WLTDroid: Repackaging Detection Approach for Android Applications

Junxia Guo, Dongdong Liu, Rilian Zhao, and Zheng Li(✉)

College of Information Science and Technology,
Beijing University of Chemical Technology, Beijing, China
{gjxia,lizheng}@mail.buct.edu.cn

Abstract. Huge number of mobile applications are downloaded every year. These benefits promote the rapid development of the mobile application industry, especially Android applications for its openness. But, because of the low cost and good profit, Android application repackaging also developed quickly, which can make a kind of malware applications and publish them to the Android market. Therefore, in order to defend against this dangerous technology, repackaging detection technology has been continuously studied in recent years. Contrary to the repackaging detection technique, obfuscation techniques are used in the application repacking to avoid detection. This makes the effectiveness of many existing methods be affected. In this paper, we propose a novel approach based on Dynamic Whole Layout Tree extraction, that we call WLTDroid, which can avoid the interference of the layout file obfuscation. The experimental results show that the approach proposed in this paper can resist the obfuscation affect better than other repackaging detection methods. In addition, our approach is more accurate than the existing method RepDroid.

Keywords: Repackaging detection · Android application · Obfuscation · Layout view transfer

1 Introduction

With the rapid development of global smart phones market, smart phone applications (apps for short) have also been rapidly developed. In the year 2018, global App downloads exceeded 194 billion, up 35% from 2016, and consumer spending reached $101billion, up 75% from 2016 [1]. However, application repackaging is becoming a serious threat to the Android ecosystem, both at the security of app users and the revenue of app developers [11]. Zhou et al. [22] found out that among 1260 malicious apps, 1083 (86%) were spread by repackaging other apps. Another study [5] showed that 14% of apps' revenue and 10% of apps' users are illegally stolen by repackaged apps.

The work described in this paper is supported by the National Natural Science Foundation of China under Grant No. 61702029, No. 61872026 and No. 61672085.

G. Wang et al. (Eds.): WISA 2020, LNCS 12432, pp. 579–591, 2020.
https://doi.org/10.1007/978-3-030-60029-7_52

In recent years, many Android app repackaging detection methods have been proposed. They are divided into three kinds, which are code-based detection, resource-based detection, and UI-based detection. For the code-based methods, large amount of app codes make time consuming. Meanwhile, the code obfuscation methods can cause low code similarity. Researchers [4] found that original app and repackaged app have high similarity at the resource level. Thus, resource-based detection methods are proposed, which detect repackaged apps by analyzing the type and amount of the decompiled resource files (such as: xml files, pictures and audios, etc.). However, this kind of methods still have problems with resource obfuscation and redundant resources in repackaged app. Layout-based detection methods have been proposed because of the high similarity between the original app and the repackaged app at the layout level. However, layout files can be easily interfered with or confused during repackaging, resulting in a decrease in layout similarity.

Considering that the repackaging app need to maintain the same user interface and user interaction, so we assume that the purpose-based measures may be more effective. Because that no matter how the features change, the high similarity of user interface and user interaction never change. Therefore, in this paper, we designed a new approach called whole layout tree (WLT), which is a purpose-based measure. We use interface transfers to represent user interactions. By analyzing user interfaces and interface transfers of running apps, we can completely extract the external view layout information and the inner view transfer information. According to these two kinds of information, we can build an WLT for app repackaging detection. The evaluation criterion is that WLT similarity and repackaging possibility have a positive correlation. The primary contributions of this paper are summarized as follows:

- We design the Whole Layout Tree based on the Android application's interface layouts and transfers. It represents the application's external display information and internal operation logic information. Even with the obfuscation of code, resources and layout, it can still accurately express the application.
- We proposed a new repackaging detection method named WLTDroid for Android applications based on the WLT. WLTDroid is a purpose-based method which has strong resistance.

The rest of this paper is organized as follows: Sect. 2 describes design principles, definitions and features of WLT in detail. Section 3 shows the architecture and implementation of our repackaging detection method. In Sect. 4 we set up the experiments and evaluate the method proposed in this paper. Related work is briefly introduced in Sect. 5. Section 6 concludes our work.

