Evaluation Supplements

TABLE I: The detection rate of DroidMOSS-based technique

Threshold	Detection Rate
90	56.8%
80	65.3%
70	72.0%

I. Analysis and Comparison

A. DroidMOSS

Approach: We first implemented a prototype based on DroidMOSS [1] technique, which use the Fuzzy hashing method to detect the repackaged apps.

We use the Ground Truth samples to evaluate the accuracy and effectiveness of my system. There are 15,295 repackaged apps pairs. We use this hashing-based method to find their similarity.

Results: Based on our experimental result,the similar value ranges from 0 to 100. The value 100 means two apps have the same opcode sequence while value 0 means two apps are totally different. The result shows that there are 849 apps pairs without any similarity. That means by using this method, it cannot find 849 app pairs. By analyzing these apps, we find that most of repackaged apps are changed a lot by comparing the original ones. The left repackaged pairs similar values are range from 18 to 100. We set the similar value as 70 and get**the false negative rate is 23.8%**.

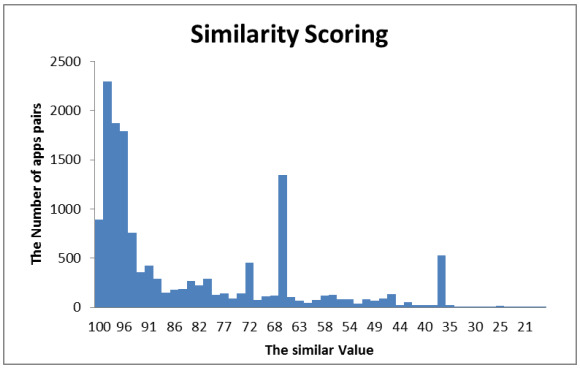


Fig. 1: Hashing-based method Detecting Result

According to the Figure 1, we can find 891 apps pairs have the same hash series, their similar value is 100, which means these apps are totally the same. 2298 application pairs own the similar value 99, which occupy the largest proportion among our real repackaged app pairs.

The detection rate can be seen from the Table I, the first column is the similar value, the value is larger, the similarity of two apps is higher.

Finding 1: The techniques based on DroidMOSS cannot find the repackaged app pairs, if the pairs has big difference of their app size. Took apk 00602B9E.apk and ABF68AB0.apk(we use the first 8 number of sha256 to represent an APK,the same below) as example. The first one is the original, and the second one is the repackaged. The size of the original app is 995.0 KB while the size of the repackaged app is 3.1MB. The size of the repackaged one is almost three times of the original one. The repackaged app adds too much new functions into the original one, so that there is a big difference between the original and the repackaged sequences.

Finding 2: This method cannot find repackaged app pairs, if their methods change the position, which lead to the changes of opcode instruction sequence. We also can see the other unmatched sample pair 01888A15.apk, 249C5562.apk. The first one is original app and the second one is the repackaged app. The size of the original one is 736.3KB and the size of the repackaged app is only the 242.7KB.Their size is very similar. The repackaged app deletes many functions from the original one and adds some new functions into the original app. This operation leads to the instruction sequence change a lot.We also found this method cannot handle 849 repackaged app pairs whose code are encrypted or obfuscated.

Conclusion: *When we set the threshold value as 70, the detection rate oly has 72%. If two apps have a obvious difference of their opcode sequence(add more functions into the original apps or change), this method loss its effectiveness. When we set the threshold value as 70, the false negative rate is 23.8%. Therefore, the precision is also not too high of this method. It is not resilient to code obfuscation or encryption.*

B. Juxtapp

Approach: We also implemented a prototype based on the technique of Juxtapp [2]. We set the k in the k-gram as 5 and the bitvector size m is 240,007. We want to verify the effectiveness of feature hash(Juxtapp based techniques). On the other hand, we want to try the cluster-based method to find the similar apps instead of pairwise comparison. We want to verify whether the efficiency can improve by using the cluster-based method.

Result: We use 30,000 free Android applications as our dataset. Among these apps, there are 12,595 real repackaged app pairs. The left apps we also downloaded from the Androzoo. We employ Jaccard similarity to filter some apps which cannot be the similar app pairs. The similarity threshold is set as t=0.9.

Finding 1: We find this method are really time-consuming. We spend nearly one week to get all the opcode sequence and three days the get the bitvector. Because the bitvector size is too large, we also spend about average 4987.46s to get the clustering result.

Finding 2: This approach is hard to find a appropriate parameters to get a ideal results. You had to try many

time to find a good experimental results. The bitvector size as large as possible and we also should consider the efficiency. We set different cluster numbers to divide different apps. We choose the purity as our judgment criteria. However, the experimental results also are not satisfactory. The specific result can be seen from the Table II. In this experiment, we set the k in the k-gram as 5 and the bitvector size m is 240,007.

TABLE II: The clustering result by using Juxtapp Technique

#	cluster cnt.	purity
1	500	0.351
2	700	0.348
3	800	0.347
4	900	0.346
5	1000	0.344
6	1100	0.3409
7	1200	0.3405

Conclusion: *According to the result,we found the choice of parameters directly affects our experimental results. Juxtapp was developed to identify repackaged apps from the Android App market. When we set the k as 5 and the bitvector size m as 240,007, the experimental detection rate is not good.*

RQ: What are the similarities and differences between the DroidMOSS and Juxtapp?

Motivation: For these two systems, we all know that they all extract the opcode sequence of Android apps as their feature. They adopt different methods to handle the code feature. In this RQ, we want to compare their performance and other indicators.

Approach: We mainly compare their processing time, computational complexity comparison algorithm and detection rate.

Result:1.DroidMOSS use Fuzzy Hashing method to get the fingerprint while Juxtapp use Feature Hashing to get the fingerprint. 2. DroidMOSS just compare the apps from Android official Market with the corresponding apps from third-party apps to identify repackaged apps. While Juxtapp using the clustering-based method to identify suspicious repackaged apps, Juxtapp is more scalable than DroidMOSS. 3. Juxtapp cost more computing resources. The fingerprint vector is much longer than that of DroidMOSS. 4. DroidMOSS just spend 2794s to process all apps in our dataset while Juxtapp cost nearly eight times than DroidMOSS.

