View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Royal Holloway - Pure

You Can't Touch This: Consumer-Centric Android Application Repackaging Detection

Iakovos Gurulian*, Konstantinos Markantonakis, Lorenzo Cavallaro, Keith Mayes

Information Security Group, Royal Holloway, University of London, Egham Hill, Egham, Surrey TW20 0EX, United Kingdom

Abstract

Application repackaging is a widely used method for malware distribution, revenue stealing and piracy. Repackaged applications are modified versions of original applications, that can potentially target large audiences based on the original application's popularity. In this paper, we propose an approach for detecting repackaged applications. Our approach takes advantage of the attacker's reluctance to significantly alter the elements that characterise an application without notably impacting the application's distribution. These elements include the application's name and icon. The detection is initiated from the client side, prior to an application's installation, making it application store agnostic. Our experimental results show that detection based on our algorithm is effective and efficient.

Keywords:
Android, application repackaging, user-centric security, user privacy, effectiveness analysis, electronic fraud

1. Introduction

One of the major challenges for a malicious user is to get a malicious application distributed to a substantial population of genuine users. On the Android platform, 86% of all malware distribution relies on *repackaged applications* [22]. In the context of this paper, we define repackaged applications

*Corresponding author
Email address: `Iakovos.Gurulian.2014@live.rhul.ac.uk` (Iakovos Gurulian)

as applications that impersonate a genuine application, by slight modifications/variations to the genuine application's artwork and/or changes to its source code in a way that the repackaged application looks and/or feels like the genuine application. The main objective of a repackaged application is to mimic a genuine application so it can target novice users that gravitate towards the popularity/functionality of the genuine application. In this definition, we do not include applications that infringe the potential intellectual property of the original application and present themselves as unique/different applications.

Repackaged applications are a serious threat and are part of the OWASP's *Top Ten Mobile Risks* for 2014 [13], posing the first and only threat in the list related to malware distribution. Many different methods for detection of repackaged applications have been proposed, but most of them rely on the application stores to perform the detection [8, 5, 21, 9, 19, 18, 6, 15] *(Section 2.2)*.

In this paper we propose a method for detecting repackaged applications, initiated on the client side, prior to an application's installation. The target application is being checked against a database of legitimate applications, hosted by a trusted third party (TTP). If it does not exist in the database, string and image similarity algorithms are used to detect original applications with similar name and icon pairs *(Section 3)*. These two elements characterise an application prior to its installation. An attacker who wants to increase the spreading rate of a repackaged application by taking advantage of the already established popularity of an original application might not be likely to alter these elements.

Our experimental results have shown that the proposed mechanism is both effective and efficient in detecting repackaged applications *(Section 5)*. Furthermore, in comparison to existing methods, our proposal scored high on a set of predefined criteria *(Section 5.3)*.

The main contributions of this work are:

- A method for detecting repackaged applications, based on elements that an attacker cannot significantly alter without substantially minimising the attack potential. The method was designed to be fast and application store agnostic, so that the detection process can be initiated from the client side, prior to an installation.

- We were capable of detecting applications that only copy the name and the icon of an original application in order to trick the users into installing them. This is a known technique that attackers use [4], but to our knowledge, we are the first to effectively detect such applications.

2. Application Repackaging

Android applications come in `.apk` containers (basically `.zip` files). These containers include the application's bytecode, resources and libraries, as well as a folder (named `META-INF`) that holds the signature(s), generated by the developer, on different elements of the respective application. An attacker that repackages an application can modify its bytecode, alter its resources and libraries, and then remove the `META-INF` folder and sign the application with his/her own key.

Each Android application has a package name that is used by the operating system as the differentiator factor between applications. If an application that is about to be installed shares the same package name with an already installed one, Android will perceive it as an update attempt on the last. The update process cannot continue, unless the signatures of the two applications match.

2.1. Threats to the Android Ecosystem

The threats posed by application repackaging can be separated into those concerning the application developer and those concerning the user.

2.1.1. Threats to the developer.

Developers are given the opportunity to gain financial profit through their applications. Repackaged applications pose a threat to them in various ways.

- *Unauthorised redistribution:* An attacker can redistribute an application, signed with his/her own signature and possibly sell it in an application store, without the original developer's permission.

- *Advertisements:* Potential revenue from an application might be redirected to the attacker, as he/she might have altered the developer's account with his/her account, thus receiving the advertisement revenue.

- *Cracking (Piracy):* Application repackaging might distribute pirated copies of a genuine paid application, by bypassing any implemented verification and validation mechanisms.

2.1.2. Threats to the user.

Repackaged applications that aim to harm the user can achieve their goal in two different ways.

- *Trojan horse:* A repackaged application could act as a Trojan horse. Malicious code can be implanted in the original application, capable of intruding the platform's security and the user's privacy. For example, repackaging an application that requires permissions to access the Internet and the device's storage can allow an attacker to steal the user's images.