2 Design of Whole Layout Tree

Basically, repackaging detection is using metric methods to find similar applications. To address the problems that existing methods are not resistant to

obfuscation technology and easily avoided by attackers, we propose a new measurement method based on Whole Layout Tree (WLT). It is a kind of expression for dynamic user interface layout of Android applications. We explain the detail of WLT, including the design principle, definition and features.

2.1 Design Principles of WLT

As we mentioned, repackaging detection methods need to have strong resistance and good accuracy. Thus, we design WLT with two design principles: strong resistance and high accuracy.

When using applications, users mainly interact with applications' interface instead of directly access applications' source code, resources or XML layout files. Therefore, users are more sensitive to applications' changes in interface than the changes in code, resources, XML layout files. To cheat users, repackaging attackers can modify everything except user interface and interface transfer. Based on above facts, WLT should be designed from the user interface and interface to get strong resistant.

The way how to denote and compare the similarity of the user interface and interface transfer will determine the accuracy of this kind of repackaging detection methods. In an Android application, the user interface is composed of widgets. Each widget has many attributes. Those widget attributes jointly determine the final display of the widgets in the user interface. Therefore, those widget information and widget attribute information should be properly recorded and used for ensure the accuracy.

2.2 Definition of WLT

The WLT is defined as follows[1].

Definition 1. ***Whole Layout Tree (WLT)***

The Whole Layout Tree (WLT) is a multi-fork tree with a hierarchical structure which is consists of a limited number of nodes and edges, denoted as T (W, R), where W is a finite set of widgets, R is a finite set of widgets' relationship.

The node set W is consisted of 3 types of nodes, which are "Container Node", "Display Node", and "Transfer Node". Container Nodes are those who have sub-widgets, such as FrameLayout and so on. Display Nodes are those who do not have any sub-widgets, such as TextView and so on. Transfer Nodes are those who can trigger interface transfer, such as Button and so on.

The edge set R is consisted of 2 kinds of relationships. (1) $iR<a,b>$, represents that the widget a and widget b are inclusion relationship, in which the widget a and b belong to the same interface view, and widget a is a layout widget which contains widget b. (2) $tR<a,b>$, represents that the widget a and widget

[1] An example figure is given at https://github.com/pro-resrc/figure. The app has four interface views, with 3 view transfers, 79 container nodes, 54 display nodes, 3 transfer nodes, 132 inclusion edges and 3 transfer edges.

Algorithm 1. Whole Layout Tree Generation

Input: HomeActivity(HomeView) from the testing APK
Output: a whole layout tree,WLT

```
dump view layout,HomeLayout from HomeActivity(HomeView)
extract core view layout,cLayout from HomeLayout by Filter
InitializationWLT(cLayout)
extract visual components,taskComponent from HomeLayout
while WLT is changing AND taskComponent is not empty do
    todoComponent=taskComponent.pop()
    transfer,newView=todoComponent.simulate()
    add transfer into WLT
    dump view layout,newLayout from newView
    extract core view layout,cNewLayout from newLayout by Filter
    updateWLT(cNewLayout)
    update taskComponent according to newLayout
return WLT
```

b are transfer relationship, in which the widget a and b do belong to two different interface views, widget a can trigger the response event and leads the transfer to the new view which widget b belongs to.

2.3 Features of WLT

According to the definition, a WLT should have following features:

1. WLT has only one root node.
2. The parent nodes of a WLT should be container nodes or transfer nodes.
3. The leaf nodes of a WLT should be display nodes.
4. A container node connects to one or more different types of child-nodes, those nodes are connected by inclusion edges.
5. A transfer node connects to a node which belongs to another interface view.
6. The order of the interface view that appears during a user operating the app is the order of the view layout in the WLT.

2.4 Generation of WLT

According to the extracted interface view layout xml file and the view transfer information, we start to generate WLT. Firstly, we initialize WLT with the home view's layout content. Then query the transfer node's information and connect the root node of the transferred new interface view as a child node to the transfer node with a transfer edge. Next, a transfer attribute is added to the transfer node for recording the related information. The above operation is repeated until the WLT no longer changes or there is no more transfer information. In addition, we make several optimization in WLT generation to make the process more efficient. Algorithm 1 shows the pseudocode of generating a WLT.

