in the Firefox OS and Tizen mobile platforms. IEEE Comput. **47**(6), 57–63 (2014)
7. Gonzalez, H., Kadir, A., Stackanova, N., Alzahrani, A., Ghorbani, A.: Exploring reverse engineering symptoms in Android apps. In: Proceedings of EuroSec. ACM (2015)
8. Guan, Q., Huang, H., Luo, W., Zhu, S.: Semantics-based repackaging detection for mobile apps. In: Caballero, J., Bodden, E., Athanasopoulos, E. (eds.) ESSoS 2016. LNCS, vol. 9639, pp. 89–105. Springer, Heidelberg (2016). doi:10.1007/978-3-319-30806-7_6
9. 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). doi:10.1007/978-3-642-37300-8_4

10. Ishii, Y., Watanabe, T., Akiyama, M., Mori, T.: Clone or relative? Understanding the originals of similar Android apps. In: Proceedings of IWSPA. ACM (2016)
11. Li, L., Li, D., Bissyandé, T.F., Lo, D., Klein, J., Le Traon, Y.: Ungrafting malicious code from piggybacked Android apps. Technical report, SnT, University of Luxembourg (2016)
12. Lindorfer, M., Volanis, S., Sisto, A., Neugschwandtner, M., Athanasopoulos, E., Maggi, F., Platzer, C., Zanero, S., Ioannidis, S.: AndRadar: fast discovery of Android applications in alternative markets. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 51–71. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08509-8_4
13. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
14. Saeys, Y., Inza, I., Larrañaga, P.: A review of feature selection techniques in bioinformatics. Bioinformatics **23**(19), 2507–2517 (2007)
15. Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L.: Towards a scalable resource-driven approach for detecting repackaged Android applications. In: Proceedings of ACSAC. ACM (2014)
16. Sun, M., Li, M., Lui, J.: DroidEagle: seamless detection of visually similar Android apps. In: Proceedings of WiSec. ACM (2015)
17. Viennot, N., Garcia, E., Nieh, J.: A measurement study of Google Play. In: Proceedings of SIGMETRICS. ACM (2014)
18. Wang, H., Guo, Y., Ma, Z., Chen, X.: WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: Proceedings of ISSTA. ACM (2015)
19. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of WiSec. ACM (2014)
20. Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., La Spina, F., Moser, E.: FSquaDRA: fast detection of repackaged applications. In: Atluri, V., Pernul, G. (eds.) DBSec 2014. LNCS, vol. 8566, pp. 130–145. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43936-4_9
21. Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: StaDynA: addressing the problem of dynamic code updates in the security analysis of Android applications. In: Proceedings of CODASPY (2015)
22. Zhauniarovich, Y., Gadyatskaya, O.: Small changes, big changes: an updated view on the Android permission system. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 346–367. Springer International Publishing, Switzerland (2016). doi:10.1007/978-3-319-45719-2_16
23. Zhauniarovich, Y., Gadyatskaya, O., Crispo, B.: Demo: enabling trusted stores for Android. In: Proceedings of CCS, pp. 1345–1348. ACM (2013)
24. Zhauniarovich, Y., Philippov, A., Gadyatskaya, O., Crispo, B., Massacci, F.: Towards black box testing of Android apps. In: Proceedings of Software Assurance Workshop at ARES, pp. 501–510 (2015)
25. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of CODASPY (2012)
26. Zhou, Y., Jiang, X.: Dissecting Android malware: characterization and evolution. In: Proceedings of S&P. IEEE (2012)