Repadiography: towards a visual support for triaging repackaged apps[*]

Gerardo Canfora[1],Sara Caruso[2], Andrea Di Sorbo[1], Marianna Fucci[2], Sonia Laudanna[1], and Corrado Aaron Visaggio[1]

[1]University of Sannio, Benevento, Italy

[2]NTT Data, Naples, Italy

canfora@unisannio.it, sara.caruso@nttdata.com, disorbo@unisannio.it, marianna.fucci@nttdata.com, {slauadanna,visaggio}@unisannio.it

Abstract

App repackaging is a method for conveying malicious or disturbing code, consisting in decompiling an existing app, adding third party code, recompiling the resulting app and distributing it on marketplaces. Recent studies claim that repackaged apps populate both third party and official marketplaces. Solutions for detecting repackaging have been proposed in the literature but few efforts have been devoted to support the triaging activities. The triage is a preliminary automatic analysis aimed at minimizing the time an analyst spends examining potentially harmful applications. Given the high volumes of apps published on the marketplaces and the high speed of production and diffusion of apps, analysts need effective means for accelerating the triaging phase. For this reason, we propose a solution for visually comparing a legitimate app with a repackaged one, and allowing the analyst to immediately locate and quantify the impact of repackaging on the original app's code.

1 Introduction

Repackaged apps are altered versions of legitimate apps: they are an insidious vector for distributing malicious code, disturbing functions, or undesired content, leveraging the popularity of

[*] Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

other apps [6]. Repackaged apps are distributed mainly by third party marketplaces, but they can be found in official marketplaces, too[26]. As pointed out by Meng et al. [14], the current feature-based malware detection approaches are not adequate for detecting repackaged apps, because they cannot provide detailed information beyond the mere detection. Repackaging is a serious threat to the Android ecosystem as it deprives application developers of the benefits of their efforts, contributes to spreading malware on users' devices and increases the workload of the official marketplace maintainers. To create a repackaged app, a malicious developer could download a known application from a legitimate app store, extract its legitimate files, make harmful changes to the app and then repackage it. Once the changes have been made, the malicious developer publishes the re-packaged app in an Android app store (particularly on independent ones, other than Google Play), and thus the attacker simply waits for users to download it, attracted by a possible activation of premium features, otherwise for a fee. These apps are usually almost identical to the original ones so that the user does not notice the altered behavior (or look). While many solutions have been proposed for detecting repackaged apps, a few was done for the triaging activities. Given the volumes (for instance, Google Play counts over 3 million Android apps[†]) and the speed of production and diffusion of apps, it is necessary to have solutions that may accelerate and make more accurate the malware triage [20].

This goal could be pursued through a visual analysis of the apps, given that the human brain has evolved for extracting quickly heterogeneous and layered information from images. Indeed, software visualization has been largely investigated [17] for helping the engineers to handle the complexity of software systems in the different phases of the software process and for capturing the diversity of software facets. Hence, we propose a tool, namely *Repadiography,* which resumes into an image different pieces of information about a pair (apk_o, apk_r), being apk_o the original app, and apk_r the repackaged one, i.e., (i) the *size* of the modifications introduced by the repackaging; (ii) the *localization* of the new or altered code with respect to the original app; and (iii) the *intensity* of the alteration produced by the repackaging.

In order to reach a proper definition of the functions that the visual tool should expose, we carried out an observational study for quantifying and characterizing the changes done on an app when it is repackaged.

Based on the results of the observational study, we established that our solution should compare the original app and the repackaged app upon three aspects: the structure, the behavior, and the interaction with the user. Characterizing the modifications to the structure of the original apps means to understand the portion of the code concerned by the repackaging.

In order to evaluate how much the behavior of the app changes, the tool examines the extent to which the control flow of the repackaged app differs from the one of the original app.

Finally, the tool takes into account how interaction with the user is modified through the alterations to the graphical interface.

The paper provides an original contribution upon:

- the characterization of the most common alterations introduced by repackaging to the original app, with quantitative indicators; and
- the technology for enabling the triaging of repackaged apps, since we developed a visual tool that converts the original app and the repackaged one into images and uses the colored strips for distinguishing parts of the code that are the same, those that are altered and those that are added; finally the shades of the color characterize the intensity of the alteration.