(a) When transfer to a interface view in the blacklist, only record the transfer information without the interface view information.

(b) If a interface view is already appeared in the WLT, it will not be added to the WLT again. We only keep the transfer information.

(c) When the task stack is empty or WLT does not change in 3 min, the WLT generation process ends.

3 Repackaging Detection Method Based-on WLT

Based on the definition and features of WLT, we design a similarity measurement method for the repackaging detection. In addition, we implement a repackaging detection method, which is named WLTDroid. We explain the details in this section.

3.1 Similarity Measurement of WLT

Hashing algorithms are widely used in file similarity comparison for its good performance. For comparing the similarity of WLT, we design a method based on the Context Triggered Piecewise Hashing (CTPH) [9]. However, the tree-structured texts of WLT can not be used for the CTPH algorithm. In addition, the information of the inclusion and transfer edges need to be calculated for the similarity measurement.

Therefore, we transform the WLT into a kind of sequential text information from two levels, which are interface layout and interface transition. First, we convert each interface layout individually, then convert the processed interface layout as a whole according to the order of interface transition by using a method similar to the binary tree preorder traversal algorithm.

For the inclusion edge information in WLT, we convert the inclusion information to the sequential information as text directly. For the transfer edge information in WLT, in addition to converting the transfer information into sequence information, a special weight needs to be given because of the special.

We have added the transfer attributes to all the transfer nodes as an identifier in the WLT generation step, so those two types of edges have been distinguished in the preprocessed WLT.

The CTPH algorithm first calculates the hash value of the testing text, then get the similarity score by calculating the minimum edit distance between the two hashes. The preprocessed WLT already contains the view layout information and the view transfer information, so the similarity score calculated by the CTPH algorithm represent the similarity of the apps. The formula for calculating the SimilarityScore is show in Formula 1, where pp means preprocess.

$$SimilarityScore = CTPH({WLT_1}^{pp}, {WLT_2}^{pp}) \tag{1}$$

The similarity score ranged from 0 to 100. A score of 0 means that two WLTs are completely dissimilar and the possibility of repackaging for these two apps is extremely low. A score of 100 means that two WLTs are exactly the same,

and those two apps most likely to be repackaged. We set a threshold θ, when the similarity score of two apps exceeds the threshold, it represents that the two apps have high similarity which need to do the signature verification for the final confirmation.

3.2 Framework of WLTDroid

The framework of repackaging detection method based on WLT is showed in the Fig. 1, which is consisted of four parts. They are data extraction, birthmark generation, similarity measurement and signature verification.