- *Denial of upgrade:* A repackaged application will not be updated by a genuine application's update. Therefore, the user will not be able to upgrade, once a repackaged application has been installed.

2.2. Related works

AndroGuard was proposed by Desnos and Gueguen [8]. Their proposal is based on Control Flow Graphs, used to measure the similarity between applications.

Crussell et al. proposed *DNADroid* that uses Program Dependency Graphs to detect similar applications [5]. The authors claim that their system produces a low false positive rate. However, they accept that advanced obfuscation techniques can go undetected. To increase the system's performance, they only compare the application in question against applications with similar names.

Zhou et al. used Fuzzy Hashing for repackaging detection [21]. This method is based on generating a hash of the application by breaking it down into small chunks and combining their hashes. They also remove string operands that can easily be altered from the instructions prior to hashing, in order to prevent common obfuscation techniques. They have created an application called *DroidMOSS*. They state that although their system is very robust, detection may fail if big chunks of code have been added to the original application.

Another solution, proposed by Hanna et al., introduces the idea of Feature Hashing for the detection of similar applications [9]. For this purpose, a tool called *Juxtapp* was created, that according to the authors is resilient to some amount of obfuscation.

Zhauniarovich et al. [19] proposed detection of repackaged applications based on the contents of the `.apk` file. Their method showed results similar to those of techniques that involve code analysis. SHA1 is used for the comparison of all the files between different applications. A repackaged application with slight changes on many files might go undetected. The authors propose a combination of their method with code analysis in order to overcome this issue.

A framework capable of measuring the obfuscation resilience of algorithms used in repackaging detection programs was also proposed in 2013 by Huang et al. [10]. Other researchers have also proposed methods for detection of repackaged applications as well, or have investigated the subject [18, 6, 15].

Due to the computational complexity of the methods discussed above, they are only meaningful on an application store level. For this reason, Zhou et al. [20] proposed client side initiated repackaging detection, with *AppInk*. Their method uses application watermarking in order to confirm the authenticity of an application. In order to achieve that, a few additional steps have to be taken by the developer in order to embed the watermark.

3. Proposed Solution

We propose a repackaging detection technique that takes advantage of the attacker's reluctance to significantly alter elements that characterise an application, without substantially minimising the attack vector. The data that characterises an application prior to its installation are its *name* and *icon*. Since there is a high probability that little or no modification is likely to have taken place on these elements of a repackaged application, image and string comparison algorithms can be used in order to determine the potential original application by comparing them against a database of authentic applications.

This technique is initiated from the client side, prior to the installation of an application. Related data from the application is transferred to the TTP's server that maintains a database of original and trusted applications, as well as a blacklist of malware and repackaged applications. Based on the analysis, the TTP will either verify that the application in question is genuine, or return a list of genuine applications it may be trying to masquerade.

In this work we assume that applications kept in the TTP's database have been sanitised and do not contain malicious code. We assume that *developer-application* pairs in the database have been confirmed and no repackaged applications exist. We are using applications from the Google Play Store for the population of the trusted database. Studies have found that 1.2% of all applications of the Play Store are repackaged [2], so it is reasonable to consider it as relatively trusted for the purposes of this work. However, in a real world scenario, building the TTP's database is a challenging task, but the construction methodology is out of the scope of this work.

3.1. Threat Model

The capabilities of an attacker for the scope of this paper are as follows:

- An attacker can access any legitimate Android application. This includes being able to decompile, modify, recompile, or copy and use elements (including, but not limited to media files) of applications.
- An attacker cannot alter the name and the icon that characterise an application to an extent, without going unnoticed by potential victims.
- An attacker can distribute applications from any channel, except the Google Play Store. In the scope of this paper we consider the Google Play Store as the trusted entity.
- An attacker cannot sign applications using the original developer's signing key.
- An attacker cannot influence the TTP's database.
- An attacker cannot forcefully install an application or have access to a victim's device.

3.2. Assumptions

- We assume that applications kept in the trusted party's database have gone through thorough analysis and do not contain malicious code.
- We assume that *developer-application* pairs in the database have been confirmed and no repackaged applications exist.

The Google Play Store contains only a small amount of repackaged applications (1.2%) [2] and is considered as trusted for the purposes of this research. Although detection details are not publicly available, the use of repackaging detection mechanisms in combination with users being able to report applications, aid towards building a safer ecosystem. This does not degrade the importance of our proposed solution, since Android users are not restricted to downloading applications from the Google Play Store, and the solution is application store agnostic.

3.3. Requirements

The requirements that should be met in order to ensure the feasibility of the proposed solution are the following:

- The process should be fast. A user who is installing an application is not likely to wait for more than a few extra seconds for the process to finish.
- Using a remote server to assist the detection process should use as little bandwidth as possible.
- The detection process should not have a noticeable impact on the user device's performance. The method should be integrated to the package installer and run prior to an application's installation.

