NSDroid: Efficient Multi-classification of Android Malware using Neighborhood Signature in Local Function Call Graphs

Pengfei Liu
Central South University, pengfei@csu.edu.cn

Weiping Wang
Central South University

Xi Luo
Hunan Police Academy

Haodong Wang
Cleveland State University, H.WANG96@csuohio.edu

Chushu Liu
Central South University

NSDroid: efficient multi-classification of android malware using neighborhood signature in local function call graphs

Pengfei Liu · Weiping Wang · Xi Luo · Haodong Wang · Chushu Liu

Abstract
With the rapid development of mobile Internet, Android applications are used more and more in people's daily life. While bringing convenience and making people's life smarter, Android applications also face much serious security and privacy issues, e.g., information leakage and monetary loss caused by malware. Detection and classification of malware have thus attracted much research attention in recent years. Most current malware detection and classification approaches are based on graph-based similarity analysis (e.g., subgraph isomorphism), which is well known to be time-consuming, especially for large graphs. In this paper, we propose NSDroid, a time-efficient malware multi-classification approach based on neighborhood signature in local function call graphs (FCGs). NSDroid uses a approach based on neighborhood signature to calculate the similarity of different applications' FCGs, which is significantly faster than traditional approaches based on subgraph isomorphism. For each node in the FCGs, NSDroid uses a fixed-length neighborhood signature to capture the caller-callee relationship between different functions and combines neighborhood signatures of all nodes to form a vector that characterizes the function call relationship in the whole application. The generated signature vector is fed into a SVM-based classifier to determine which family the malware belongs to. Experimental results on large-scale benchmarks show that, compared with state-of-the-art solutions, NSDroid reduces average detection latency by nearly 20×, and meanwhile improves many evaluation index such as recall rate and others.

Keywords Android application · Neighborhood signature · Graph structure · Machine Learning

1 Introduction

1.1 Research status and motivation

Android has become the main operating system platform for smartphones in recent years. While bringing convenience to people's daily life, Android also faces serious security and privacy issues because of its openness. Tencent antivirus laboratory [1] reported that more than 14.94 million malicious Android Apps were detected in 2017. Many malicious applications have homologous code, because they usually use different variant of malicious code originating from the same malicious library or written by the same author. Determining the homology of malicious applications is thus helpful to track the organizational behavior of malicious applications, which has attracted much research attention in recent years [2–4].

Existing malware detection and classification approaches are usually based on graph similarity analysis [2–7]. They rely on graph algorithms with high time complexity (e.g.,

Table 1 Application type in each malicious family

Category	Family name (total number)	APK name of each family (number)
A	FakePlayer (6)	1
A	Gone (9)	1
B	ADRD (22)	14
B	AnserverBot (187)	9
B	Bgserv (9)	2
B	BaseBridge (120)	31
C	Asroot (8)	8
C	BeanBot (8)	8
C	DroidDream (16)	16
C	DroidDreamLight (46)	46

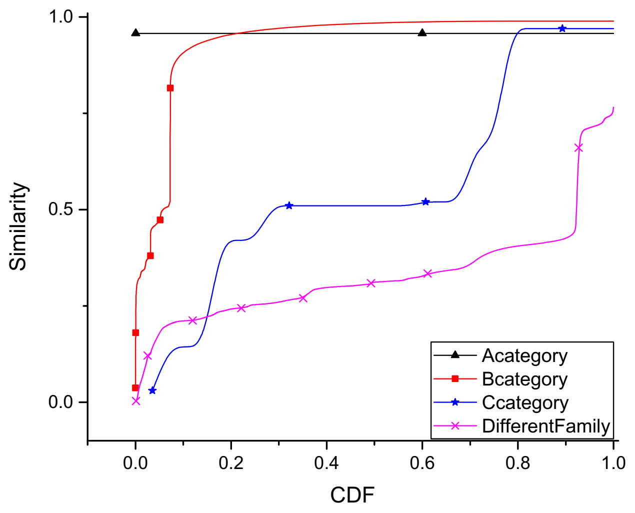


Fig. 1 Similarity of applications in different categories

subgraph isomorphism) to judge the similarity between the graph structure of different applications' code, which is time-consuming. Different types of graphs are extracted from the applications' code and used for malware detection and classification, such as the API dependency graph [2–5], the control flow graph [6], and the function call graph [7]. Subgraph similarity approach [8] constructs frequent subgraph to represent the common behaviors of malwares in the same family for familial classification of Android malware. In these methods, the code of an application is converted into a graph structure. Then the similarity among applications is determined by the subgraph isomorphism [9,10]. However, the subgraph isomorphism has been proved to be an NP-complete problem [11]. In practice, it is extremely time-consuming to match the subgraph structure by using the subgraph isomorphism. More time-efficient malware classification approach needs to be developed.