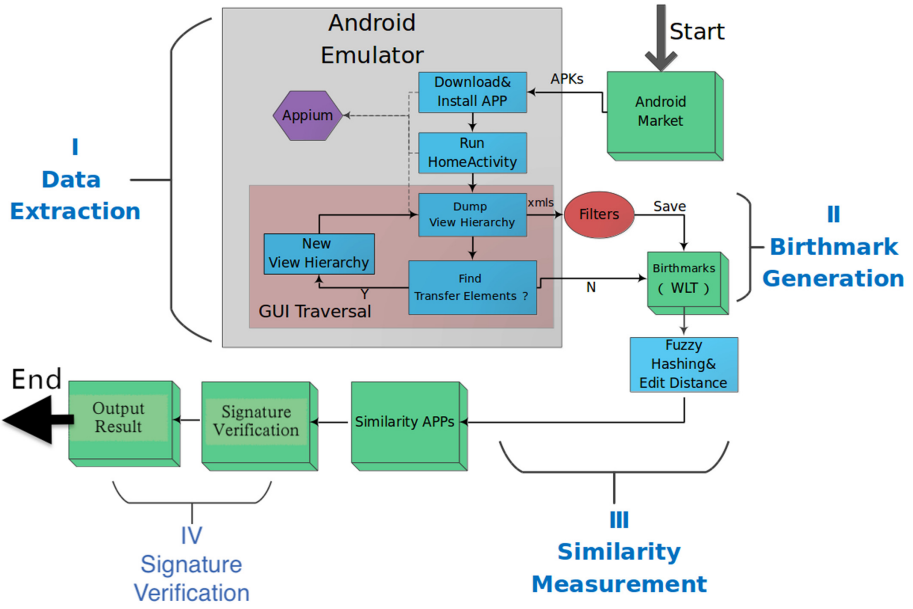

Fig. 1. The framework of WLTDroid

Firstly, WLTDroid dynamically extracts the view layout information of a testing app, and records the view transfer information. Testing app may jump to other apps (such as sharing button, etc.), whose view information does not belong to the original app, so we only save the view transfer information without further data extraction. In addition, for a special widget who names *WebView* we only retain the information of the native widgets.

However, the original layout contents contain redundant information, which do not have any effect on the structure of the interface layout and transfer, but will seriously affect the results of similarity measurement. Because that the attacker can easily add this kind of redundant information in the xml file for avoiding the detection. Therefore we need to filter such redundant information.

(a) Redundant UI widgets: Layout widgets that contain only one or no child are redundant. (b) Redundant attributes of widgets: There are some redundant attributes of the widgets. We found six redundancy attributes which are *text, package, bounds, resource-id, index, and content-desc*. For example, although the bounds attributes directly reflect the position of the widget, a subtle changes will not lead to a huge change in the view which may affect the results of our detection.

Next, generation module will construct WLT. Then, the similarity score between WLTs is calculated when the WLTs no longer change. Finally, if the score exceeds the setted threshold, signature verification will starts.

4 Experimental and Evaluation

Here we design two research questions to verify the effectiveness of our repackaging detection method.

RQ1: Does the WLTDroid has strong resistant when facing the obfuscated apps?

RQ2: How is the accuracy of the WLTDroid?

4.1 Dataset

The data sets used for the experiments are all from the Wandoujia app store (the largest third-party app store in China) [17] and F-Droid (an open source Android app store) [6]. We used three data sets named SR, S_1 and S_2. SR contains 30 Android apps from Wandoujia, which is used for resistance experiment. It includes the latest version of 30 apps randomly selected from Wandoujia's top 50 apps. We leverage Shengtao Yue's work (RepDroid) [12] to create S_1 and S_2 for Accuracy experiment. Unfortunately, we can only download 45 (41 of 45 apps can run) of 58 apps for S_1, and 115 (103 of 115 apps can run) of 125 apps for S_2 because that they are out of stock. The apps cover different fields, such as News, Reading, Education, etc. Totally, 190 apps are used in the experiments.

4.2 Resistance Testing

The resistance testing experiment adopts the current four major Android app packers in China (360, Tencent, Aliyun, Bangcle) to obfuscate or encrypt data set SR. Through code obfuscation, resource file encryption, and Dex file packing, we can simulate the real world Android repackaging environment. Then set the obfuscated app and the original app as a pair. In addition, we use XML files for obfuscation by adding redundant XML files to the decompiled 30 original apps to generate obfuscated apps.

We perform detection using WLTDroid, SUIDroid [11], AndroGuard [8], and FSquaDRA [20] for comparison with the existing methods. AndroGuard is a

code-based repackaging detection method. FSquaDRA is a code-and-resource-based repackaging detection method. SUIDroid is a layout-based repackaging detection method.

Using the 30 original apps and the apps obfuscated by above five packers, totally 150 comparison pairs are prepared for the experiment as shown in the upper part of Table 1. The results are shown in Fig. 2.

Table 1. Obfuscated or encrypted apps

Repackaging type	Data set	Original app count	Repackaged app count
360	$5 * SR$	$5 * 30(\text{Wandoujia}_{top50})$	30
Tencent			30
Aliyun			30
Bangcle			30
RedundantXML			30
IJiami	$4 * S_1$	30(Wandoujia)	30
AndroCrypt		$3 * 11$(F-droid)	11
FakeActivity			11
NestedLayout			11