1) Advantages and Disadvantages: **Advantages**: Both DroidMOSS and Juxtapp use the syntactic instruction sequence information as the birthmark and use hash-based method to find repackaged apps. The feature extraction is not complicated to implement and can find the lazy-attack repackaged apps quickly.

Disadvantages: We can conclude that if the repackaged app adds, deletes and modifies too much code of the original one, The opcode-based method will lose its effectiveness.Besides, they just consider the syntactic feature of apps and ignore the semantic features, therefore, these systems cannot handle with the code obfuscation problem.

C. DNADroid

Approach: According to our summary in section **??**, we find that there are many PDG-based methods finding the repackaged apps. In this section,we implemented a prototype based on DNADroid technique. DNADroid [3] employs the PDG as the birthmark, and use the subgraph isomorphism algorithm(VF2) to find the potential repackaged apps.

DNADroid use WALA to create the PDG and use VF2 to find similar apps. This system check whether an APP A is matching an APP B by using the equation 1.

$$sim_{A(B)} = \frac{\sum\limits_{f \epsilon A} |m(f)|}{\sum\limits_{f \epsilon A} |f|} \tag{1}$$

For each method f (excluding the methods in public libraries) in app A, $|f|$ indicates the number of nodes in this method's PDG. Find the best match of this PDG in B's PDGs and denote it as $m(f)$. The similarity score is the ratio between the sums of the $|f|$ values and $|m(f)|$ values. The comparison of each pair of apps A and B produces two similarity scores, $sim_{A(B)}$ and $sim_{B(A)}$. The $sim_{A(B)}$ is the percentage of code in A that is matched by code in B. In some situations, the value of $sim_{A(B)}$ and $sim_{B(A)}$ may be different. In PDG-based method, researchers choose the max similarity score of the pair to decide whether these apps are similar. Given that a malicious writer may insert or delete significant amount of code in abundance to the original app, which causes the decrease of overlap ratio of similar code. Actually, many malicious developers won't change some original code in order to give the users the same experience of the feel and look. The original parts still highly match the cloned parts. By using the larger similarity score can avoid the aforementioned problems. Therefore, in this system, the researches define two apps similar when at least one of the apps has a similarity score over 70% $(max(simA(B), simB(A)) > 70\%)$.

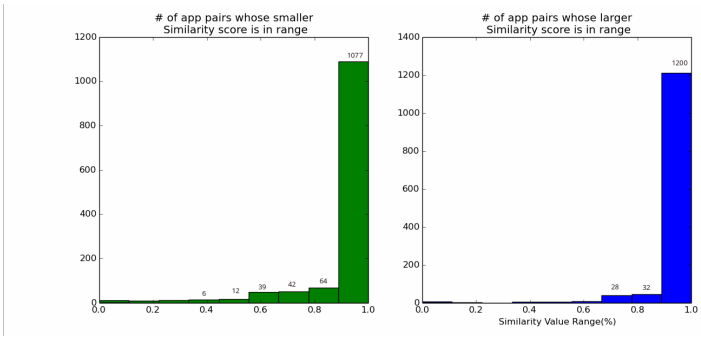


Fig. 2: Distribution of similarity score

Figure 2 shows the distributions of the similarity scores among a part of the pairs of apps which we randomly choose from the ground truth dataset. The analyzing result can be seen from the two diagram: Figure 2(a) uses the smaller similarity score in each app pair while Figure2(b) uses the larger score. When we set the similarity score threshold as 70%, using the larger similarity score can get more repackaged pairs. Figure2(a) shows that 1183 app pairs have similarity scores

TABLE III: The detection rate of DNADroid-based technique

Threshold	Detection Rate
90	91.5%
80	95.9%
70	97.7%

over 70%, and Figure 2 b shows that 1260 app pairs have the similarity scores above 70%. In sum, we use the larger similarity score can get high detection rate.

1) Filter Effectiveness: We try to use some methods to reduce the number of comparison so that it can improve the speed and scalability. According to some previous research such as DNADroid [3] adopt the lossless filter to delete the third-party libraries and some methods whose number of nodes is less than 10. Besides, it is necessary to delete some common public libraries. If a cluster of apps which contain many common libraries, the system will consider them as the similar apps. Without a doubt, it has a negative effect on our correctness and seriously slows down our processing rate. It also adopts lossy filter to discard method pairs which are impossible to match due to a great difference in the distribution of types of nodes in the two PDGs.

If we want to find the original app and repackaged app, the naive way need to require $O(n * m)$ method comparisons, where n and m are the number of methods of the app pair which we need to compare. The total nodes number is 37,859,026 of our dataset and after deleting the library classes, the nodes number decline down to 21,102,473. The library class filter excludes on average 44.24% of each app's classes. In lossless filter stage, we remove PDGs that are smaller than a specified size($< 5nodes$). A graph with smaller number of nodes is more likely isomorphic to other graphs. The lossless and lossy filters on average exclude 25.43% of the methods in an app.

2) Repackaged APP detection: We measure false positives and false negatives by using the dataset in section 2. When we use the filter to delete some apps, DNADroid can handle 8,773 repackaged app pairs in our dataset. When set the similar value threshold as 70%, it can get the false negative rate is 2.31%.

As for the false positives detection, we choose 81 app pairs from the ground truth and combine these app into other apps we crawl from the Android app markets. We run our system and detect whether we can find all the repackaged pairs. Therefore, the false positive rate is 0%.

Figure 3 shows the repackaged app pairs whose similar value distribution. The similar value equals to 1, it means the two apps are totally the same. The similar value equals to 0, it means two apps are totally different. It can be seen from the Figure 3,over 90% of app paris have more than

The detection rate can be seen from the Table III, the first column is the threshold of similar value, the second column is the detection rate.

It cost about 144min to compare 17,466 app pairs, the average comparison time is 0.49s for each app pair.

In our experiment, there are 19 repackaged app pairs getting a zero similar value by using this approach. We manually checked these 19 apps and found these repackaged app modified the data dependency relations of corresponding original apps. Therefore, their PDGs are not isomorphic.