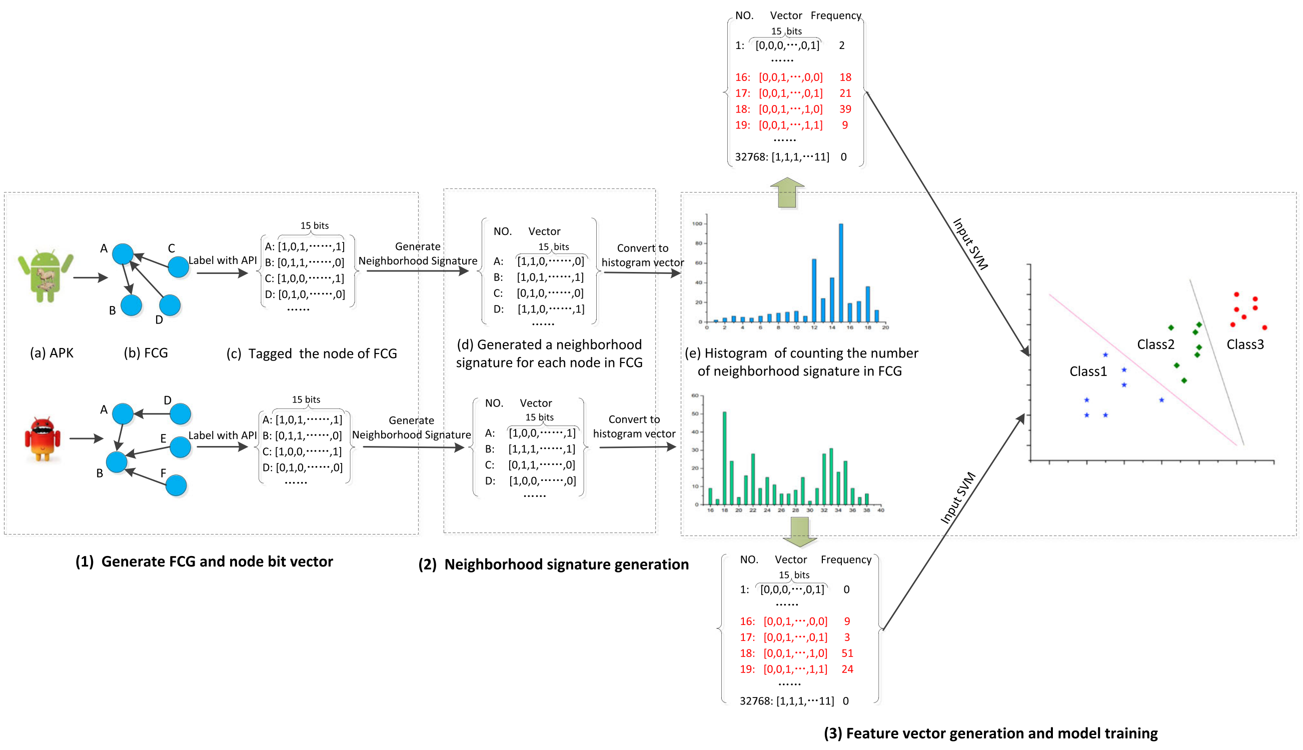
1.2 Code similarity between malicious applications

It is believed that the malware families have a certain level of homogeneity and thus the code of malicious applications from the same family has high similarity [7,12,13]. We have verified this hypothesis with a series of experiments. In the experiment, we randomly select a number of malicious applications from 10 families and conduct the analysis in the Malgenome Project sample [14]. As shown in Table 1, multiple malicious applications from the same family have the same application name. For example, there are 187 applications in the malicious family named "AnserverBot," but there are only nine distinct application names. These applications with the same application name possess a high-degree code similarity.

To further study the difference in code similarity between malicious applications from the same family and those from different families, we divide the 10 families into three categories: Category A contains families in which all the applications have the same name, category B contains families in which only a part of applications have the same name, while category C contains families in which all applications have different names. We use UltraCompare [15] to compare the pairwise similarity between the smali code of different applications in different categories.

We plot the cumulative distribution function (CDF) curve of applications in different categories in Fig. 1. It clearly shows that the applications in category A are highly similar to each other, with an average similarity rate higher than 95%. The applications in category B also have high similarity. In category B, more than 95% of applications have a similarity above 90%. Compared with category A and category B, the similarity among applications in category C is slightly low, but more than 70% of applications still have similarities higher than 50%. We also plot CDF curve of code similarity for applications from different families, marked with category D in Fig. 1. Applications from different families have a very low similarity: Only 10% of the applications have the similarity rate higher than 30%. Apparently, the applications from the same family are much more similar than those from different families.

Based on the above analysis, we propose NSDroid, an efficient malicious application multiple classification schemes. NSDroid constructs the function call graph of the Android application and leverages the function call local structure, which is represented by the neighborhood signature [16], to compare the similarity with the trained malicious application families. An support vector machine (SVM) model is used to complete the malicious application classification. We compare the classification performance and the operation efficiency between NSDroid and FalDroid [8]. FalDroid uses frequent subgraph similarity to classify Android fam-



(a) APK
(b) FCG
Label with API
(c) Tagged the node of FCG
Generate Neighborhood Signature
(d) Generated a neighborhood signature for each node in FCG
Convert to histogram vector
(e) Histogram of counting the number of neighborhood signature in FCG
Input SVM
Class1
Class2
Class3
(1) Generate FCG and node bit vector
(2) Neighborhood signature generation
(3) Feature vector generation and model training

Fig. 2 Overview of NSDroid

ilies. The results show that NSDroid significantly reduces the running time by 20 times and improves the classification accuracy and recall by 2.2% and 1.4%, respectively.

The rest of this paper is structured as follows. Section 2 describes the overall process of our detection method. Section 3 introduces the experiments and results. Limitations and related work are given in Sects. 4 and 5, respectively. Finally, Sect. 6 concludes the paper.

2 Classification method: NSDroid

2.1 NSDroid overview

NSDroid extracts the subgraph of FCG for measuring the similarity of applications. Different from the existing studies, we use the neighborhood signature to represent the subgraph and calculate the similarity between different graphs. The neighborhood signature of a node in the FCG contains the call dependency between the node and its neighborhood nodes (the direct caller and callee). We expect the neighborhood signature to uniquely represent the characteristics of a subgraph. Therefore, the signatures constructed from different subgraph should be different. If two different subgraphs obtain the same neighborhood signature, we denote it as a collision. To satisfy the above requirements, neighborhood signatures need to meet the following conditions: (1) simplicity in a calculation and (2) low collision probability.

Based on the similarity of the same family, the classification is based on the proportion of neighborhood signatures that are more similar for applications belonging to the same malicious family. In NSDroid, the SVM model is adopted as the classifier. All neighborhood signatures of function call graph are taken as the input of a trained SVM classifier. The SVM classifier is generated based on existing classification dataset training, and classification is carried out by calculating similarity. NSDroid's operation goes through three steps, as depicted in Fig. 2: (1) FCG generation, and each node is represented by a sensitive API vector, (2) generating a corresponding neighborhood signature for each node in the function call graph, and (3) the SVM model is used for classification.

In the rest of this section, we depicted each of these steps in detail.