This paper is composed of the following sections: the next section discusses the related work, section 3 introduces the design of the observational study, while the following section examines the results. Section 5 presents the visual tool, while section 6 draws the conclusions and the future work.

[†]https://www.appbrain.com/stats/number-of-android-apps

2 Related work

The methods proposed till now for detecting repackaged apps rely on two approaches: static and dynamic analysis. It is worth noticing that some static approaches do not analyze the bytecode of apps but solely examine the resource files accompanying the code. Different features have been investigated for determining when an app is a repackaged version of another app. These features are extracted from metadata (e.g., permissions recorded in the Manifest file), from the code (e.g., call graphs), or from runtime data (e.g., execution traces) [13].

The most common approach in the literature is the similarity computation that relies on the assumption that the metadata of the repackaged app is very similar to that of the original app. Since Androguard [2], [8], which has proposed algorithms for pairwise comparison of apps, several variants using code information have been developed, as well as: DNADroid [7] using dependence graphs, DroidMOSS [21] applying fuzzy hashing-based fingerprints, DroidEagle [18] leveraging layout/resource information or a combination of both like ResDroid [16] and ViewDroid [23], while Zhou et al. in [25] have built vectors using normalized values of extracted features for fast and scalable detection of piggybacked apps. A second widely used approach is based on monitoring app execution and extracting runtime information. In [24], the authors proposed to adopt a dynamic graph based watermarking mechanism and introduced the concept of manifest app, which is a companion app for an Android app under protection.

Supervised learning-based approaches extract feature vectors from app data and train classifiers that will be used to predict whether an app is repackaged or not. In [19], Tian et al. proposed a technique based on code heterogeneity analysis, which partitions the code structure of an app into multiple dependence-based regions (subsets of the code). Each region is independently classified on its behavioral features. Finally, a symptom discovery-based intuitive approach is presented in [9]. This is a lightweight approach for detecting symptoms of repackaging in Android apps using a novel feature, the String Offset Order (SOO from here on), which is extracted from string identifiers list in the *classes.dex* bytecode file. As opposed to the existing methods, SOO offers a robust and fast mechanism to differentiate between original and repackaged code even in the presence of obfuscation.

Recently, several visualization techniques have been proposed to compensate or to help malware analysis. Kancherla and Mukkamala in [12] presented an image visualization-based malware detection technique. First the executable is converted into byteplot image. Later analysis is performed on this byteplot and used Support Vector Machines (SVMs) to classify the images, obtaining an accuracy of 95%. Han et al. in [10], introduced a malware family classification method using visualized images and entropy graphs. In [15], Ni et al illustrated a malware classification algorithm that uses static features called MCSC (Malware Classification using SimHash and CNN) which converts the disassembled malware codes into gray images based on SimHash and then identifies their families by a convolutional neural network.

Our approach converts apps in images formed by strips of color, where each strip represents a line of code, while the different colors indicate the similarity degree among the two apps.

3 The Study Design

This study aims at characterizing the most widespread traits of the repackaging concerning: the code's structure, the app's behavior, and the interaction with the user.

We posed the following research question:

RQ1: *Which is the impact of repackaging on the original app?*

For answering the research question RQ1, we collected the following metrics:

- the *number of type 1 clones*, the number of exact copies without modifications for evaluating how much malicious or disturbing code is used as-is in repackaging;
- the *number of added methods*, that measures how much new code the repackaged app brings;
- the *number of altered methods*, that quantifies how extended is the portion of original code that has been modified by the repackaging;
- the *percentage of alteration for each altered method*, that provides the magnitude of alteration to the original code;
- the *variation of cyclomatic complexity*, that resumes how the behavior of app changes with repackaging; and
- the *variation of GUI elements,* that characterizes whether and how much the user's interface of the app was interested by the repackaging.

The dataset consisted of 1200 pairs of original apps and repackaged apps, taken by the RePack database [4] which contains repackaged Android apps extracted from AndroZoo [1].

For collecting the metrics, all the apps in the dataset have been first decompiled with apk2java [5], for having the source code available. Type 1 clones were detected with Simian [11], which provides: number of lines of code the clone is composed by, the corresponding list of all the locations within the source code of the analyzed app expressed in the form of path, the number of the starting line of the clone and final line number.