3.4. Proposed solution's overview

A high level overview of the proposed solution would be as follows:

1. On the client side, when the user has selected to install an application, prior to the initiation of the installation, data required for the detection process is extracted from the `.apk` file and transferred to the TTP's server through a secure channel (using SSL).
2. The TTP's server, after receiving the data from the device, processes it and takes a decision whether the target application is safe, unknown to its database, potentially repackaged, or repackaged/malicious. The decision is then returned to the client.
3. According to the returned result, the client device will either continue by installing the application without any warnings (in case it is classified as safe), or warn the user that it is unknown, or potentially malicious. In case it is classified as potentially repackaged, a list of possible original applications is also presented to the user, with links to safe ways of obtaining them.

3.5. The detection process

The TTP's server requires *five* elements for the detection process. These elements are extracted from the target `.apk` file, on the client side, prior to installation, and are transmitted to the server over a secure channel. The five elements are:

1. The hash of the `.apk` container (referred as *"signed .apk"*). It is used to speed up the process, in case the application being tested is legitimate (exists in the trusted applications database).

2. The hash of the `.apk` container, excluding the `META-INF` folder (referred as *"unsigned .apk"*). It is used to detect blacklisted applications, regardless of the developer's signature, or applications that are legitimate, but have been signed with a different developer signature.

3. The hash of the developer's signature. It is used to determine whether a potentially repackaged application could be a version of a legitimate application that is not maintained in the TTP's database. In such case, the application being tested and the potentially legitimate application should be sharing the same developer signature.

4. The application's name.

5. The application's icon. In case the application being tested does not exist in the trusted application database, the application name and icon are used for detection of visually similar applications.

A pre-computed list of these elements for all trusted applications is stored in the TTP's database, against which the received data is checked. The names and icons of older versions of an application are also kept, in case they have changed.

Algorithm 1 explains the process followed by the server in order to check the originality of an application. In the algorithm, the function *blacklisted(String unsignedHash)* queries the blacklist database and returns true if the input unsigned hash exists. This first step increases the overall performance of the system by quickly detecting known malicious applications. *signedHashExists(String signedHash)* and *unsignedHashExists(String unsignedHash)* query the trusted database for matches and return true if the signed or unsigned application's hash exist, respectively. The function *getOriginalApp(unsignedHash)* returns the original application that is linked to that

Algorithm 1: Repackaged application check

```
Input: String signedHash, String unsignedHash, String appName, Image appIcon
/* Check the blacklist database for an unsigned hash match                 */
if blacklisted(unsignedHash) then
    return getOriginalApp($unsignedHash$)
/* Check the applications database for a signed hash match                 */
else if signedHashExists(signedHash) then
    return applicationSafe()
/* Check the applications database for an unsigned hash match              */
else if unsignedHashExists(unsignedHash) then
    return getOriginalApp($unsignedHash$)
/* Find similar applications based on their name and icon                  */
else
    return getSimilarApps($appName, appIcon, devSignature$)
```

hash, if such exists. *applicationSafe()* returns a code that denotes that the application is legitimate. However, if no trusted application is found, *getSimilarApps(String name, Image icon, String devSignature)* returns an array of trusted applications that are similar to the input, based on their name and icon. The detection process is described in *Section 3.6.* In case an application does not exist in the database, but is found to be similar to another, it may be an updated (or older) version. For this reason, the hashes of the developer signatures of the tested application are compared and if they match, the server notifies that it may be a different version of a certain legitimate application. A developer should use the same signatures for different versions of an application, otherwise the operating system will refuse to complete the update process.

The result is finally returned to the user's device. If the application does not exist in the database, but similar applications are detected, a list of these is presented to the user. If the application is found to be legitimate, the installation process will continue without any further warnings. If the application is found to be blacklisted, signed with a different signature than the original, or not found in the trusted database, the user is warned.

Many users might still choose to install pirated applications. Usually no code modifications are required for an application to be pirated and can be distributed without modifications, including modifications on the signature. Any application with signature or code modifications is treated as non-trusted by the proposed solution. Threats from such applications have been discussed in *Section 2.1.*

Table 1. Jaro-Winkler and Levenshtein distance top 10 results based on input *"googl app stoy"*

Results	**Jaro-Winkler**		**Levenshtein**	
	Rank	**Similarity**	**Rank**	**Similarity**
Google Play Store	1	0.904008	1	7
Google Earth	2	0.873016	3	8
Google Classroom	3	0.86756	2	8
Google Fit	4	0.866667	5	8
Google Sheets	5	0.864469	6	8
Google Docs	6	0.862284	4	8
Google Apps Device Policy	7	0.855801	-	-
Google Translate	8	0.839782	-	-
Google Slides	9	0.839744	-	-
Google Goggles	10	0.83631	8	9
GoPro App	-	-	7	8
Google Finance	-	-	9	9
Hack App Data	-	-	10	9
Average query execution time*	*3.1 seconds*		*6.8 seconds*	

*Based on 50 executions of the query

3.6. Similarity detection

During the course of this work, well-known string and image similarity methods were evaluated. For completion reasons we provide a comparison and rationale behind the choice of the selected methods for our solution.