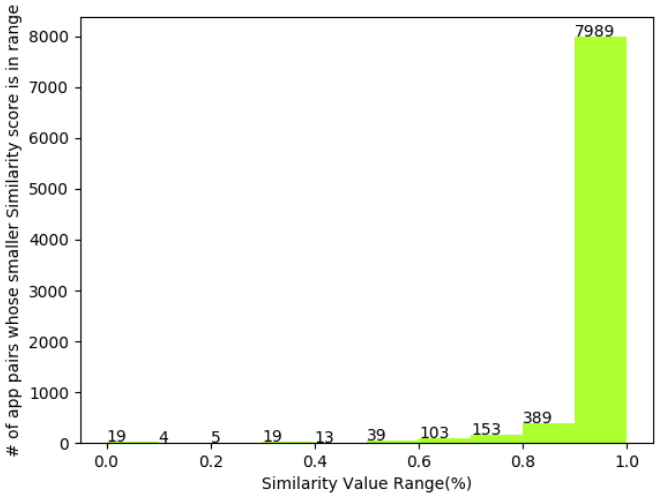


Fig. 3: The similar value histogram of DNADroid-based technique

Conclusion: We can find DNADroid-based method has a low false negative rate(2.31%) and false negative rate(0.0%). It has high precision and recall. When we set the similar value as 0.7, it can find over 97% of repackaged app pairs. This method has it limitations, using subgraph isomorphism algorithm(VF2), it has to consider the number of node in each graph. To improve the accuracy, it has to delete some graphs with node less than 5, it may delete some apps. Besides, some apks are repackaged app pairs while their PDGs are not isomorphic.

D. DroidSim Techniques

Approach: We first implement the system which refer to the DroidSim [4].We execute pair-wise comparison for each Android app and a similarity score will be calculated based on each app's signature. A signature contains a developer certificate and a set of CB-CFGs. The concrete process is as follows. Considering an app A, if we want to identify whether app B is repackaged from A, it will detect the two apps' identifiers. We neglects the app pairs with the same developer certificate. Otherwise, We calculates the similarity value for this pair based on the equation 2.

$$Sim(B) = \frac{\sum_{b \epsilon B} |b|}{min(|A|, |B|)} \quad (2)$$

We apply the VF2 subgraph isomorphism algorithm to measure the similarity between two CB-CFGs. In Equation 2, the similarity score of B is calculated in comparison with A. In the denominator, the symbol $|A|$ and $|B|$ represent the number of CB-CFGs in A and B, respectively. $min(|A|, |B|)$ means the minor value in $|A|$ and $|B|$. In numerator, the symbol b stands for each CFG in B. $|b|$ equals to 1 if b is subgraph isomorphic to any CFG in A and equals to 0 otherwise. In

TABLE IV: The detection rate of DroidSim-based technique

Threshold	Detection Rate
0.9	99.2%
0.8	99.4%
0.7	99.4%

practice, We removes CFGs that are smaller than 5 nodes, because small graphs are more likely to be the same by chance.

1) Performance: We compare the whole dataset which contains 15,297 app pairs. It costs about 159.86 hours to finish all operations. The average time to deal with each app is 37.73s. The shortest execution time is 0.16mms and the longest execution time is 18.94 hours. There are 7 apps execution time more than 10 hours. Besides, there are 6 apps execution time more than 2 hours. 98.8% of apps execution time is less than 100s. To some degree, the method of CFG-based method is efficient for our dataset.

2) Repackaged Apps detection: DroidSim can handle about 99.8% of apps in our dataset. There are 35 apps that DroidSim cannot handle. Because some of these apps just contain a small number of nodes in a CFGs. Some apps just use the third-party libs to develop their own apps, DroidSim delete these code and the left code cannot match other apps. We set the threshold value as 0.8, if the similar value is larger, app pairs have more common parts. The detection rate is 99.4% and the false negative rate is 0.63% while the false positive rate is 0.13%.

The detection rate can be seen from the Table IV, the first column is the threshold of similar value, the second column is the detection rate.

Conclusion: We can find DroidSim use the semantic features can also detect repackaged apps with code obfuscation. When set the threshold as 0.8, the detection rate is 99.4% and the false negative rate is 0.63% and the false positive rate is 0.13%. It has high detection rate and precision and recall. The detection rate can reach a high number, using this method can find repackaged app effectively.

E. 3D-CFG

Motivation: In this RQ, we want to verify the effectiveness of *centroid algorithm*.

Approach: According this paper [5], we also implement the *centroid* algorithm to verify its validity. Centroid algorithm has its unique method to judge the similarity between two apps.

Definition(3D-CFG) Each method can be denoted by a 3D-CFG. 3D-CFG is a special CFG in which each node is a basic block and has a unique coordinate.The coordinate is a vector$< x, y, z >$. x is the sequence number in a CFG. y is the number of outgoing edges of the node. z is the depth of loop in a CFG.

Definition (Centroid) A *centroid* $\vec{c}$ of 3D-CFG is a $vector < x, y, z, \varpi >$. $x = \frac{\Sigma_{e(p,q)\epsilon 3D-CFG}(\varpi_p x_p + \varpi_q x_q)}{\varpi}$,
$y = \frac{\Sigma_{e(p,q)\epsilon 3D-CFG}(\varpi_p y_p + \varpi_q y_q)}{\varpi}$,
$z = \frac{\Sigma_{e(p,q)\epsilon 3D-CFG}(\varpi_p z_p + \varpi_q z_q)}{\varpi}$
$\varpi = \Sigma_{e(p,q)\epsilon 3D-CFG(\varpi_p + \varpi_q)}$

ϖ is used to indicate the number of statements in a basic block. From the definition of the ϖ, we can find that changing the sequence of two statements of a basic block does not affect the ϖ. Adding or deleting a statement will make little effect on the centroid. Changing more statements will have more impacts on the centroid. In order to detect the repackaged apps, ϖ^i is introduced to denotes the number of invoke statement in a method. ϖ^i is define as "ϖ^i = ϖ in the basic block".