The added code was identified with Androsim [3], by comparing all the pairs (apk_o, apk_r) of the dataset, being apk_o the original app and apk_r the repackaged one. For each pair (apk_o, apk_r), Androsim produces a report with the information for computing the altered and added methods and the percentage of alteration.
For gathering the amount of added code, we extracted the total size of the .java files of the legitimate app and of the repackaged versions. Finally, we proceed to compare the dimensions obtained.

Once these measurements have been gathered, we extrapolated the number of lines of code that the developer of the repacked app has added. The cyclomatic complexity of each structured program (single entry / exit point) is computed as:

$$\pi + 1$$

where π is the number of decision points (IF, FOR, WHILE) contained in the program.
The percentage of GUI alteration introduced by the repackaged application is computed as:

$$alteration = \frac{\#\ added\ GUI\ elements}{\#\ original\ GUI\ elements} * 100$$

where the numerator refers to the graphic elements (declarative descriptions of layouts and widgets contained in the res/layout folder of each android application) added by the repackaged app and the denominator those that were already part of the original application.

4 Discussion of results

The analysis of type 1 clones has highlighted the absence of methods that are the exact copy of an added one: this means that the code added for repackaging is written ad-hoc for the target app.

The impact of repackaging on the original app is very limited to a small portion of the original code and also the added code represents a little part of the overall app: this suggests that the analyst must isolate a very small part of the repackaged app, during triaging.

Type of method	**Percentage [%]**
Unaltered methods	93.49
Added methods	5.18
Altered methods	1.33

Table 1: Dispersion of the repackaged code

Table 1 shows the totality of the methods that are part of the repackaged application. Specifically, it has three distinct sections designed to represent the dispersion of the repackaged code:

- 93.49% of the methods were not affected by the alteration. They are therefore identical to those of the original application;
- 5.18% of the methods were not part of the original app, thus this percentage represents the portion of new methods added in the repackaged app; and
- finally, only 1.33% of the methods in the original app have been partially modified.

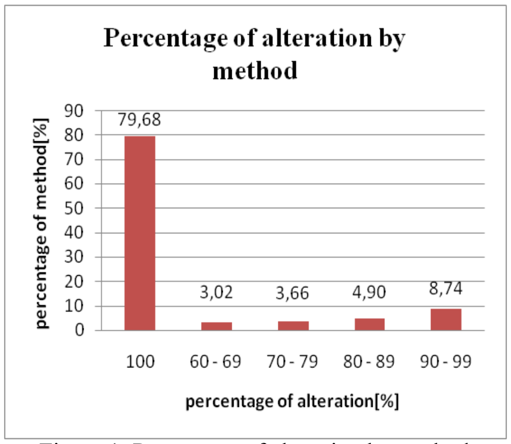


Figure 1: Percentage of alteration by method

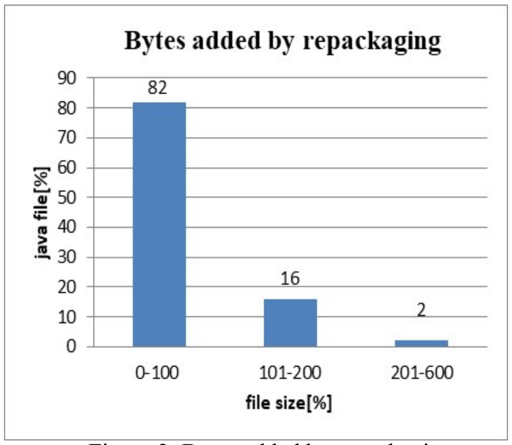


Figure 2: Bytes added by repackaging

Figure 1 plots the changes that have affected the original apps: only 7% (added methods and similar methods) of the total methods belonging to the original app are the target of repackaging:

- 79.68% of the methods have undergone a 100% alteration. This means that the methods affected by this percentage are those created and added to the original application;
- 8.74% of the methods were changed to 90-99%. These methods, although native to the original application, have been almost completely altered;
- 4.90% of the altered methods have undergone a modification between 80 and 89 percent;

- 3.66% of the methods affected by the changes have undergone a variation between 70 and 79 percent, and
- the body of 3.02% of the methods has undergone a change that does not exceed 69%.

In conclusion, it can be said that repackaging mainly introduces new methods, while the portion of modified methods is localized to a number of methods. In this case the most methods are quite completely rewritten.