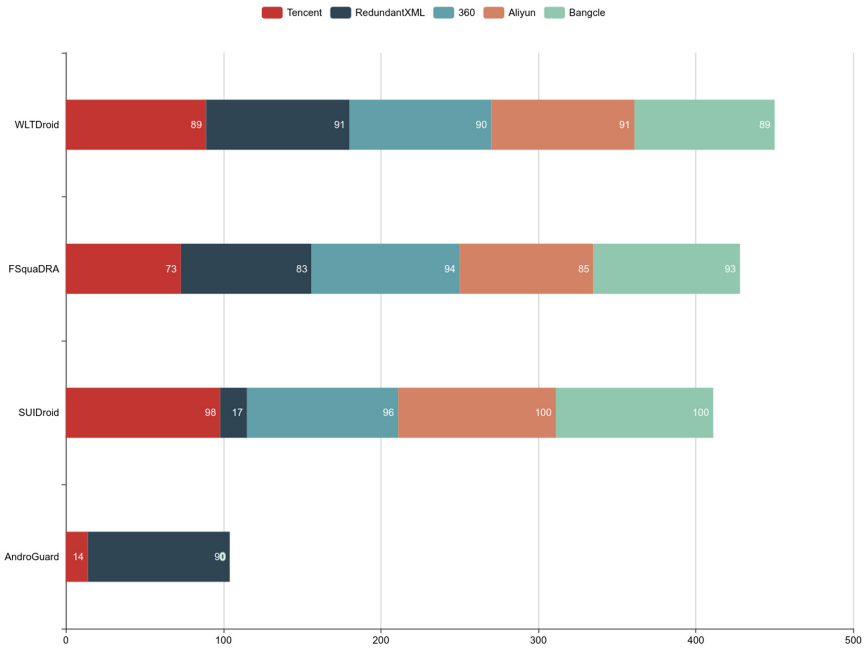

Fig. 2. Results of resistance testing

The horizontal axis represents similarity scores, also represents resistance of methods. The four stacked bars represent four repackaging detection methods. The stacks of five colors represent five types of app packers. The total similarity score of detections is 500 points. The scores from high to low are: 450 points for WLTDroid, 428 points for FSquaDRA, 411 points for SUIDroid, and 104 points for AndroGuard.

We can found that WLTDroid performs much better than AndroGuard. AndroGuard detects repackaged apps based on the decompiled source code. Once the source code was confused, it will be affected much. Except for the XML packer, AndroGuard almost cannot work for the other four packers. In contrast, SUIDroid performs poorly when detecting redundant XML packer. Only scores 17 points (WLTDroid scores 91 points). But, it performs well when detecting the other four types of app packers. That is because SUIDroid is based on analyzing the decompiled XML layout files. Once the attacker adds a large number of redundant XML files to the repackaged app, it will cause the repackaged app's schema layout to be completely different from the original app's schema layout, which makes SUIDroid get the wrong result. FSquaDRA has a better performance than AndroGuard and SUIDroid. FSquaDRA uses a combination of source code and resource files for detecting repackaged apps. Code obfuscation still affects FSquaDRA and may lead to instability during detection.

4.3 Accuracy Testing

We use S_1 and S_2 for Accuracy experiment. We compare the accuracy results of WLTDroid with RepDroid, because that RepDroid is also a repackaging detection technology based on interface layout and transition. The experiment is divided into two steps. Step one, we use S_1 to determine the thresholds of WLTDroid and RepDroid. Step two, we use S_2 to evaluate the counts of false positive(FP) and false positive rates(FPR) of WLTDroid and RepDroid, besides time consumption.

S_1 contains 41 apps that can run, of which 30 apps are from Wandoujia, 11 apps are from F-Droid. We use Ijiami to encrypt 30 Wandoujia's apps, and use three encryption tools: AndroCrypt, FakeActivity and NestedLayout, to encrypt 11 11 F-Droid apps. Totally 33 encrypted apps are used here. The encrypted apps and the original apps are combined as a pair. There are a total of 63 comparison pairs as shown in lower part of Table 1. S_2 contains 103 apps that can run from Wandoujia. The apps are compared with each other, and formed 5253 comparison pairs.