3.6.1. Name similarity.

For the name similarity detection, the *Jaro-Winkler* [16] method was used. Previous work has shown that it is more accurate than other string distance metrics for short strings, like names [3].

It was compared against the popular *Levenshtein* distance metric. The results are presented in *Table 1.* The input string was checked against the whole set of data stored in our trusted database. More information regarding the construction of the database and the data sets that were used is provided in *Section 4.2.* In the table, the *Rank* column demonstrates the order, while the *Similarity* column contains the similarity between the input string and the returned string, as calculated by the algorithms. The results that *Jaro-Winkler* returned are perceptually more accurate and the performance of the algorithm significantly better.

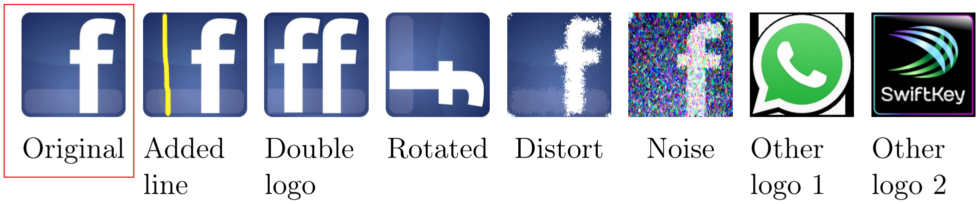


Figure 1. Test set

Table 2. Image comparison methods

	Haar Wavelet	**Histogram**	**Binarised Icon's Histogram**
Original	100	100	100
Added line	57.49	61.55	100.00
Double logo	46.97	71.98	94.33
Rotated	15.72	88.84	99.99
Distort	89.70	53.48	99.97
Noise	47.15	38.81	91.68
Other logo 1	4.01	7.67	52.87
Other logo 2	8.52	0.70	99.81

3.6.2. Icon similarity.

Various methods for the icon similarity detection were tested. In contrast with string similarity, image similarity detection algorithms are more complicated and the accuracy of their results is highly dependent on the given image set.

The state of the art in image recognition are training algorithms [7]. Since application icons in most occasions are drawings that do not frequently change, training an algorithm can be extremely challenging, because the lack of input data that is required. Therefore, a more applicable method for tackling this problem had to be used.

Comparing the histograms of two images produced promising results during our experiments. Colour similarity between the images and pHashes (perceptual hashing) were also tested, but gave significantly worse results. Finally, a method introduced by Jacobs et al. [11] was tested. Their method uses *Haar wavelet decomposition* techniques in order to determine how many significant wavelet coefficients the input image has in common with a target image and therefore determine the amount of similarity.

The results of the image comparison between the *original image* and the rest of the images shown in *Figure 1*, using the Haar wavelet and histogram comparison methods are presented in *Table 2*. More experiments were conducted, using different sets of images, that produced similar results in terms of accuracy.

The histogram comparison was tested for binarised versions of the images as well. The OpenCV[1] library was used for the histogram comparison and the *Correlation Algorithm* was used to calculate the distance between two histograms. The Correlation Algorithm was chosen because it produced significantly better results than the other histogram comparison algorithms that OpenCV provides (Chi-Square, Intersection and Bhattacharyya distance).

The Haar Wavelet method was noticed to be more resistant to images with added noise or distortion, while the histogram method could detect rotated images better. Histogram comparison is also more likely to return high similarity in cases where the target image has similar colours as the original one, but a different pattern is depicted. This could possibly lead to many false positives. Moreover, since application icons are relatively small, it is more likely that added noise or distortion will go unnoticed, than rotation. Therefore, the Haar Wavelet method was chosen to be used. More efficient algorithms for image and string similarity that will further improve our results may exist, but this is not the main focus of this work.

3.6.3. Application similarity metrics.

A user is more likely to trust (and consequently install) an application in which he/she can recognise symbols that are familiar to him/her [14]. Therefore, the higher the resemblance, the more likely it is for an application to get installed.

In our technique we used two similarity detection algorithms; name similarity and icon similarity. These algorithms return their similarity score, $\mathcal{X}$ and $\mathcal{Y}$ for the name and icon similarity respectively. To combine these similarity scores and compute the overall similarity we used *Equation 1*.

$$Sim_{a,b} = 10\frac{\mathcal{X} \times 10^{\mathcal{X}} + \mathcal{Y} \times 10^{\mathcal{Y}}}{2}\% \tag{1}$$

where $\mathcal{X}$ is the similarity percentage between the name of application a and b and $\mathcal{Y}$ is the similarity percentage between their icons, as returned from the algorithms described in *Sections 3.6.1* and *3.6.2*, respectively. The proposed equation weights the name and icon similarities in a way that high similarity

[1] OpenCV: `http://opencv.org/`

in one of the two elements would have a significant impact on the overall similarity. The same weight is given to the two similarity scores, which are then averaged.