ϖ^i can be used to get the centroid $\vec{c^i}$. By using both $\vec{c}$ and $\vec{c^i}$ together, the false positive rate can decline. The centroid vector $< \vec{c}, \vec{c^i} >$ can represent a method.

In order to determine the similarity of APP, the authors introduced two decision rules: the method level similarity and Application Similarity Degree.

Method level Similarity

Definition *Centroid Difference Degree*(CDD) defines to measure the distance between two centroid. When $|\vec{c_1} - \vec{c_2}|$ is smaller, the difference between two methods will be less. When $|\vec{c_1} - \vec{c_2}|$ is larger, there will be more difference. The centroid difference degree between two centroids c and c' can be defined as :

$CDD(\vec{c}, \vec{c^i}) = max(\frac{|c_x - c'_x|}{c_x + c_x}, \frac{|c_y - c'_y|}{c_y + c_y}, \frac{|c_z - c'_z|}{c_z + c_z}, \frac{|\varpi - \varpi'|}{\varpi + \varpi'})$

CDD is the normalized distance for each dimension.

Given two methods $m < \vec{a}, \vec{a^i} >$ and $m' < \vec{b}, \vec{b^i} >$ *Method Difference Degree* MDD = $max(CDD(< \vec{a}, \vec{a^i} >), CDD(< \vec{b}, \vec{b^i} >))$, which can be used to compare the similarity between two methods. If the two methods are the same, MDD = 0 . If the two methods are totally different, MDD = 1.

Application Similarity Degree

Definition:Application Similarity Degree (ASD) can be get by using the method-level similarity. Application Similarity Degree can defined as $ASD(a_1, a_2) = \frac{a_{1,2}}{a_2}$. $|a_1|$ is the number of methods in a_1. $a_{1,2}$ is a set of methods. The set $a_{1,2}$ should meets the conditions: when a method m_i in the application A_2, $m \epsilon a_2$, there is at least the other method m' in the Application A_1, $m' \epsilon a_1$ which satisfies $MDD(m, m') \leq \sigma$. σ is a threshold of the method-level similarity.

1) Method-Level False Positive Rate: We can use the centroid to measure the similarity of method. We need manual check to verify whether they are real repackaged apps.

Actually, we do not need to pair-wise comparison of each method, we use the soot to create CFGs. We first compare the structure of CFG, if there is a big difference, we will not compare them. We randomly selected 500 apps from our data set and used them to measure the false positive rate at the method level. We set the threshold δ of method similarity, the false positive rate is 1.14%.

We find some CFGs are very small, their CFGs are all less than 4 nodes and have the same structure of CFG, the centroid are the same or very similar. In this situation, the false positive happens.

2) Accuracy in Detecting Repackaged Apps : According to the method-level similarity, we can find the repackaged apps. Based on the definition of Application Similarity Degree, we should set the a threshold σ to identify the repackaged apps. The value of σ can directly impacts how well the method separates repackaged apps. We the the threshold $\sigma = 0.8$ to detect app clones in our data set.

The false positive rate is 5.5% and the false negative rate is 0.36%. In this method, we just use the whitelist to filter

TABLE V: A comparison of detection result between DroidSim and Centroid algorithm

	DroidSim	Centroid algorithm
FNR	0.63%	0.36%
FPR	0.13%	5.5%

the third-party libraries. On one hand,the number of public libraries is limited, on the other hand some names of public libraries are obfuscated. We can not filter all the public libraries, in this situation, the false positives will occur.

Some apps have limited core code and the piggybacked code different from each other and the account for a large proportion of the whole source code. In this scenario, the false negative will occurs.

Conclusion: This algorithm has low false negative rate and false positive rate. At the same time, it is also has good efficiency and scalability.

RQ: What are the differences between the DroidSim prototype and centroid algorithm?

Motivation: For these two techniques, they all extract the CFG as their code feature to detect repackaged apps. We want to compare their differences and using scenario.

Approach: We mainly compare their false negative rate and false positive rate and performance.

Result: Table V shows the comparison result between the DroidSim and Centroid algorithm. According to the Table V, we can find DroidSim has higher FNR than that of Centroid algorithm while the result of FPR is totally opposite.

Conclusion: According to the result, we can get the number of node in CFGs has more effect on DroidSim method. Because DroidSim use CB-CFG to identify the repackaged apps, the code source granularity of DroidSim is larger than Centroid algorithm. Centroid algorithm consider the instructions in each basic block of CFGs, it is more sensitive than DroidSim can find the code modifications. If the invocation relations changes, it can bring more effects on Centroid algorithm.

F. ViewDroid

There are two main approaches (Static Analysis and Dynamic Analysis) to detect the repackaged app by using the view-based feature. According to the existing system, static analysis use the taint analysis technique to get the Activity transferring relation, some researchers name it *View Feature Graph*. As for the dynamic method, they usually install an app on a mobile phone or Android emulator, use the uiautomatic tool to get the running UI information.

RQ: Can ViewDroid-based prototype effectively detect the repackaged apps?

Motivation: In this RQ, we want to verify the effectiveness of ViewDroid.

Approach: We first implement the prototype that refer to the ViewDroid [6].We use static analysis to get the view feature graph(Activity invocation Graph) and use the VF2 algorithm to identify similar apps. Before find similar apps, we set several filters to reduce unnecessary comparisons.

1) Similarity Checker: We apply the VF2 [7] subgraph isomorphism algorithm to measure the similarity between two feature view graphs.

We extract the feature view graph for app A and app B, with n_A and n_B nodes in their feature view graphs, respectively. If apps A and B have m matched nodes, their similarity score can be calculated as *equ* 3.

$$similarityscore = \frac{m}{min(n_A, n_B)} \tag{3}$$

2) False Negative: We use our ground truth as our test samples and detect the efficiency of ViewDroid. There are 15,295 app pairs in our dataset, but we should do some preprocessing to remove some apps which cannot n meet the inspection requirements.