2.2 FCG and node bit vector generation

The first step in NSDroid is to extract the application's function call graph. We use a general tool of the Android framework, androgexf [17], to extract the function call graph in order to ensure the preservation of the call relationship. The androgexf tool has been widely used to generate FCG

Table 2 System APIs categories and their corresponding positions

API category	Position	API category	Position
Location	1	Calendar	9
Network	2	Settings	10
Account	3	Browser	11
File	4	nfc	12
Bluetooth	5	Phone state	13
Database	6	Audio	14
Email	7	Contact	15
sms/mms	8		

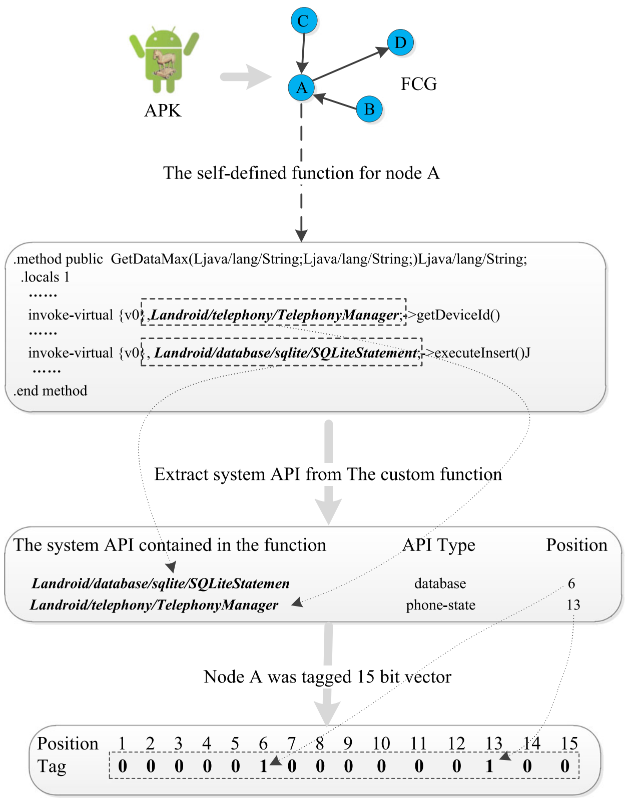
from APKs [2,4,18,19], and it can recover precise function call relationship at the bytecode level.

We use a tool named androgexf to generate the call graph, in which the node and the edge represent the method and the call relations between two methods, respectively. Particularly, the function call graph is a directed graph. As depicted in Fig. 2, edge(A,B) indicates that A calls B in this program. Taking the sensitive APIs called by these methods into consideration, we use a label of 15 bit to mark these nodes based on the 15 types of sensitive APIs in Table 2. In detail, when a node calls a special type sensitive APIs, the bit corresponds to that type is set as 1. Because the APIs in the same type behave similarly, we consider two nodes with the same label (e.g., in the same type) functions similarly.

As shown in Fig. 3, we give a tagging example of a node with sensitive APIs categories. In Fig. 3, FCG corresponds to a subgraph consisting of node A in a function call graph and its nodes that have a direct call relationship with A. This subgraph indicates that C, D call A, and A calls B. To mark Node A, we extract two sensitive API (marked in bold) from Node A, respectively: *Landroid/telephony/TelephonyManager* and *Landroid/-database/sqlite/SQLiteStatement*. The first API belongs to the category of a database, which corresponds to the 6th bit in the vector. The second API belongs to the category of phone state, which corresponds to the 13th bit in the vector. We tag the 6th bit and the 13th bit of the 15-bit vector as 1 and the other bits as 0. So the corresponding tagged value of node A in Fig. 3 is 000001000000100.

2.3 Neighbor signatures generation and analysis

Now we describe the details of generating the neighborhood signatures for nodes in the FCG. The neighborhood signature of a node is obtained by XORing its label with all its neighborhoods' labels. To better distinguish the relationships of the caller from callee and avoid the collisions, we first adopt the circular shift of these labels. In particular, the node itself shift left a bit circularly. As for its neighborhoods, the node that caller shift left two bits circularly while the nodes that


APK
C
D
A
B
FCG
The self-defined function for node A
Extract system API from The custom function
The system API contained in the function
API Type
Position
Landroid/database/sqlite/SQLiteStatemen
database
6
Landroid/telephony/TelephonyManager
phone-state
13
Node A was tagged 15 bit vector
Position 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Tag 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0

```
.method public  GetDataMax(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
 .locals 1
   ......
  invoke-virtual {v0},Landroid/telephony/TelephonyManager;->getDeviceId()
   ......
  invoke-virtual {v0}, Landroid/database/sqlite/SQLiteStatement;->executeInsert()J
   ......
.end method
```

Fig. 3 Tagging example of a node with sensitive APIs categories

called shift left three bits circularly. The caller and callee of a node are circularly shifted by different bits because this can help distinguish different function invocation directions. On the other hand, the method of circular shift will bring side effects, but the impact is small. The details are described in Sect. 2.3.2.

Now we have realized the structural representation of the call relationship between nodes in the function call graph through different circular shifts left. In order to map node structure information directly connected to a node to that node, we introduce XOR operation. The neighborhood signature value of the node is generated by XOR marking the node and its directly connected node. We expect the neighborhood signature to uniquely represent the characteristics of a subgraph. Therefore, the neighborhood signatures constructed from different subgraphs should be different. If two different subgraphs obtain the same neighborhood signature, we denote it as a collision. To make a neighborhood signature uniquely represents a subgraph, the collision probability should be small. The details are described in Sect. 2.3.3.