Figure 2 shows that repackaged apps increase the size of the app, result that is consistent with the previous finding about the added methods. In the 82% of cases the repackaged applications double the size of the original app. While, for 18% of cases repackaging increases the size more than twice, up to six-fold almost in one case.

Looking at figure 3, it is possible to understand how the repackaging phenomenon produces, for 99.66% of the original apps, an increase of cyclomatic complexity for a value ranging from 0 to 30%. This suggests that the behavior of the repackaged app undergoes significant deviations from that of the original app, even keeping a large part of the original app's functionality. This is explainable by the necessity of a repackaged app of implementing some new undesired or malicious functions, but hiding them behind those of the original app.

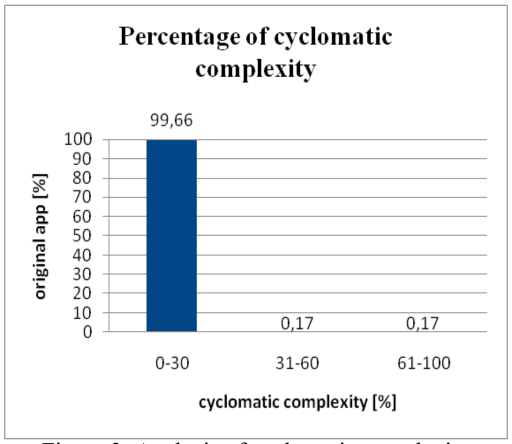


Figure 3: Analysis of cyclomatic complexity

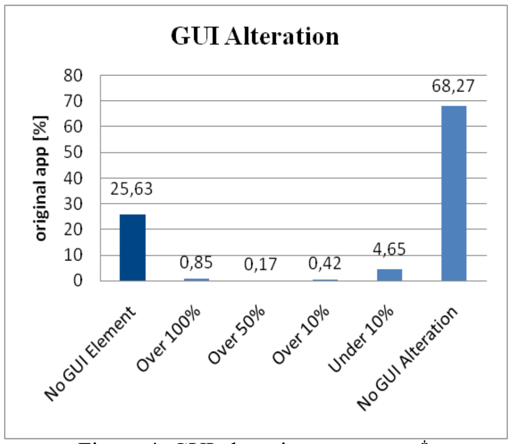


Figure 4: GUI alteration percentage[‡]

Finally, figure 4 shows the data related to the alterations that the repackaged application developer introduces at the graphic interface level:

- in 25.63% of cases there is no element typical of user interfaces at all: in this case repackaged app runs in background or is a system utility, which is one of the best strategies to evade the detection of user;
- for 5.07% of cases the GUI was changed between 1% and 50%; and
- more than 50% of the repackaged applications, precisely 68.27%, make no changes to the original user interface, which makes sense in the perspective of camouflaging itself at the best.

[‡]The "over 100%" label means that the GUI of the repackaged app introduces a completely different GUI with respect to the GUI of the original app (i.e., all the existing elements are modified and new ones are added).

In summary, our study leads to the following conclusions: repackaging rarely affects GUI, the alterations of the original app consist mainly of adding new methods, and the modification of the existing methods is limited to a small part of the code. In this scenario, a visual tool could be useful to localize the added and altered methods, and characterize the amount of alteration for the method, and especially can support the visual comparison of different repackaged apps at the same time.

5 Repadiography

The proposed solution, namely Repadiography[§],is conceived to facilitate the visual comparison between repackaged apps and the legitimate original version, allowing: the localization of the altered and added code; the evaluation of the portion of altered or added code; and, finally, to estimate the alteration's impact.

For pursuing this goal, Repadiography transforms an Android app into an image, where each method corresponds to a strip of color. The color strip is determined by a function converting the alphabet used for writing the code and the RGB components.

By aligning the two images corresponding to the two apps to examine (original and repackaged one) the analyst can obtain an immediate evaluation of the actions done for the repackaging, at a glance; the resulting images have a minimum of one and a maximum of three distinct sections that will result in four possible cases:

- in the first case, the two images are equal and both include one only section of the same color, meaning that the apps contain exactly the *same methods*;
- in the second case, the images have two sections: the first one is the same for both the images and corresponds to the common part of the two apps; the second section has a different shade of color and refers to the methods that have been *changed* in the repackaged app: the intensity is proportional to the diversity among the methods;
- in the third case the image of the first app consists of a single section referring to the *identical* methods in the two apps, while the image of the second app is made by two sections: a section is equal to the first app, while the additional section in a different color identifies the *added* methods; and
- in the fourth case the image of the first app has two sections, one referring to the *identical* methods and the other one to the *similar* methods; the image of the second app has two sections: one for the *identical* methods while the other one for the *added* methods.