1) Determine the threshold

By comparing all similar scores of 63 comparison pairs, we select the lowest score as the best threshold. Because that 63 comparison pairs all belong to repackaged pairs, similarity scores represent the effectiveness of repackaging detection method, where the lowest similarity score represents the lower limit of

the method. The FNR and FPR of the method with this threshold are both to be at a lowest level. For WLTDroid, 59 similar scores are not less than 80 points, 34 similar scores are not less than 90 points, 4 similar scores are between 75 and 80 points, and the lowest similarity score is 75 points. For RepDroid, 59 similar scores are not less than 80, 45 similar scores are not less than 90, 4 similar scores are between 76 and 80, and the lowest similarity score is 76 points.

2) Evaluate FP, FPR, and time consumptions

We use WLTDroid and RepDroid to detect all the 5253 comparison pairs of apps. The detection results are shown in Fig. 3. We use logarithmic coordinates to make them more intuitive. The horizontal axis represents similar score sections. The vertical axis represents the number of comparison pairs belong to the section.

According to statistics, the distribution of similarity score is almost the same. More than half of the comparison pairs score 0. In detail, there are 6 comparison pairs with a value equal to or greater than 75 points for WLTDroid. Through manually check, we find that 3 pairs are actually repackaged apps, and the other 3 pairs belong to FP. The FPR of WLTDroid is 0.057%. There are 7 comparison pairs with a value equal to or greater than 76 points for RepDroid. Through manually check, there are 3 comparison pairs are repackaged apps, and the other 4 pairs belong to FP. The FPR of RepDroid is 0.076%.

In addition, we recorded the time consuming of two methods. The results are shown in Table 2. It shows that the most time consuming part is to generate the birthmark. However, there is 36% shorter for our method.

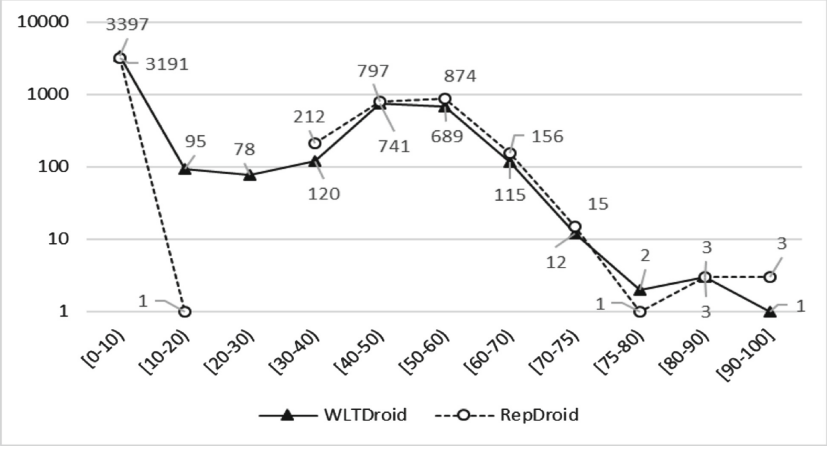

Fig. 3. Distribution of the similarity for WLTDroid and RepDroid

Table 2. Accuracy and Time-consuming for WLTDroid and RepDroid

		WLTDroid	RepDroid
3 * S1	θ	75	76
	FN	0	0
	FNR	0%	0%
2 * S2	FP	3	4
	FPR	0.057%	0.076%
2 * Time consumptions	Birthmark generation	10.02 min	15.74 min
	Similarity calculation	0.029 s	0.034 s

5 Related Work

Most of the studies in earlier years were based on decompiled source code. For example, DroidMOSS [21] handle the Dalvik bytecode to get the features of code by using the fuzzy hashing, then calculate the similarity of the two apps by using the edit distance algorithm [10]. DNADroid [3] generate program dependency graph as the program features based on the source code, then use the subgraph isomorphism algorithm to calculate the similarity of two dependency graphs. Steve Hanna et al. designed a repackaging detection system named Juxtapp, hashing the extracted features of testing apps' code for detection [7]. Jonathan Crussell et al. proposed AnDarwin, which combines the code information with other related information, for example the store information of testing app, for detection [2].