2.3.1 Neighbor signature generation

The generation of neighborhood signatures mainly consists of two parts: circular shift left for the label of a node and XOR with neighborhood node's labels. The specific opera-

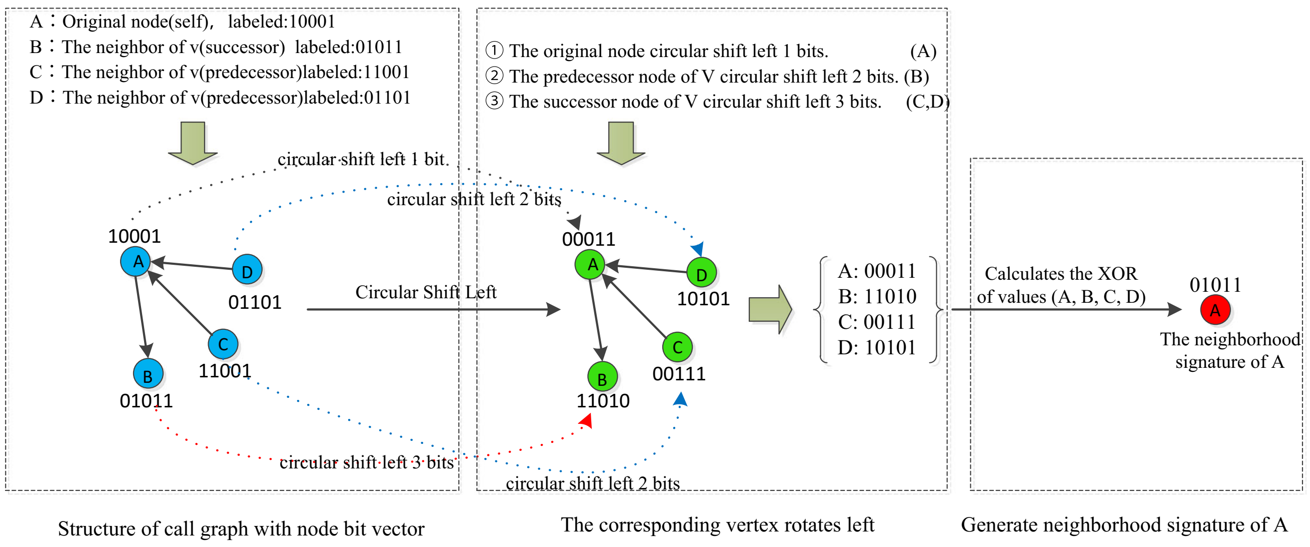

A：Original node(self), labeled:10001
B：The neighbor of v(successor) labeled:01011
C：The neighbor of v(predecessor)labeled:11001
D：The neighbor of v(predecessor)labeled:01101
① The original node circular shift left 1 bits. (A)
② The predecessor node of V circular shift left 2 bits. (B)
③ The successor node of V circular shift left 3 bits. (C,D)
circular shift left 1 bit
circular shift left 2 bits
circular shift left 3 bits
circular shift left 2 bits
Circular Shift Left
10001
01101
11001
01011
00011
10101
00111
11010
A: 00011
B: 11010
C: 00111
D: 10101
Calculates the XOR of values (A, B, C, D)
01011
The neighborhood signature of A
Structure of call graph with node bit vector
The corresponding vertex rotates left
Generate neighborhood signature of A

Fig. 4 An example of generating neighborhood signature of original node A

tion process is as follows. The node itself shift left by one bit circularly. As for its neighborhoods, the node invoking the current node shift left two bits circularly, while the node invoked by the current node shift left three bits circularly. Once the circular shift left is completed, we do the bitwise XOR on the resulting bit vectors, including the given node, its predecessors and its successors. The generated XOR value is the neighborhood signature of the specific node.

In the following, we use an example to illustrate the neighborhood signature generation process. Given a function call graph shown in Fig. 4, we calculate the neighborhood signature value of node A. For simplicity of description, we tag the vector of the node in the function call graph as 5 bits.

As shown in Fig. 4, node A has its vector 10001. A's predecessors, C and D, have their vectors 11001 and 01101, respectively. A's successor, B, has its vector 01011. In the circular shift shown in Fig. 4, node A circularly shifts 1 bit to the left and gets 00011. Both C and D circularly shift 2 bits to the left and get 00111 and 10101, respectively. Node B shifts 3 bits to the left and gets 01011. Once the circular shift is completed, we apply XOR operation and obtain the bit vector 01011 as the neighborhood signature for node A.

2.3.2 Explanation of circular shift left

In order to distinguish the same function but different call relationship, we adopt the circular shift left. As shown in Fig. 5, because of employing this method, neighborhood signatures of the eight types of subgraph structures are different.

On the other hand, the method of circular shift and XOR can be used to distinguish different call relations of the same node. But we observe that it may bring side effects, which will make some graphs with different structures generate the

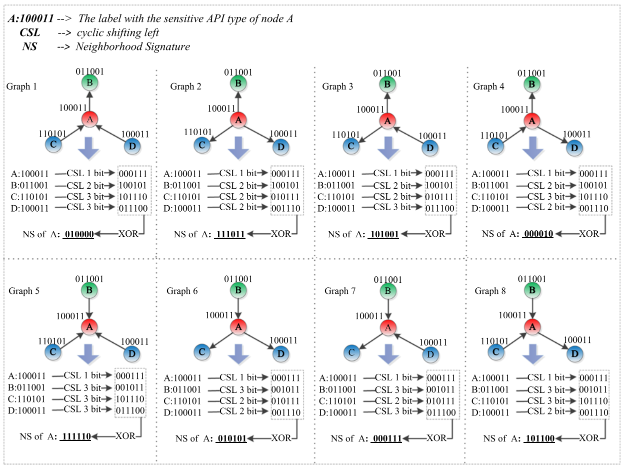


Fig. 5 Reason of different circular shift

same neighbor signature. As shown in Fig. 6, although the six kinds of graphs are different, because of the circular shift operation, the neighbor signatures of the central node are all the same, which is called "circular shift collision."

In order to verify the possibility of the "circular shift collision," we circularly shifted a function label to left and to right 1 or 2 bits. Then we observed the appearance of new labels in real samples. We have made statistics on the node labels in the 11038 APK in the dataset [20], and we have obtained 203 different node labels in the sample. We selected the top 30 node labels that appeared most frequently as the experimental labels.