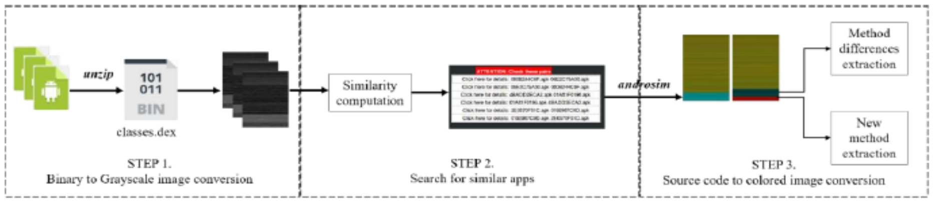
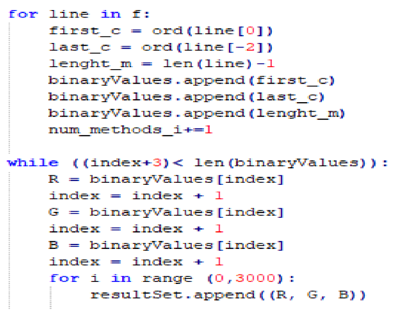


Figure 5: Visual tool process

The production of the images for the visual comparison consists of three steps, as shown in Figure 5:

[§]Repadiography is available at: https://github.com/sonialau/repadiography

• *Step 1*: the *class.dex* of the selected apk is converted into a grayscale image. All the bytes that make up the selected *classes.dex* file are acquired, and the decimal value of each byte is assigned to the components R, G and B in order to create a pixel that assumes a value ranging from 0 to 255 (0 black, 255 white).
In order to identify each method in a unique way, from each method a triple was extracted made of the first and the last character, along with the number of characters forming the method.
The ASCII codifications of the two characters and the length have been concatenated and this value was used for determining the R,G, and B components of the pixel. This algorithm is shown in figure 6.



```
for line in f:
    first_c = ord(line[0])
    last_c = ord(line[-2])
    lenght_m = len(line)-1
    binaryValues.append(first_c)
    binaryValues.append(last_c)
    binaryValues.append(lenght_m)
    num_methods_i+=1

while ((index+3)< len(binaryValues)):
    R = binaryValues[index]
    index = index + 1
    G = binaryValues[index]
    index = index + 1
    B = binaryValues[index]
    index = index + 1
    for i in range (0,3000):
        resultSet.append((R, G, B))
```

Figure 6: Algorithm implementing the conversion of methods

Since catching the different pixels between two images is difficult for a human eye, each generated pixel has been replicated on a row for obtaining a strip.
Each strip that composes the image represents a method and each image contains all the methods of the app.
Once all the bytes have been converted, the respective images of the apps will be created.

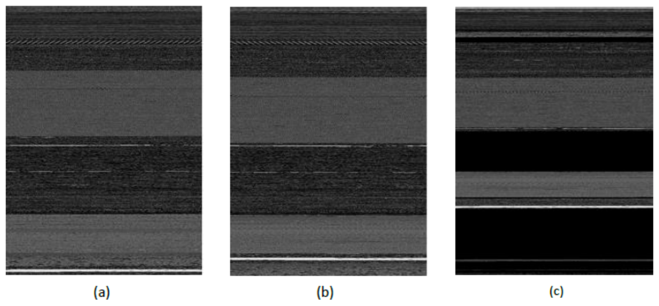


Figure 7: Examples of grayscale images representing apps

As can be seen from figure 7, the images (a) and (b) refer to similar apps. They expose the same texture except for small variations, indeed: this is the typical case of a pair containing the original and the repackaged app.
Figure (c), on the other hand, shows an image that appears to be completely different from the other two, so it can be said that this belongs to an app that has not relationship with the previous ones;
• *Step 2*: the similarity between the images created in Step 1 is computed. Precisely, to calculate the similarity we used the Levenshtein distance [22], or edit distance, which evaluates how much two strings are different. The Levenshtein distance between the string *a* and *b* is the minimum number of elementary modifications which allow to convert *a* to *b*.