Later, many resource-based methods are proposed. FSquaDRA [20] decompile the testing app to extract all the code and resource files, then encode them with a hashing algorithm for calculating the similarity with Jaccard distance. ViewDroid [19] builds the view graph of the testing app by counting keywords in the app's source code for detection. ResDroid [14] calculates all the resource files contained in the apk file and transforms them into vectors for detection. Research [15] extracts the resource files, such as pictures, videos and so on to detecting repackage applications by using machine learning methods.

Recent years, the layout-based methods for repackaging detection are often used. DroidEagle [16] gets the layout files in the testing apk file, then uses the edit distance to calculate the similarity between individual layouts. SUIDroid [11] also gets the layout files from the testing apk file, then combine multiple layout files to get the app's schema layout for detection. RepDroid [12] proposes a layout group graph (LGG) based repackaging detection method, which built LGG from UI behaviors. Research [13] extracts user interfaces from mobile apps and analyzes the extracted screenshots to detect repackaging apps. RegionDroid [18] proposed an approach based on the app UI regions extracted from app's runtime UI traces.

The method in this paper is different from the above methods. This method dynamically extracts the interface layout and view transfer information of the

testing apps, then combines those two types of information to form a whole layout tree (WLT) for similarity detection.

6 Conclusion

We proposed a novel repackaging detection approach, WLTDroid, for Android applications by dynamically extracting the interface layout and transfer information of the testing apps. WLTDroid firstly builds the whole layout tree (WLT) for the apps, then calculate the similarity score. If the score exceeds the threshold which is set according to the experiment, signature verification will start to give the final result. The experimental results show that WLTDroid has strong resistance than the other three existing methods. Meanwhile, the accuracy of WLTDroid is good.

References

1. State of Mobile 2019. https://www.appannie.com/en/go/state-of-mobile-2019/?utm_source=AppStats2019&utm_medium=appdata&utm_campaign=AppAnnie
2. Crussell, J., Gibler, C., Chen, H.: AnDarwin: scalable detection of semantically similar android applications. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 182–199. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_11
3. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on android markets **81**(13), 2454–2456 (2012)
4. Gadyatskaya, O., Lezza, A.-L., Zhauniarovich, Y.: Evaluation of resource-based app repackaging detection in android. In: Brumley, B.B., Röning, J. (eds.) NordSec 2016. LNCS, vol. 10014, pp. 135–151. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47560-8_9
5. Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: AdRob: examining the landscape and impact of android application plagiarism. In: Proceedings of the International Conference on Mobile Systems, Applications, and Services, pp. 431–444 (2013)
6. Gultnieks, C.: f-droid.org/packages/ (2018)
7. 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). https://doi.org/10.1007/978-3-642-37300-8_4
8. Kim, J.-H., Im, E-.G.: AndroGuard: similarity analysis for android application binaries. J. Korean Inf. Sci. Soc. (2014)
9. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. **3**, 91–97 (2006)
10. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Phys. Doklady **10**, 707–710 (1966)
11. Lyu, F., Lin, Y., Yang, J., Zhou, J.: SUIDroid: an efficient hardening-resilient approach to android app clone detection. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 511–518. IEEE (2016)
12. Ma, J.: RepDroid: an automated tool for android application repackaging detection. In: ICPC (2017)

13. Malisa, L., Kostiainen, K., Och, M., Capkun, S.: Mobile application impersonation detection using dynamic user interface extraction. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 217–237. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_11
14. Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L.: 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 (2014)
15. Sibei, J., Lingyun, Y., Yi, Y., Yao, C., Purui, S., Dengguo, F.: An anti-obfuscation method for detecting similarity among android applications in large scale. J. Comput. Res. Dev. **51**(7), 1446 (2014)
16. Sun, M., Li, M., Lui, J.: 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 (2015)
17. Wang. www.wandoujia.com/apps (2018)
18. Yue, S., Sun, Q., Ma, J., Tao, X., Xu, C., Lu, J.: RegionDroid: a tool for detecting android application repackaging based on runtime UI region features. In: IEEE International Conference on Software Maintenance & Evolution (2018)
19. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: ViewDroid: towards obfuscation-resilient mobile application repackaging detection (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). https://doi.org/10.1007/978-3-662-43936-4_9
21. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: ACM Conference on Data and Application Security and Privacy, pp. 317–326 (2012)
22. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE (2012)