In order for two applications to be considered similar, the similarity score returned by *Equation 1* should exceed 40%. The threshold of 40% was chosen because during our experimentation it was giving the most promising results in terms of false-positive and false-negative rates.

4. Implementation

In order to test the effectiveness of the solution, two tools were built. An Android application capable of extracting the required features from an `.apk` file, and a server responsible for the repackaging detection process.

4.1. The Android application

The Android application takes as input an `.apk` file and extracts the required features, as described in *Section 3.5*. It first extracts the name, the icon, the package name and calculates the hashes of the signatures that exist in the application. The name (since it may contain special characters) and the icon are encoded using Base64 encoding in order to be transmitted over the network. Finally, it decompresses the application to a new directory, removes the `META-INF` directory, compresses it again and calculates the SHA256 of the signed and unsigned applications. The extracted data is being sent to the detection server, using SSL in order to maintain the integrity and confidentiality.

In our experiments, a mid-range Android smartphone was used, *Samsung Galaxy S5 mini (SM-G800F)*, running Android 4.4.2.

4.1.1. Challenges.

Ideally this mechanism should be built in the package manager process or automatically run when an application is about to be installed. Due to Android limitations, the previous is not possible without further system modifications. A customised version of Android can integrate both, since the package manager is part of the Android Open Source Project. Since the integration of the solution to the operating system would not have any impact on proving its robustness, a proof of concept application that takes an `.apk` file as input and performs the whole process was built instead.

4.2. The server application

The server application was mainly written in Java, running on top of a Tomcat server, version 7.0.57. The host machine was a laptop with 8GB of RAM and an Intel Core i7-2620M CPU, running at 2.7GHz. The operating system was Ubuntu 14.10, Desktop edition. The database software used in the experiments was MySQL, version 5.5.41. It consisted of 3 tables.

A set of 2676 free applications were crawled from the Google Play Store (referred to as $\mathcal{S}_T$) and were added in our database, considered as authentic and non-repackaged.

4.2.1. Application similarity

- *Name similarity:* The name similarity was performed in the database, using an implementation of Jaro-Winkler written in SQL[2], in order to increase the performance. No further actions were taken in order to optimise the code for performance.

- *Icon similarity:* It was developed in Perl, using the `Image::Seek` module, which is capable of calculating image similarity based on the Haar wavelet decomposition method. Coefficients of the decomposition of the application icon in question are compared with pre-computed coefficients of trusted applications in the database.

5. Results and Evaluation

After the population of the trusted server with legitimate applications, as described in *Section 4.2*, three experimental phases were conducted in order to prove the robustness of the proposed solution. In the first two, applications from the Drebin malicious dataset[3] ($\mathcal{S}_M$) were used as input on the client device. During the third phase, legitimate applications that exist in the trusted database were used, in order to measure the performance of the solution in this scenario.

In the first phase of the experiment, the accuracy of the proposed solution was tested against applications with very high probability of being repackaged. According to previous research [12], altering the package name

[2] Jaro-Winkler code: `https://androidaddicted.wordpress.com/2010/06/01/jaro-winkler-sql-code/`

[3] Drebin dataset: `http://user.informatik.uni-goettingen.de/~darp/drebin/`

Table 3. Experimental results

	Set size	True positive	True negative	False positive	False negative	Average time/check
Phase 1	$\mathcal{S}_F$(227 apps)	203	3	2	19	3.42 seconds
Phase 2	$\mathcal{S}_R$(100 apps)	11	88	1	0	3.9 seconds
Phase 3	$\mathcal{S}_L$(50 apps)	50	0	0	0	0.6 seconds

of an application would lower its visibility in markets. Therefore, a set of such applications can be generated by extracting applications from the set $\mathcal{S}_M$, with package names that exist in the Play Store. 227 free (non-paid) applications ($\mathcal{S}_F$) were extracted and used in our experimentation.

In the second phase, 100 random applications ($\mathcal{S}_R$) were chosen from $\mathcal{S}_M$, excluding $\mathcal{S}_F$ ($\mathcal{S}_R \subseteq \mathcal{S}_M \setminus \mathcal{S}_F$), in order to further investigate the accuracy of the solution with applications that are most likely *not* repackaged (false positive). The main goal of this test was to measure the amount of non-repackaged applications that would be detected as repackaged.

In the last phase, 50 representative legitimate applications ($\mathcal{S}_L$, where $\mathcal{S}_L \subseteq \mathcal{S}_T$) were randomly selected and tested in order to measure the performance of the system when legitimate applications are being tested.

The first two phases of the experiments were analysed in terms of accuracy and performance. Manual inspection of the applications in the sets was finally performed in order to assess the quality of our results.