Only those that exceed a limited threshold of similarity (55%) will be proposed as potential pairs (*original*, *repackaged*). This threshold value was chosen relying on the trials done in the case study, in order to guarantee a good compromise between false positives and false negatives. It is evident that in a real scenario, this parameter can and should be set according to the baselines and the experience of the analyst; and

• *Step 3*: Androsim tool compares the selected apks pair and, once selected, the corresponding pair of the images is generated where each strip represents the color coding of an apk method. The result of step 3 is shown in the figure 8.

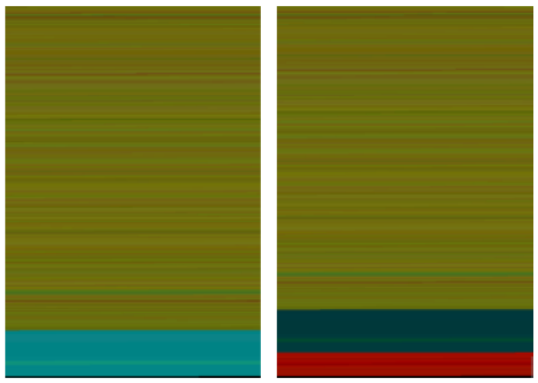


Figure 8: On the left the original app, on the right the repacked app

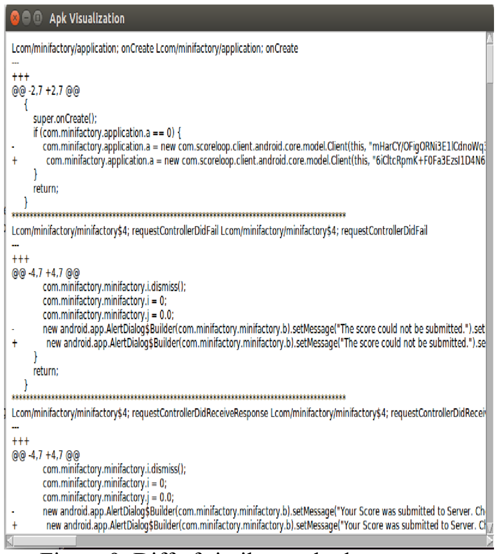
In the figure, there are three sections:

- the green section represents the *identical* methods: each strip of color is a method that is common to both the apps;
- the blue section represents *similar* methods: in the image on the right we notice a more intense color(blue) than the one on the left (sky blue); the greater intensity of the color is given by the value of similarity: the more intense the colors are, the less similar the methods are; and
- the red section, occurring only in the image on the right, represents the methods that have been *added* in the repackaged app.

In this case the analyst at a glance can acquire three pieces of information:

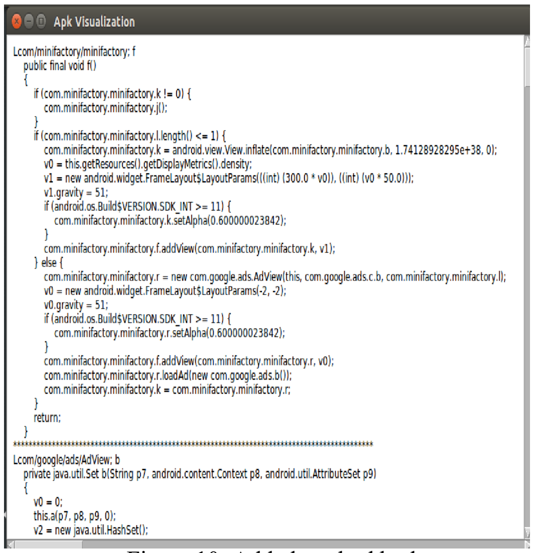
1. the repackaged app includes both altered methods and added ones;
2. the portion of interested code is around one fourth of the original code; and
3. the altered methods have been deeply changed.