We respectively performed the four operations with the selected experimental label by shifting to left 1 or 2 bits, or shifting to right 1 or 2 bits circularly. Then we observed

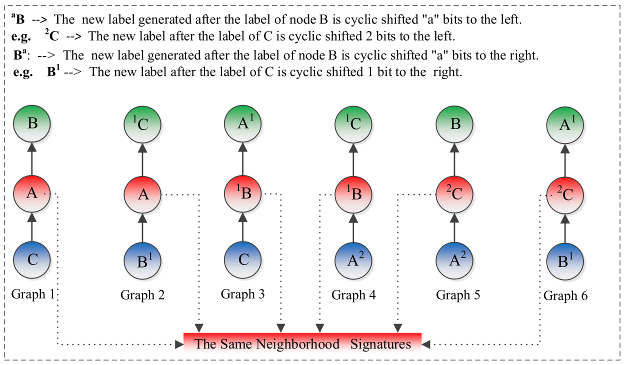

^{a}B --> The new label generated after the label of node B is cyclic shifted "a" bits to the left.
e.g. ^{2}C --> The new label after the label of C is cyclic shifted 2 bits to the left.
B^{a}: --> The new label generated after the label of node B is cyclic shifted "a" bits to the right.
e.g. B^{1} --> The new label after the label of C is cyclic shifted 1 bit to the right.
B
A
C
Graph 1
^{1}C
A
B^{1}
Graph 2
A^{1}
^{1}B
C
Graph 3
^{1}C
^{1}B
A^{2}
Graph 4
B
^{2}C
A^{2}
Graph 5
A^{1}
^{2}C
B^{1}
Graph 6
The Same Neighborhood Signatures

Fig. 6 Circular shift collision occurs in different graph

Table 3 The real situation of the experimental label after four operations

Operation	Number of real labels after operation
Circular shift left 1 bit	0
Circular shift left 2 bit	4
Circular shift right 1 bit	6
Circular shift right 2 bit	0

Table 4 Collision distribution of neighborhood signature

ID	NS	Quantity of NS	Distribution of NS
1	100000000001001	3	√
2	001000000000010	5	(1,1,1,2)
3	000100000000001	3	√
4	000000000010001	94	(24,20,16,34)
5	000000000100000	233	(1,3,229)
6	000000000000100	591	(7,5,3,576)
7	000000000000111	90	(1,2,87)
8	000000010000010	7	√
9	100000000000010	372	(5,2,3,362)
10	000100000000011	1	√
11	100000000101010	1	√
●	●	●	●
K	●	n	$(n_l, n_2, n_3, \ldots n_m)$
●	●	●	●
295	000000110000001	1	√
296	000000000100001	5	(3,2)
297	000001000000000	919	(9,3,1,906)

ID: Serial number of neighborhood signature
NS: Neighborhood signature. ●: Ellipsis
Quantity of NS: Total quantity of the NS
√: This kind of NS structure is all the same
Distribution of NS: Quantity of the same structure in NS

whether the label that appeared after the operation really existed. As shown in Table 3.

Table 3 shows the number of labels that actually exist after the circular shift. Table 1 shows that the number of labels that actually exist after the shifting is very small, especially the experimental label which was shifted to left 1 bit or 2 bits did not exist in the original sample. At the same time, even if the label after the circular shift exists, it still needs to satisfy the calling relationship to cause "circular shift collision." Therefore, although we cannot give a rigorous proof of probability, we can conclude that the impact of this collision is small.

2.3.3 Analysis of the neighborhood Signature's Collision

To guarantee the uniqueness of the neighborhood signatures, we have analyzed the collision probability of the neighborhood signature in this section.

We took 11038 APKs from the dataset [20] as the experimental samples. We extracted a total of 853,995 neighbor signatures from the function call graphs generated in these APKs. These neighbor signatures were divided into 297 different types. Table 4 shows the type of neighborhood signatures and the collision data distribution we extracted from the dataset.

As shown in Table 4, the column "NS" corresponds to the neighbor signature represented by a 15-bit vector. The column "Quantity of NS" indicates the number of times the neighbor signature appears in the entire dataset. For example, the first row of Quantity of NS is 3, which means that the neighbor signature with the value of 100000000001001 appears three times in the entire dataset. "Distribution of NS" indicates that the neighbor signature corresponds to the quantity distribution of different structures. "√" indicates that the corresponding substructures are all the same. $(n_l, n_2, n_3, \ldots n_m)$ indicates that there are m different substructures in the neighbor signature, and nm indicates the number corresponding to the mth substructure. For example, (1, 1, 1, 2) corresponding to the second line indicates that the neighbor signature (001000000000010) has four different substructures, and the numbers are 1, 1, 1, and 2.

Definition 1 We define the collision rate of neighbor signatures (the neighbor signature values are the same but the substructures are different) as P.

$$P = \sum_{i=1}^{K} \left(\frac{C_{w_i}^2 - \left(C_{w_{i,1}}^2 + C_{w_{i,2}}^2 + C_{w_{i,3}}^2 + \cdots + C_{w_{i,m}}^2 \right)}{C_{w_i}^2} * \frac{w_i}{N_{\text{Total}}} \right)$$

As shown in Definition 1, because the ID of Table1 is 297, the value of K is 297. The "N_{Total}" indicates that the total number of extracted neighbor signature is 853,995, and the "w_i" indicates the number contained in each signature of 297 different neighbor signatures. "$w_{i,1}, w_{i,2}, w_{i,3}, \ldots w_{i,m}$" represents the number of different structures each neighbor signature contains. Take the value in the 4th row as an example, the number of NS(000000000010001) is 94, and the number of different

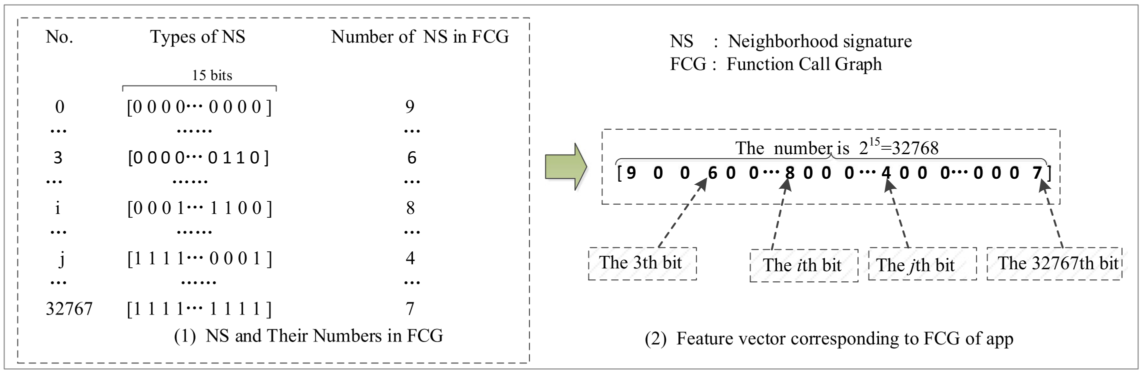


Fig. 7 Feature vector generation

substructures is 24, 20, 16, 34. Then $w_4 = 94, w_{4,1} = 24, w_{4,2} = 20, w_{4,3} = 16$, and $w_{4,4} = 34$. These values are substituted into the above formula, and the probability P of the collision of neighbor signatures in this data set is calculated to be 7.072%. As follows:

$$
\begin{aligned}
P &= \sum_{i=1}^{297} \left(\frac{C_{w_i}^2 - \left(C_{w_{i,1}}^2 + C_{w_{i,2}}^2 + C_{w_{i,3}}^2 + \cdots + C_{w_{i,m}}^2\right)}{C_{w_i}^2} * \frac{w_i}{N_{\text{Total}}} \right) \\
&= \frac{C_5^2 - C_2^2}{C_5^2} * \frac{5}{853995} + \frac{C_{94}^2 - (C_{24}^2 + C_{20}^2 + C_{16}^2 + C_{34}^2)}{C_{94}^2} \\
&\quad * \frac{94}{853995} + \cdots + \frac{C_n^2 - (C_{n_1}^2 + C_{n_2}^2 + C_{n_3}^2 + \cdots + C_{n_n}^2)}{C_n^2} \\
&\quad + \cdots + \frac{C_{919}^2 - (C_9^2 + C_3^2 + C_{96}^2)}{C_{94}^2} * \frac{919}{853995} = 7.072\%
\end{aligned}
$$

We also analyze whether the collision impacts the malware family classification or not. As depicted in Fig. 1 in our manuscript, we found that the code similarity of the same family was very high. Moreover, in different family the similarity was much lower than that of the same family. For example, the similarity of family A and B is between 90 and 95%. And the applications with their similarity threshold value up to 50% accounts for 70% in family C As a consequence, two applications of the same family are more likely to have the same neighbor signature. In this case, we can regard the collision probability of our datasets as noise. The classification result will not be impacted by these collision signatures as well.

2.4 Feature vector generation

In order to classify malicious codes by similarity, we use neighborhood signatures to generate feature vectors corresponding to Android applications. The feature vector is expressed as a vector of $2^{15} = 32{,}768$ dimensions. Each dimension value represents the number of nodes corresponding to the neighborhood signature of the function call graph. Detailed description is illustrated in Fig. 7. The node corresponding to a function call graph can represent the distribution of the number of substructures containing different function calls graph.

Figure 7(1) shows the number of different neighbor signature corresponding to an APK, and Fig. 7(2) shows its corresponding feature vector. The value of the ith dimension of an APK eigenvector represents the times of occurrence of a neighbor's signature corresponding to the binary value of i. For example in Fig. 7(2), the value of the third dimension is 6, and the binary value of the neighbor signature (000000000000011) is converted to decimal 3, which corresponds to the third dimension. The value 6 indicates that the neighbor signature (000000000000011) appears 6 times in the APK.

Support vector machine (SVM) judges the similarity of two malicious codes according to the similarity of this distribution. We input 32,768 dimensions vectors into the SVM. The SVM is trained by using the existing family datasets to obtain an effective classifier. We use Weka, a tool that implements typical machine learning algorithms, to perform model training and malware classification.

3 Evaluation

We evaluate the performance of NSDroid on five datasets as listed in Table 5. To verify the effectiveness and efficiency of NSDroid, we conduct the following four experiments:

(1) The efficiency of NSDroid. First, FalDroid uses frequent subgraph similarity to classify Android families. Therefore, we use the same dataset (DataSet1) to verify the efficiency of NSDroid by comparing its execution time with FalDroid (Sect. 3.2).
(2) Effectiveness of NSDroid. In the second experiment, we verified the effectiveness of NSDroid on the same dataset (DataSet1) by comparing with the classification results of FalDroid.

Table 5 Datasets used in our experiments

ID	Dataset source	Number of APK	Number of family
DataSet1	Paper [8]	6547	30
DataSet2	Paper [14]	1169	19
DataSet3	Paper [21]	24,533	71
DataSet4	Paper [18]	4941	33
DataSet5	Merge datasets	32,190	121

(3) Generalization of NSDroid. In the third experiment, we use five datasets to test the classification effect of NSDroid and verify the generalization of NSDroid (Sect. 3.4).

(4) Verification of classification algorithm. Finally, we use DataSet2 to verify the classification effect of NSDroid in different classification algorithms and verify that SVM classification algorithm is best (Sect. 3.5).

What we need to explain is that we chose FalDroid for comparison because it has a good classification effect. We also provided datasets and execution code.

3.1 Experimental dataset and environment

As shown in Table 5, we also obtained four multi-family datasets provided by other literature. The detail of these datasets is shown in Table 5. The DataSet5 dataset is a merged dataset after merging the previous 4 datasets (DataSet 1, DataSet 2, DataSet 3, DataSet 4). Repeating parts have been removed. The dataset contains 32,190 malicious samples that are divided into 121 families. Our experimental machine features a dual-core CPU with frequency 3.2 GHz, 16GB memory, and installs Ubuntu 14.04 (64 bit).

3.2 The efficiency of NSDroid

We perform the comparison test to compare the time efficiency between NSDroid and FalDroid. The procedure is designed as following three steps: (1) FCG generation; (2) the feature vector generation from FCG; and (3) the feature vector importation for classification. Since the first step is the same for both classification methods, we only compare the time consumption for step 2 and step 3.

(1) Generating feature vectors by FCG

We compare the time consumption of two methods to generate feature vectors from FCG. The same dataset (DataSet2) and the same classifier (SVM) are used for both methods.