At the beginning, we filter some apps whose total number of activity is less than 2 and set a whitelist to delete common third-party libraries. If a graph with one node or two nodes could be isomorphic to many graphs, it will cause high false positives. Besides, we find some apps will use a lot of common third-party libraries, some of them even use more than 20 public libraries. If we cannot filter the common public libraries, it will introduce some noise in the detecting process. The accuracy might not be guaranteed.

We totally compare 10,635 app pairs. We set the similarity score threshold at 0.7. ViewDroid successfully finds 10,544 app pairs are similar, it fails to find 91 app pairs. The false negative rate is 0.85%. Besides, all the 10,544 repackaging cases have the similarity score 1.0. From this statistics, we can find that actually many malicious writer will not change the views of an app in most situation, in order to give the users, the same ”feel and look”. Besides, it will take significant amount of time to understand the logic of the code, most developer tend to add some functions into the original app or just add the third-party libraries into an original app. They seldom add a new activity into an original app or change the invoking relation between two activities.

We then manually check the 91 app pairs which ViewDroid cannot find them as the repackaged app pairs and find that some invoking relationships among different activities are changed or some repackaged apps own new activities which makes them cannot be isomorphic to original one. As we can see from the Figure 4 and Figure 5 . It an example of repackaged app pairs whose Activity relations have been changed.

From the Figure 4. we can see it a view graph of an original app 42A5B.apk. The package name is de.emomedia.diaetrezeptefree and the Main Activity is de.emomedia.plaetzchen.Startwebescreen. This original app possesses 16 activities. We can see the repackaged app 02674.apk from the Figure5.Comparing these two graphs, we can find some Activity translation relations have changed.

These two apps have the same number of activity and the same Main Activity. The repackaged app developer changes the packaged name to com.einszumanderen.plaetzchenappfree. In original app, the component RezeptlistenFragment can jump to InAPPBillingHelper, while in repackaged app, the component only can jump to Rezepteseite. In original app,

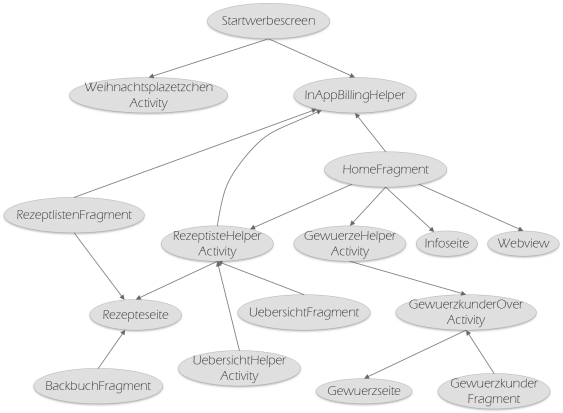


Fig. 4: A view graph of an original app

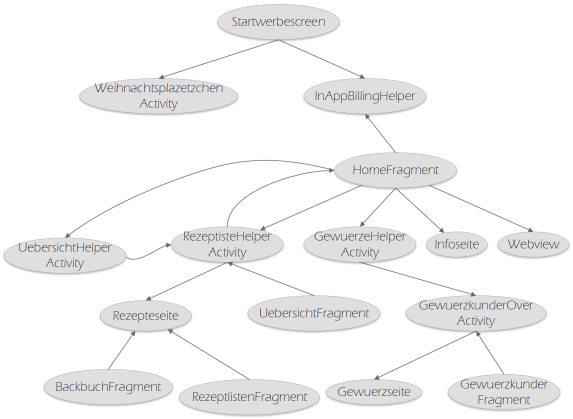


Fig. 5: A view graph of a repackaged app

the component HomeFragment cannot reach to the UebersichtHelperActivity, while the HomeFragment can reach to the UebersichtHelperActivity in repackaged app. The two graphs have the same nodes but the different edge relationships, so these two graphs cannot be isomorphic.

We check our detection results and all final results are right. But this method will be invalidated if the repackaged apps change the invoking relationships of activities or add some new activities into the original apps.

3) False Positive: In order to measure false positives, we choose 100 samples from our ground truth, and put all of them into 300 app clusters. These app have the similar size and activity number with our test sample. We also set the similarity score threshold at 0.7. After applying View-based method to detect the repackaged apps we manually check the detected pairs to ensure the accuracy. We set three criteria for manually checking: (1) We execute all the app pairs on an Android emulator to verify the similarity of their functionality; (2) We check the code, including the smali code, layout files and AndroidManifest.xml file and activity transition graphs the third-party libraries; (3) We also check the author certificate to make sure these similar apps from different developers. Only these criteria are satisfied, could we consider these app as the real repackaged cases.

Through our observation, we find that even two apps have the same activity number, if they are not repackaged cases, the view graph features, activity name, invoke functions generally are totally different. In our test samples, we do not find any false positive. Therefore, the false positive rate is 0%.

4) Performance and Efficiency: The execution time of view-based method for the exception samples are listed in TableVII. If we exclude nine apps with significant amount of views, it just takes 11.12s to handle all the left apps graph comparison. As for the 9 abnormal apps, their number accounts for 0.061% of the total number of test samples. We can see the graph comparison execution time from the TableVI of these abnormal apps. The longest comparison time takes about 6 hours while the shortest it also takes 9.49s.

TABLE VI: abnormal app pairs execution time of graph comparison

original apk	repackaged apk	executive time
2FA09CF23	93756E7F	21745.74s
331300B1	DE21EC0C	2946.94s
695F7447	FCAE02D9	2310.06s
331300B1	DE21EC0C	2220.76s
080B244C	06E2C75A	386.99s
6A63474C	27DC54AA	53.38s
4F1DF69C	37824CEA	25.90s
DE8BF84	2FCC1268	19.59s
16EB51A8	E1062323	7.50s

Because we use VF2 subgraph isomorphism algorithm to measure the similarity between two feature view graphs. Some graphs nodes are more than 100, it takes a lot of time to compare. Moreover, if many nodes point to one node at the same time, as can be seen from the Figure **??**(a) and (b), it will take great amount of time to compare the similarity of two View graphs, even if the number of nodes in a graph is not countless.