5.1. Results

Table 3 presents the results from the three experimental phases described in the previous section. *Average time/check* refers to the average time required from the moment the device initiated the process, until it displayed the results on the screen. Cases in which the original version of the repackaged application being checked was included in the returned results were marked as true positives. Returned results that did not include the original application were marked as false positives. True negatives were considered cases in which no results were returned from the server and no corresponding original application existed. Finally, we marked as false negatives cases in which no similar applications were returned and our trusted database included the original application.

During the first phase of the experiment, 189 applications (83.3%) returned exactly one result. 15 applications (6.6%) returned two results. One application (0.4%) returned three results and the 22 remaining (9.7%) did not return any result. The time required for the extraction of the features from the `.apk` file on the device was approximately 0.18 seconds. Approximately 2.6 seconds were spent on the application name comparison on the server side. The time required for the image similarity detection was negligible. The average application size was 2.1MB. A slight time overhead will be added on the client side in case of larger applications, since the decompression phase, as well as the computation of the hashes will take longer to complete. This process only takes place once, prior to the applications installation though, causing a negligible impact on the overall operating system's performance.

During the second phase, 9 applications returned exactly one result. Two returned two results and one returned four results. 3.5 seconds were required for the server to take a decision, 3.2 of which were required for the string similarity comparison. 0.14 seconds were required for the Android device in order to extract the required data from the `.apk` file. The average application size was 1.5MB.

Finally, during the third phase, only applications that already existed in our database were tested. This phase was performed in order to calculate the system's performance in this occasion. 0.04 seconds were required by the server for the detection process. Data extraction on the client required 0.3 seconds to complete and the average `apk` file size was 4MB.

The database that holds the Haar transformation norms and gets populated when new icons are added to the set of application icons, requires an average (after 5 runs of the population script) of 1 minute and 7 seconds, for a total of 2676 images. Therefore, 0.025 seconds for each new added icon.

5.2. Evaluation

The results from the performed experiments were evaluated.

5.2.1. First phase.

After manual analysis, both false positives detected in this phase were found not to be repackaged applications of the applications with the same package names in the Google Play Store. Both the package names were simple (`com.hotel` and `com.acid`) and the existence of applications with the same package name was most probably due to a coincidence.

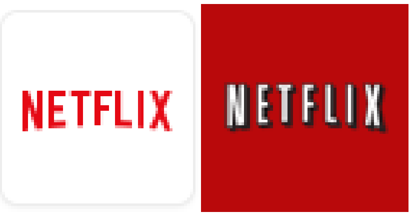
Nineteen false negatives were also detected. Fifteen of them shared similar icons with the corresponding original applications. Their application names were in Chinese, in contrast with the original applications whose names are in

English. The measurement of the name similarity returned 0 and the amount of the icon similarity was not enough to compensate. Since the samples in $\mathcal{S}_M$ were collected between 2010 and 2012 and the original corresponding applications may have been updated numerous times since then, it is unclear whether the application names of the original applications were in Chinese at the time of capture, or the names of the repackaged applications have been altered. The original application developers in all cases seem to be Chinese, based on Play Store data.

The remaining four false negative results were returned by applications that had a completely different icon than the original application, and a different name. Again, it was not clear whether the attacker who performed the repackaging altered the name and icon, or these were the name and icon of the original application at the time of repackaging.

The remaining three applications that did not return any results were not repackaged versions of applications that exist in the Google Play Store. They only shared the same package names with Play Store applications. Therefore, correctly, no results were returned.

Forty applications that were correctly recognised had similar, but different icons than the original ones in our database. For example, a repackaged version of Netflix contained the old application logo, which compared to the new one has reverse colours, different fonts, shadows and rounded corners *(Figure 2)*. Applications with resized icons between the original and the repackaged version were not counted, since all icons get resized to 128x128 prior to the similarity detection.



(a) Original **(b)** Repackaged

Figure 2. Netflix app icons

Moreover, it was observed that in all occasions, when the server would suggest more than one potentially original applications, the one with the highest similarity to the application being tested was the correct suggestion.

5.2.2. Second phase.

During this phase, no false negatives were detected. There was one false positive and 11 true positives.

The true positive results were not detected as repackaged when only looking at the package name, that suggests that it might be a common practice for attackers to alter it. Four applications were also detected to have a slightly different icon than the original applications.

Finally, the false positive was caused by an application that had the default icon that *Eclipse IDE* adds on newly created Android projects. A total of 4 applications with this particular icon existed in our database and were returned. It is a common practice for developers to change the default icon.

5.2.3. Observations.

The proposed solution demonstrated its effectiveness in detecting repackaged applications (91.5% detectability, according to the results of the first experimental phase). During our experiments we detected repackaged applications that were not using the original application's source code, but only its name and icon. These applications would probably have gone undetected by countermeasures based on other methods, discussed in *Section 2.2*.