As shown in Fig. 8, the time consumption of FalDroid is distributed between 0 and 20 s (diamond symbol). The runtime overhead of NSDroid is concentrated between 0 and 1 s (asterisk symbol), and 99% of runtime overhead is concentrated between 0 and 0.5 s. By comparing these two curves,

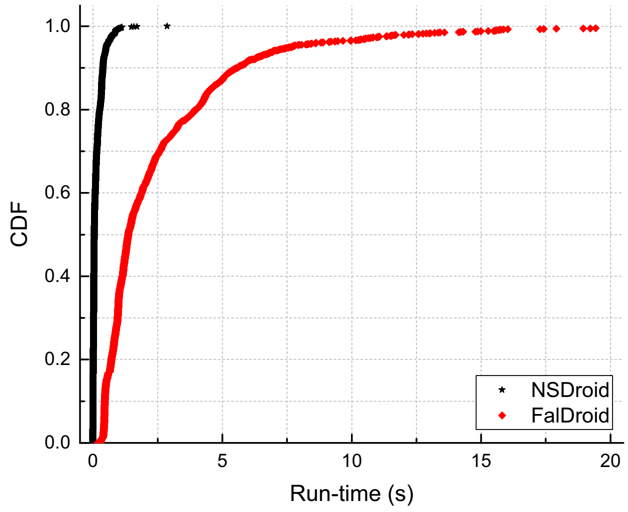


Fig. 8 Runtime contrast between NSDroid and FalDroid

Table 6 Time comparison of different dimensions

Method	Classifier	Dataset	Dimension	Time (s)
NSDroid	SVM	DataSet1	32,768	11.71
FalDroid	SVM	DataSet1	1000	1.87

NSDroid has better efficiency of runtime overhead, speeding up the classification process by nearly 20×.

(2) Importing feature vectors into the SVM model for classification

Since the dimensions of the feature vectors produced by the two methods we compare are different, we verify the effect of different dimensions on the time consumption of classification. NSDroid generates a feature vector of 32,768 dimensions, while FalDroid generates a 1000-dimensional feature vector. We put the feature vectors of two different dimensions into the same SVM model.

As shown in Table 6, the number of dimensions by NSDroid is 30 times more than that of FalDroid, while the time consumption is only six times more in step 2. It can be seen from the above analysis that the dimension has little effect on time consumption for SVM. Table 7 shows the test runtime of the two methods on DataSet1. As you can see from the total time, although the NSDroid takes longer in step 2, the total time is still much less than the FalDroid.

Table 7 Total time of two method

Method	Step 1: total time (FCG → vector) (s)	Step 2: total time (vector → SVM) (s)	Total time
FalDroid	2448	1.87	2449.87
NSDroid	83.2	11.71	94.91

Next we explain why NSDroid is more efficient than FalDroid. In NSDroid, the features of malware is captured by the neighborhood signature, which is generated by only using XOR operations. FalDroid, however, constructs frequent subgraphs to represent the malware features. The construction of frequent subgraphs and the relevant graph matching are computing-intensive and therefore consume more running time.

3.3 The effectiveness of NSDroid

In order to verify the effectiveness of NSDroid, we compare the classification performance between NSDroid and FalDroid on DataSet1. Because the metrics of FalDroid is TPR (true positive rate), FPR (false positive rate), *P* (precision), *R* (recall rate), *F* (*F* value), AUC (area under the curve), the metric of our classification is the same as above.

We use tenfold cross-validation to train and test DataSet1 with NSDroid. The experimental results are shown in Table 8.

Table 8 Performance comparison of two methods

MalFamily	NSDroid method						FalDroid method					
	TPR	FPR	P	R	F	AUC	TPR	FPR	P	R	F	AUC
adwo	0.959	0.003	0.943	0.959	0.951	0.995	0.879	0.002	0.946	0.879	0.911	0.938
airpush	0.813	0.001	0.924	0.813	0.865	0.973	0.600	0.001	0.833	0.600	0.698	0.799
basebridge	0.931	0.002	0.956	0.931	0.943	0.983	0.950	0.002	0.960	0.950	0.955	0.974
boqx	0.475	0.002	0.607	0.475	0.442	0.892	0.375	0.001	0.667	0.375	0.480	0.687
boxer	1	0	1	1	1	1	1	0	1	1	1	1
clicker	1	0	1	1	1	1	1	0	1	1	1	1
dowgin	0.960	0.002	0.974	0.960	0.967	0.991	0.908	0.006	0.933	0.908	0.921	0.951
droiddream	0.941	0.001	0.960	0.941	0.950	0.988	0.941	0.002	0.865	0.941	0.901	0.969
droidkungf	0.970	0.003	0.974	0.970	0.972	0.992	0.975	0.011	0.918	0.975	0.988	0.988
fakedoc	1	0	1	1	1	1	1	0	1	1	1	1
fakeinst	0.993	0	1	0.993	0.996	0.999	0.996	0.002	0.987	0.996	0.991	0.997
fakeplay	1	0	1	1	1	1	1	0.001	0.875	1	0.933	1
geinimi	0.990	0	1	0.990	0.995	0.994	1	0	1	1	1	1
gingermast	0.934	0.003	0.950	0.934	0.942	0.992	0.893	0.010	0.838	0.893	0.865	0.942
golddream	0.966	0.001	0.934	0.966	0.899	0.980	0.963	0	1	0.963	0.981	0.981
hongtoutou	1	0	1	1	1	1	1	0	1	1	1	1
iconosvs	1	0	1	1	1	1	0.980	0	1	0.980	0.990	0.990
imlog	1	0	1	1	1	1	1	0	1	1	1	1
kmin	0.996	0	0.996	0.996	0.996	1	0.976	0	1	0.976	0.988	0.988
kuguo	0.964	0.003	0.948	0.964	0.956	0.997	0.924	0.003	0.940	0.924	0.932	0.960
mobiletx	1	0	1	1	1	1	1	0	1	1	1	1
pjapps	0.971	0	0.987	0.951	0.971	0.996	0.963	0	1	0.963	0.981	0.981
plankton	0.989	0.001	0.995	0.989	0.992	0.999	0.981	0.002	0.986	0.981	0.983	0.990
smskev	0.991	0	0.991	0.991	0.991	0.995	0.972	0.001	0.946	0.972	0.959	0.986
smsreg	0.964	0	0.969	0.964	0.955	0.958	0.825	0.006	0.733	0.825	0.776	0.910
utchi	1	0	1	1	1	1	1	0	1	1	1	1
waps	0.973	0.003	0.968	0.973	0.970	0.996	0.959	0.005	0.950	0.959	0.955	0.977
youmi	0.858	0.003	0.843	0.858	0.851	0.983	0.711	0.003	0.794	0.711	0.750	0.854
yzhc	1	0	1	1	1	1	1	0	1	1	1	1
zitmo	1	0	1	1	1	1	1	0	0.909	1	0.952	1
Avg	0.959	0.002	0.966	0.959	0.961	0.993	0.945	0.004	0.944	0.945	0.944	0.971