The specific execution time can be seen form the Table VII. There are three categories in Table VII, including code extraction, graph construction and graph comparison. Code Extraction includes using the reverse engineering tool apktool [8] to decompile an apk file to get the smali code and performing the static analysis on the smali code and AndroidManifest.xml file and removing the third-party libraries. We get the navigation relationship based on the Activity invocation relations. Therefore, it takes a lot of time to handle this part. From our evaluation of our dataset, it seems takes approximately 39s to conduct per app pair comparison. In fact, when applying the view-based methods to detect the similar apps, code extraction and view graph construction for each app is only performed once. We cut our dataset into different parts, so that we can conduct the three detection processes(Code Extraction,Graph Construction and Graph Comparison) at the same time. It remarkably reduces the overall processing time if we need to handle a large number of apps.

TABLE VII: The execution time of View-based method

	Code Extraction	Graph Construction	Graph Comparison
Max	1419s	10.42s	0.98s
Min	3.16s	14.11ms	0.104ms
Avg	38.82s	0.069s	1.05ms

Conclusion: 1)Due to the property of VF2 algorithm,

ViewDroid method cannot detect all the repackaged app pairs. If the number of node is too small, the filter will delete these apps. 2) This method has low false positive rate and false negative rate. 3) Over 90% app pairs have the same feature view graphs. 4)The modification of activity invocation relations can make this method loss it effectiveness. 5) If we want to find repackaged apps as much as possible, we had better not just use the UI feature as the birthmark.

G. *Resource-based Technique*

We want to design some experiments to answer the following questions and verify the accuracy of the previous study [9].

- **RQ :** Are there any difference between the repackaged app pairs and totally independent app pairs cite the statistical features.

1) Answer to RQ: To answer RQ, we set two experiments through the following settings.

- **E1:** We extract the 15 statistical features from the real 15,295 repackaged app pairs and calculate their similar value.
- **E2:** We extract the 15 statistical features from 175 totally different apps and we conduct pair wise comparison for them and finally get 15,225 app pairs. We calculate the similar value for these unrelated app pairs.

We extract 15 resources features and use Euclidean distance to calculate the similarity of two apps. The results of this experiment can be seen from Table VIII.

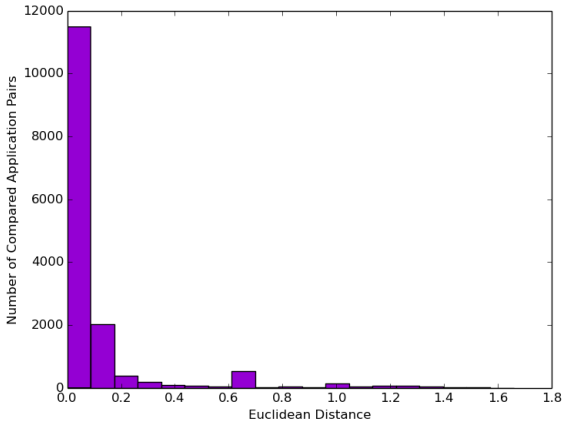
TABLE VIII: Results of the similar value between repackaged app pairs and unrelated app pairs

	repackaged pairs	unrelated pairs
the same	8,706	0
max distance	1.657	1.868
min distance	0	0.025
average	0.090	0.54
$< avg\%$	82.3%	63.1%

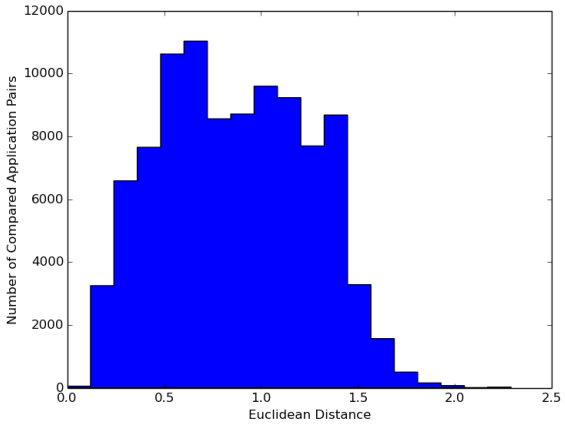
We can see from the Table VIII, the second and the third line stand for the maximum distance and minimum distance between two apps,respectively. The fourth line represents the average distance, and we can find that more than 82.3% of repackaged app pairs their similarity value of statistical features are smaller than average value and unrelated app pairs account for about 63.1%. There are 8,706 app pairs have the same 15 statistical features while no unrelated pairs have the same feature vector.

Result: According to the Figure 6, we can clearly see the differences between the repackaged app pairs and unrelated app pairs. Figure 6(a) shows the frequency distribution of the Euclidean Distance between the original apps and repackaged apps. We can see from the Figure 6(a) over 90% repackaged app pairs their feature vectors have closer distance while the unrelated app pairs seldom have very similar feature vectors, their distances between the different app pairs are also various. Figure 6 is a Gaussian distribution.

Conclusion: To sum up, there is a big difference between the repackaged app pairs and unrelated app pairs. Most



(a) original apps



(b) repackaged apps

Fig. 6: The difference between the repackaged app pairs' and unrelated app pairs' Histogram of *Euclidean Distance*

repackaged apps rarely change the resources of original apps. Based on the statistical features, we can roughly identify some repackaged apps.

ResDroid

Approach: We implement the prototype of resource-based techniques and verify their effectiveness.

ResDroid is one of the most important resource-based repackaged apps detection system, we first verify it effectiveness and then compare it with MassVet and ViewDroid.

2) Similarity: We randomly selected 11,785 app pairs to calculate the distance between two apps. We use the equation 4 to measure the similarity between two apps.

$$dis(s_1, s_2) = 1 - \frac{len(LCS(s_1, s_2))}{min(len(s_1, len_s 2))} \tag{4}$$

Suppose there are two apps, s_1 is the feature sequence we extract from the app A and s_2 is the feature sequence we extract from the app B. $dis(s_1, s_2)$ means the distance between two apps, when the value is larger, the apps have more difference. When the value is smaller, they have more similarity.