Finally, the code interested by repackaging can be quickly obtained as each strip is mapped with the corresponding lines of code in the apps; in the implementation, the lines of code can be retrieved by simply clicking on the strip of the imaged, as illustrated in figure 9. Figure 9 shows the differences between the two apps, while figure 10 shows the body of the methods added in the repackaged app.



```
Lcom/minifactory/application; onCreate Lcom/minifactory/application; onCreate
---
+++
@@ -2,7 +2,7 @@
    {
       super.onCreate();
       if (com.minifactory.application.a == 0) {
-          com.minifactory.application.a = new com.scoreloop.client.android.core.model.Client(this, "mHarCY/OFigORNi3E1lCdnoWq
+           com.minifactory.application.a = new com.scoreloop.client.android.core.model.Client(this, "6iCltcRpmK+F0Fa3EzsI1D4N6
       }
       return;
    }
****************************************************************************************
Lcom/minifactory/minifactory$4; requestControllerDidFail Lcom/minifactory/minifactory$4; requestControllerDidFail
---
+++
@@ -4,7 +4,7 @@
          com.minifactory.minifactory.i.dismiss();
          com.minifactory.minifactory.i = 0;
          com.minifactory.minifactory.j = 0.0;
-         new android.app.AlertDialog$Builder(com.minifactory.minifactory.b).setMessage("The score could not be submitted.").set
+          new android.app.AlertDialog$Builder(com.minifactory.minifactory.b).setMessage("The score could not be submitted.").se
       }
       return;
    }
****************************************************************************************
Lcom/minifactory/minifactory$4; requestControllerDidReceiveResponse Lcom/minifactory/minifactory$4; requestControllerDidRecei
---
+++
@@ -4,7 +4,7 @@
          com.minifactory.minifactory.i.dismiss();
          com.minifactory.minifactory.i = 0;
          com.minifactory.minifactory.j = 0.0;
-         new android.app.AlertDialog$Builder(com.minifactory.minifactory.b).setMessage("Your Score was submitted to Server. Ch
+          new android.app.AlertDialog$Builder(com.minifactory.minifactory.b).setMessage("Your Score was submitted to Server. C
```

Figure 9: Diff of similar methods



```
Lcom/minifactory/minifactory; f
   public final void f()
   {
      if (com.minifactory.minifactory.k != 0) {
         com.minifactory.minifactory.j();
      }
      if (com.minifactory.minifactory.l.length() <= 1) {
         com.minifactory.minifactory.k = android.view.View.inflate(com.minifactory.minifactory.b, 1.74128928295e+38, 0);
         v0 = this.getResources().getDisplayMetrics().density;
         v1 = new android.widget.FrameLayout$LayoutParams(((int) (300.0 * v0)), ((int) (v0 * 50.0)));
         v1.gravity = 51;
         if (android.os.Build$VERSION.SDK_INT >= 11) {
            com.minifactory.minifactory.k.setAlpha(0.600000023842);
         }
         com.minifactory.minifactory.f.addView(com.minifactory.minifactory.k, v1);
      } else {
         com.minifactory.minifactory.r = new com.google.ads.AdView(this, com.google.ads.c.b, com.minifactory.minifactory.l);
         v0 = new android.widget.FrameLayout$LayoutParams(-2, -2);
         v0.gravity = 51;
         if (android.os.Build$VERSION.SDK_INT >= 11) {
            com.minifactory.minifactory.r.setAlpha(0.600000023842);
         }
         com.minifactory.minifactory.f.addView(com.minifactory.minifactory.r, v0);
         com.minifactory.minifactory.r.loadAd(new com.google.ads.b());
         com.minifactory.minifactory.k = com.minifactory.minifactory.r;
      }
      return;
   }
****************************************************************************************
Lcom/google/ads/AdView; b
   private java.util.Set b(String p7, android.content.Context p8, android.util.AttributeSet p9)
   {
      v0 = 0;
      this.a(p7, p8, p9, 0);
      v2 = new java.util.HashSet();
```

Figure 10: Added method body

6 Conclusions

This paper proposes *Repadiography*, a tool for visually comparing original and repackaged apps and for characterizing the impact of the alterations on the original app, i.e. localizing the added or altered code. In particular, our work demonstrates that modifications applied on repackaged apps rarely affects GUIs. Indeed, the alterations mainly consist of adding new methods, while changes to existing methods are quite rare.