An application that is known to use this repackaging technique, disguising itself as the *Google Play Store* [17], but using the icon of an older version of the application and *"googl app stoy"* as the name, was successfully detected. Only one result was returned by the server; the original Play Store. Although this type of attack is not new [4], to the best of our knowledge, there has been no prior solution capable of effectively detecting this kind of applications. More applications like the previous were detected during our experiments.

The best example is a fraudulent application with the package name `net.android.app`. Four variants of the application were found in the set $\mathcal{S}_F$, using this package name. After manual inspection of the disassembled code, we concluded that all variants share the same source code, which is capable of sending text messages to premium-rate numbers, without the user's authorisation.

The first version of the sample (based on that it was found in the first chronologically ordered directory of $\mathcal{S}_M$), uses *"браузер"* (Russian for *"browser"*) as a name and the logo of *Internet Explorer* as an icon. Since no similar applications were included on the server, correctly, no similar applications were returned.

The later variants of the sample used different names and icons. The first two used *"Skype"* as their name and the original Skype icon. The last used *"Opera Mini"* as its name and the Opera Mini browser's icon. We believe that this was a well thought move from the attackers side towards increasing the distribution vector of the application.

Table 4. Comparison with previous works

Criteria	DNADroid	DroidMOSS	AppInk	Proposed
1. App store centric	✓	✓	✗	✓*
2. Download source agnostic	✗	✗	✓	✓
3. Client notification	✗	✗	✓	✓
4. Detects stolen branding	✗	✗	✗	✓
5. Detects stolen source code	✓	✓	✗	✗
6. Resilient to code obfuscation	✗	✗	✓	✓
7. Requires trusted sample	✓	✓	✓	✓

*Although the current implementation focuses on the client side, there are no restrictions for scanning an application store using the proposed solution

None of the existing application repackaging detection methods should be capable of detecting this kind of repackaging. Our results suggest that it is not a common practice for attackers to alter the names and icons of repackaged applications.

5.3. Comparison with previous works

Table 4 presents a comparison between our proposed solution and popular previous solutions, described in *Section 2.2*. A short description and explanation is provided below:

Most of the previous works rely on the detection being performed on the application store side *(criterion 1)*. It has been noticed that many third-party markets do not remove malicious applications, and in many occasions the distribution of malicious applications and adware is being done intentionally [12]. Since anyone with technical knowledge can potentially create an application market, it is unsafe to rely on them for the detection and removal of potentially harmful applications. Many are likely not to scan for repackaged applications due to the computational cost and the lack of technical skills. Finally, there are dedicated websites and forums that distribute Android applications. Scanning application market servers for malicious and repackaged applications does not protect against applications that have been downloaded from such sources. Our proposed solution was designed to be able to initiate the detection process from the client side, making it application download source agnostic *(criterion 2)*, and notifying the user in order to prevent the installation of a potentially repackaged application *(criterion 3)*.

Another shortcoming of the previous techniques is that they are mostly based on code analysis, or use features that are strongly linked with the application's code. Therefore, they would not be capable of detecting applications that claim to be legitimate applications by just imitating their name and application icon, but using a completely different codebase *(criterion 4)*, or when advanced obfuscation techniques have been applied. Solely in 2014, at least two samples were detected claiming to be the *Google Play Store* application, none of which was using the original application's source code [1, 17]. Such solutions though may be able to detect whether an application has stolen parts of the source code of another application *(criterion 5)*.

Finally, all proposed solutions have to maintain samples of trusted applications (or some other trusted derivative), against which a tested application is compared *(criterion 6)*. A repackaged application cannot be detected by any method, unless that method maintains the necessary information of the authentic corresponding application.

6. Conclusion and Future Work

In this paper, we proposed a repackaging detection method that takes advantage of the attacker's inability to significantly modify the application's name and icon, while maintaining the attack vector. Therefore, an attacker is not likely to perform such modifications. Our experimental results showed that our initial argument was valid. We were capable of detecting the original applications, given a repackaged application as an input.

Compared to previous work, our solution was found to be about as accurate as application market level detection techniques, but capable of initialising the detection process from the client side, allowing for source independent detection. Only a few kilobytes of data is required for the detection, which adds a slight overhead only on the application installation process. Moreover, no special actions are required from the developer's side. Finally, it is capable of detecting a variation of repackaged applications that only share the same name and icon with the original. As far as we know, it is the first one of its kind.

Our solution can be expanded by further investigating string and image similarity algorithms and techniques. A more in depth study can assist towards the better selection of the threshold over which two applications are considered similar, leading to even lower false-positive and false-negative rates. Finally, code optimisations can further improve our solution's performance.

References

[1] Android FakeMarket Analysis. http://tinyurl.com/pwvam9w, 2014. [accessed 24-March-2015].

[2] Loredana Botezatu. 1.2 Percent of Google Play Store is Thief-Ware, Study Shows. http://tinyurl.com/kvf7xvc, 2013. [accessed 10-June-2015].

[3] William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IJCAI-03 Workshop on Information Integration on the Web*, pages 73–78, 2003.