$len(s)$ denotes the length of sequence s. $LCS(s_1, s_2)$ is the longest common sequence of s_1 and s_2.

3) System Performance: To ensure the accuracy and rigor of this study, we set up a series of comparative tests to find a suitable weight. In order to figure out which one (the Activity structure(ATG feature) or the Event Handler) has more significant impact on repackaging detection, and we calculate the distance between two apps by using the ATG feature and Event Handler. Our data set is a real repackaged app pairs, we use the 11,785 app pairs as our ground truth.

In terms of the Activity sequence feature, the false negative rate is 1.1%. In order to compare the effectiveness fo our method, we also use edit distance to calculate the similarity and we get the false negative rate is 5.4%.

4) Exception Analysis: We manually analyze all the app pairs and summarize all reasons for the exception, and the specific reasons can be seen from the following list.

1. Some adversaries change original apps' package name or activity name. For example, one of the activities name of an original app is Bingo, while the corresponding activity's name is changed into Bingocrittercism.android.f. Some developers also try to change the package name of app. We find an app whose package name is com.programmi.skull2 but the package name of repackaged app is changed into com.programmi.invenio_best_music.

2. Some adversaries add or delete some activities into/from the original apps, which lead to the structure change of ATG.

3. We use the white list to filter some public libraries. However, the white list cannot includes all the public libraries. Besides, some libraries are obfuscated, we cannot filter these apps by using the packaged name.

4. Some apps are obfuscated, their package name and activity has been changed.

5. We also find the static analysis tools cannot parse some apps correctly. We cannot distill the complete ATG.

Compared with the feature of Activity sequence, we find the malicious authors or adversaries seldom change the view widgets. There are two main reasons: 1) it need more efforts to understand the logic of the original app and modify some interactive UI controllers. 2) if the appearance gap is too large between the original app and repackaged app, it may cause the users' suspicion.

In our data set, we find that there are 9 app pairs have greater difference comparing with the original apps. 147 app pairs has subtle changes. We manually analyze these apps and find the following reasons:

1) The 9 repackaged apps are repackaged from the different visions of the original apps. They are the same app while some original apps have already updated and they may add or delete some widgets.

2)The adversaries has made some subtle changes to the original apps.

In order to calculate the distance between the two apps, we should set the weight(the layout weight w_l and the event handler weight w_e).

Actually,according to the above description and analysis we can find that adversaries usually tend to change the Activity structure instead of the Event Handler. If $w_l > w_e$, more emphasis is paid to the layout features. Otherwise,the event handler features may be regards as more important. Based on the above series of experiments, we can easily speculate that if we set w_e a higher value, the false negative rate will be lower. In order to verify our speculation, we set up nine experiments, the results of the experiment as shown in Table IX. According to our experimental results, we can see the first group can get the smallest false negative rate.

TABLE IX: The influence of different weight on experiment results

weight	false negatives
w_l=0.1,w_e=0.9	9
w_l=0.2,w_e=0.8	25
w_l=0.3,w_e=0.7	97
w_l=0.4,w_e=0.6	228
w_l=0.5,w_e=0.5	368
w_l=0.6,w_e=0.4	456
w_l=0.7,w_e=0.3	561
w_l=0.8,w_e=0.2	613
w_l=0.9,w_e=0.1	664

The detection rate can be seen from the Table X, the first column is the threshold of similar value, the second column is the detection rate. The smaller the value is, the more similar the two apps are. When we set the edit distance as 0.01, the detection rate can reach to 98.1%. The false negative rate is 1.9% and false positive rate is 0.0%.

TABLE X: The detection rate of ResDroid-based technique

Threshold	Detection Rate
0.001	92%
0.01	98.1%
0.02	99.8%

Conclusion: In this section, we find ResDroid-based method also can find repackaged apps. The detection rate can reach about 98%. It also has low false positive rate(0.0%) and false negative rate(1.9%).

ViewDroid[6] applies the VF2 subgraph isomorphism algorithm to find similar apps. However, it first will filter some ATGs whose number of nodes of a graph is smaller than 2. Because a graph with a smaller number of nodes can match numerous graphs, it may add the false positive rate to some degree. There are 4,657 app pairs whose nodes number is small than 3 in our data set. ViewDroid first will filter these apps. For our data set, ViewDroid's coverage rate is about 69.6%. However, ResDroid cannot be affected by the number of activity node, the coverage rate is 100% for ResDroid.

An adversary sometime will add some functions into the components, it will change the call sequence and relations. Some malicious authors may add new activities into an original app. If two nodes transition relations is changed, two graphs may not isomorphism any more. We find that 91 app pairs are this situation in our date set. ViewDroid also cannot handle this situation. Therefore, the false negative rate of ViewDroid is 0.85% for our data set. Actually, in terms of the detecting method ViewDroid used, the similar score is 0 or 1 in most

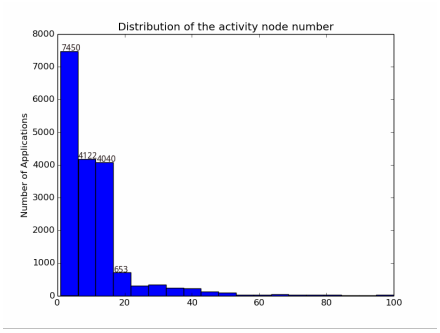


Fig. 7: Histogram of the number of activities' nodes of dataset

situations. In contrast, our system can deal with this problem very well.

5) ResDroid vs. Massvet: MassVet is a system using to catch Android malware, especially the malicious repackaged apps.

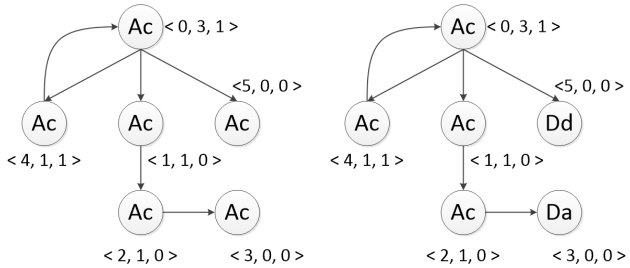
They use a new analysis method named DiffCom to finding unknown malicious behavior, without resorting to any heavyweight information-flow technique. A common analysis is used to identify apps sharing the similar view structure. When two apps own the similar view structure, a differential analysis will be conducted to find suspicous code segments(at the method level). An intersection analysis happens when a new app share a different view structures with existing apps.