Repadiography was developed to provide the community with a solution that can make more effective and efficient the triaging activities of malicious apps. However, while the current implementation of *Repadiography* represents an effective solution to deal with the triaging of native Android apps, it could be ineffective when processing cross-platform apps, since it is tailor-made for analyzing Android-specific code. For this reason, one of the directions of our future research is to understand how to adapt our visual approach to the case of cross-platform apps and make *Repadiography* able to process those kinds of Android apps. Additionally, we plan to semantically characterize the modifications, in terms of added or altered behaviors.

References

[1]Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2016, May). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (pp. 468-471). IEEE.

[2] androguard: Reverse engineering, malware analysis of android applications, https://github.com/androguard, (accessed on 13th of November 2019).

[3] androsim, https://code.google.com/archive/p/androguard/downloads (accessed on 13th of November 2019).

[4] AndroZoo, "RePack: A repository of repackaged Android apps", https://github.com/serval-snt-uni-lu/RePack (accessed on 13th of November 2019).

[5] apk2java, automatically decompile APK's using Docker. https://github.com/duo-labs/apk2java (accessed on 13th of November 2019).

[6] Berthome, P., Fecherolle, T., Guilloteau, N., &Lalande, J. F. (2012, August). Repackaging android applications for auditing access to private data. In *2012 Seventh International Conference on Availability, Reliability and Security* (pp. 388-396). IEEE.

[7] Crussell, J., Gibler, C., & Chen, H. (2012, September). Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security* (pp. 37-54). Springer, Berlin, Heidelberg.

[8] Desnos, A., &Gueguen, G. (2011). Android: From reversing to decompilation. In *Proceedings of the Black Hat Abu Dhabi*, 77-101.

[9] Gonzalez, H., Kadir, A. A., Stakhanova, N., Alzahrani, A. J., &Ghorbani, A. A. (2015, April). Exploring reverse engineering symptoms in Android apps. In *Proceedings of the Eighth European Workshop on System Security* (p. 7). ACM.

[10] Han, K. S., Lim, J. H., Kang, B., &Im, E. G. (2015). Malware analysis using visualized images and entropy graphs. *International Journal of Information Security*, *14*(1), 1-14.

[11] Harris,S."Simian-Similarity Analyser," https://www.harukizaemon.com/simian/ (accessed on 13th of November 2019).

[12] Kancherla, K., &Mukkamala, S. (2013, April). Image visualization based malware detection. In *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)* (pp. 40-44). IEEE.

[13] Li, L., Bissyande, T. F., & Klein, J. (2019). Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering*.

[14] Meng, G., Xue, Y., Xu, Z., Liu, Y., Zhang, J., & Narayanan, A. (2016, July). Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 306-317). ACM.

[15] Ni, S., Qian, Q., & Zhang, R. (2018). Malware identification using visualization images and deep learning. *Computers & Security*, *77*, 871-885.

[16] Shao, Y., Luo, X., Qian, C., Zhu, P., & Zhang, L. (2014, December). Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference* (pp. 56-65). ACM.

[17] Stasko, J. T., Brown, M. H., & Price, B. A. (1997). *Software Visualization*. MIT press.

[18] Sun, M., Li, M., &Lui, J. (2015, June). DroidEagle: seamless detection of visually similar Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (p. 9). ACM.

[19] Tian, K., Yao, D., Ryder, B. G., & Tan, G. (2016, May). Analysis of code heterogeneity for high-precision classification of repackaged malware. In *2016 IEEE Security and Privacy Workshops (SPW)* (pp. 262-271). IEEE.

[20] Ucci, D., Aniello, L., &Baldoni, R. (2018). Survey of machine learning techniques for malware analysis. *Computers & Security*.

[21] Wu, X., Zhang, D., Su, X., & Li, W. (2015). Detect repackaged android application based on http traffic similarity. *Security and Communication Networks*, *8*(13), 2257-2266.

[22] Yujian, L., & Bo, L. (2007). A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, *29*(6), 1091-1095.

[23] Zhang, F., Huang, H., Zhu, S., Wu, D., & Liu, P. (2014, July). ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks* (pp. 25-36). ACM.

[24] Zhou, W., Zhang, X., & Jiang, X. (2013, May). AppInk: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (pp. 1-12). ACM.

[25] Zhou, W., Zhou, Y., Grace, M., Jiang, X., & Zou, S. (2013, February). Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy* (pp. 185-196). ACM.

[26] Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012, February). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (pp. 317-326). ACM.