[4] Peter Coogan. More fraudware headaches for the Android Marketplace. http://tinyurl.com/o58dryj, 2012. [accessed 10-June-2015].

[5] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Computer Security - ESORICS 2012*, volume 7459 of *LNCS*, pages 37–54. Springer, 2012.

[6] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Computer Security - ESORICS 2013*, volume 8134 of *LNCS*, pages 182–199. Springer, 2013.

[7] Ritendra Datta, Jia Li, and James Z. Wang. Content-based Image Retrieval: Approaches and Trends of the New Age. In *the 7th ACM SIGMM International Workshop on Multimedia Information Retrieval*, pages 253–262, USA, 2005. ACM.

[8] Anthony Desnos and Geoffroy Gueguen. New "open source" step in Android application analysis. PacSec Conference 2012, 2012.

[9] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, Berlin, Heidelberg, 2013. Springer.

[10] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Trust and Trustworthy Computing*, volume 7904 of *LNCS*, pages 169–186. Springer, 2013.

[11] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast Multiresolution Image Querying. In *the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 277–286, USA, 1995. ACM.

[12] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, pages 51–71. Springer, 2014.

[13] OWASP. Projects/OWASP Mobile Security Project - Top Ten Mobile Risks. `http://tinyurl.com/pytw57n`, 2014. [accessed 18-March-2015].

[14] Jens Riegelsberger. *Trust in Mediated Interactions.* PhD thesis, University College London, July 2005.

[15] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In *the 30th Annual Computer Security Applications Conference*, USA, 2014. ACM.

[16] William E. Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *the Section on Survey Research*, pages 354–359, 1990.

[17] Jinjian Zhai and Jimmy Su. What are you doing? - DSEncrypt Malware. `http://tinyurl.com/o2fuzzm`, 2014. [accessed 24-March-2015].

[18] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In *the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, USA, 2014. ACM.

[19] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast Detection of Repackaged Applications. In *Data and Applications Security and Privacy XXVIII*, LNCS, pages 130–145. Springer, 2014.

[20] W Zhou, X Zhang, and Xuxian Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. *Proceedings of the 8th ACM SIGSAC*, pages 1–12, 2013.

[21] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *the second ACM conference on Data and Application Security and Privacy - CODASPY '12*, pages 317–326, 2012.

[22] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Security and Privacy, 2012 IEEE Symposium on*, pages 95–109, May 2012.

Iakovos Gurulian (BSc, MSc) received his BSc (Hons) in Computer Science from University of Surrey in 2011 and his MSc in Information Security from University College London in 2012. He is currently an Information Security researcher at Royal Holloway, University of London. His main research interests include smart-device security, network security and user-centric security.

Dr Konstantinos Markantonakis (B.Sc, M.Sc, MBA, Ph.D) received his BSc (Hons) in Computer Science from Lancaster University in 1995, his MSc in Information Security in 1996, his PhD in 2000 and his MBA in International Management in 2005 from Royal Holloway, University of London. He is currently a Reader (Associate Professor) in the Information Security Group. His main research interests include smart card security and applications, secure cryptographic protocol design, Public Key Infrastructures (PKI) and key management, embedded system security, mobile phone operating systems/platform security, NFC/RFID security, grouping proofs, electronic voting protocols. He has published more than 130 papers in international conferences and journals.

Dr Lorenzo Cavallaro is a Senior Lecturer (roughly equivalent to Associate Professor in the USA) of Information Security in the Information Security Group (ISG) at Royal Holloway University of London. His research focuses largely on systems security. To this end, he has founded and is leading the recently-established Systems Security Research Lab (S2Lab) within the ISG, which focuses on devising novel techniques to protect systems from a broad range of threats, including those perpetrated by malicious software. In particular, Lorenzo's lab aims ultimately at building practical tools and provide security services to the community at large.

Prof Keith Mayes B.Sc. Ph.D. (Bath) CEng FIET received his BSc (Hons) in Electronic Engineering in 1983 and a PhD degree in Digital Image Processing in 1987. He spent much of his career working in industry for Pye TVT, Honeywell, Racal and Vodafone, and today is the Director of the Information Security Group and Smart Card Centre at Royal Holloway University of London, as well as the Director of Crisp Telecom Limited. He is an active researcher with 100+ publications and current interests include the design of secure protocols, mobile/fixed communications systems and security tokens/NFC/RFID as well as associated attacks/countermeasures. Keith is a Fellow of the Institution of Engineering and Technology, a Founder Associate Member of the Institute of Information Security Professionals, a Member of the Licensing Executives Society and a member of the editorial board of the Journal of Theoretical and Applied Electronic Commerce Research (JTAER). He has had director experience with a London stock market listed company and an American communications company. He led the expert team that carried out counter-expertise work on the Ov-Chipkaart for the Dutch transport ministry, (following attacks on MIFARE Classic); and was recently the ESORICS2013 General Chair.