The use a original algorithm named centroid to get the v-core and m-core of an app. A v-core is a geometric center of a view graph. A m-core is a geometric center of the control flow graph of a method. All v-cores and m-cores are sorted to enable a binary search during the vetting of a new app.

The researchers claim that their system can vet an app within 10 seconds at a low false detection rate and the performance outstriped all 54 scanners in VirusTotal

Figure 7 shows the frequency distribution histogram of the number of activities nodes. According to this Figure 7, we can find the distribution is skew to right, more than 93% of our test samples' number of activities nodes is less than 20. About 7,450 apps' the number of activities nodes is less than 5. About 8,200 apps' number of activities nodes range from 5 to 15. In our dataset, the view graphs with the number of nodes less than 2 are 2,216. In these apps, there are 407 view graphs with one node and 1809 view graphs with two nodes, respectively. However, MassVet has a serious limitation in finding similar view features, if the apps only has one or two activity nodes. These apps have the same centroid. For our data set, it means there are 2,216 apps have the same centroid. Therefore,MassVet is less effective for detecting these small apps.

To solve this problem, the researchers enrich the view graph with additional widgets. They add some widgets such as different kinds of Dialog into the view graphs. However, this will lead to two problems. 1) That not means some apps with few activities have the UI element Diglog. 2) This could cause some false positives. As can be seen from the Figure 8, it shows two different view graph of two apps with the different certificates. Figure 8(a) has five activities while Figure 8(b) has only three activities. MassVet sets a three dimensional vector $\vec{c} =< \alpha, \beta, \gamma >$ to indicate a node in a view graph. Here α is a sequence number assigned to each node in a view graph. The second element β is the out degree of the node. The third element γ is the number of "transition loops" the current node is involved. However, they have the same centroid value in this situation. In order to distinguish these two apps, MassVet has to conduct the next comparison.



(a) view graph 1 (b) view graph 2

Fig. 8: Two View-graph expamles
Ac:Activity; Da: AlertDialog; Dd: DatePickerDialog

MassVet avoid utilizing subgraph isomorphism algorithm to measure the similarity between two apps. The centroid algorithm does dramatically speed up the vetting speed. However, because many Android app's may contain some public libraries or some Android repackaged apps may have many different packages. MassVet use a whitelist to filter these public libraries, it hard to clear all public libraries,especially when some public library packaged are obfuscated by some tools. We can not totally filter these common libraries just depending on the package names. Therefore, an app may consist of several separated view subgraphs. We randomly select 7,500 apps as our test samples, and Table XI shows the explicit statistical results about the view subgraphs number distribution of apps. According to the Table XI, we can find approximately 85% of apps only with one view graph. However, the left apps own more than 2 graphs. To address this problem, MassVet try to use this formula $\Sigma_l |G_i(l)| / \sum_i |G_i| \geq \theta$ to judge whether the two apps are similar. $G_i(l = 1...m)$ is a subset which two apps share. G_i is a set of an app's view graphs. Two apps are similar in their UI structure when the threshold is larger than θ. If the graph G has m separated subgraphs and the graph T has n separated subgraphs. We should compare min(m,n) times and calculate the UI similar value. Due to the multi-entry feature of Android app, MassVet have to compare several times to find the UI similar app. Because of this characteristics of Android apps, the efficiency of MassVet is discounted to some extent.

TABLE XI: The view graph number distribution of apps

# of subgraph	1	2	3	4	5	6
# of app	6446	730	180	77	33	14
# of subgraph	7	8	10	11	12	
# of app	9	5	2	2	2	

Conclusion: 1. VF2 algorithm has some defects to handle apps with a small number of node in a graph. 2. The mod-

TABLE XII: The experimental result of FSquaDRA's detection rate with different threshold

Threshold	Detection Rate
0.9	23.2%
0.8	27.2%
0.7	30.0%
0.6	36.9%

ification of node relations can affect the MassVet detection result. 3. An apps can generate more than one CFGs, which possible bring more time consumption for MassVet detection.4 If attacker change all the resources in an app, it can affect the detection result of ResDroid.

H. FSquaDRA

Can FSquaDRA system effectively detect the repackaged apps?

Motivation: In this RQ, we want to verify the effectiveness and performance of FSquaDRA.

Approach: We run FSquaDRA and use the real 15,295 repackaged app pairs in our dataset to test it effectiveness. At the same time, we also extract the comparing time and feature extraction time.

Result:

1) detection rate: Table XII shows the detection rate of FSquaDRA with different threshold.Unlike ViewDroid, the detection result affect by the number of activity node, F-SquaDRA can coverage more apps. The threshold indicates the similar value. When the similar value is 1.0, which means the two apps are totally the same. While the similar value is 0.0, which means these two apps are totally different. According to the Table XII,we find the general detection rate is not good. When we set the threshold as 0.7, the detection rate just reach to 30.0%.

2) Performance: FSquaDRA processed all the repackaged app pairs just cost 2942.73s. It cost 184.32s to compare all the repackaged app pairs. The average comparing time is 58ms, the longest comparing time is 244ms and the shortest comparing time is 9ms. It takes 1215.52s to get the apk features to the memory.

Conclusion: We can find FSquaDRA just use resources as their birthmark, it can easily evade this method by add some junk resources into the apps. The false negative rate is rather high, which reached to 70%. The accuracy is not good but this system is efficient.

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

[1] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. ACM CODASPY*, 2012.

[2] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: a scalable system for detecting code reuse among android applications," in *Proc. DIMVA*, 2012.

[3] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Porc. ESORICS*, 2012.

[4] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *IFIP*, 2014.

[5] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. ICSE*, 2014.

[6] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM WiSec*, 2014.

[7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," in *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 2004.

[8] "android-apktool," https://code.google.com/p/android-apktool/.

[9] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.