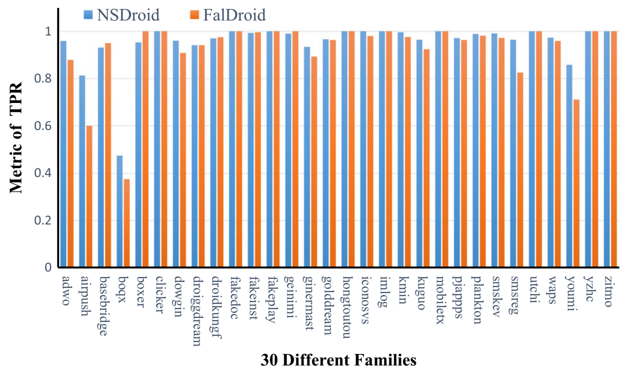


Fig. 9 Detection of 30 families with metric of TPR

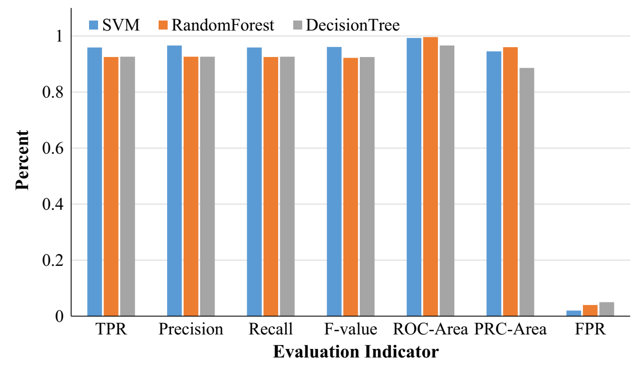


Fig. 11 Classification performance contrast of three different classifiers

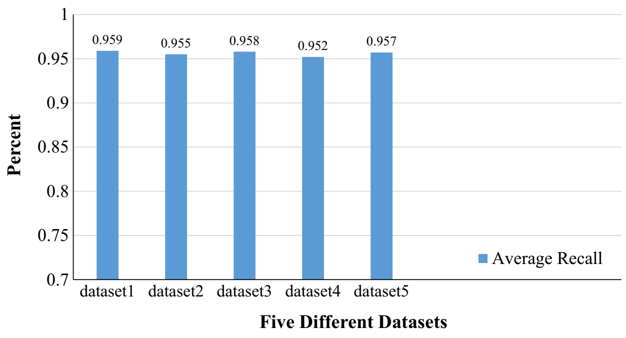


Fig. 10 The average recall in five different datasets

Table 8 shows the performance values of various classification metrics of NSDroid and FalDroid on the same dataset. Through the comparison of the two methods in various metrics, it is found that the TPR value of NSDroid is higher than 0.9 in most families. In addition, the TPR value of 11 families even reaches 1, and the FPR value of FalDroid is as low as 0 in 17 families.

Figure 9 shows the comparison of TPR indexes between the two methods in 30 malicious families. As can be seen from the figure, 27 of the 30 families are better than the analysis results of FalDroid. In conclusion, these show that the NSDroid method has a good performance in classification.

3.4 The generalization of NSDroid

In order to verify the generalization of the NSDroid method, we use five datasets (as shown in Table 5) to verify the classification effect of NSDroid. We use a tenfold cross-validation method to train and test these five different datasets, and the experimental results are shown in Fig. 10.

In Fig. 10, the NSDroid has achieved very good classification results in Dateset1, Dateset2, Dateset3, Dateset4, Dateset5, and the classification metric is stable (concentrated at about 95.5%). To sum up, the classification effect of NSDroid method on these five datasets shows that our method is very generalized.

3.5 Parameter selection of NSDroid

Three classifiers are considered in our testing progress: Random Forest, Decision Tree, and SVM. We use the neighborhood signature to classify the Dateset2. The results are shown in Fig. 11. SVM has an excellent classification performance in TPR, precision, recall, and F-values. Although the performance of RandomForest on RocArea is the highest, SVM achieves the highest on PRCArea. It is noted that PRCArea can better measure the classification capability when the proportion of the positive samples to the negatives is not balanced. In addition, Fig. 11 illustrates the average FPR of the three classifiers. SVM clearly achieves the smallest value, which suggests the most effective in classification. Based on the above evidences, we select SVM as the classifier of NSDriod.

4 Limitations

The experimental comparison shows the effectiveness of NSDroid in multi-family classification of Android malicious applications. Because NSDroid is implemented on the basis of function call graph, our method has a strong dependence on the function call graph. The function call graph generated in this paper is implemented by using the currently common function call graph generation tool (Androguard). Because the tool is a static method to generate a function call graph, NSDroid is also limited to the static function call graph. The dynamic distribution mechanism is determined by different parameters in the dynamic execution process. The calling relation of this part cannot be obtained directly from static code analysis, which is also the common limitation of static analysis methods. This paper is based on static analysis, so these limitations also exist.

5 Related work

At present, there are mainly two kinds of detection technologies for the research of Android malware detection: the dynamic detection and the static detection.

The dynamic detection requires the actual running of the application (or in the sandbox [22] to simulate the execution of malicious applications). Wang et al. [23] proposed a dynamic detection system for malicious applications. By monitoring the running process of the Android system, they got the frequency of the application calls and the sequence of the application calls. Finally, the system extracted 65 system calls to generate a co-occurrence matrix and adopts a machine learning method to determine malicious classification. Bläsing et al. [24] constructed a sandbox system simulator in Android environment. The sandbox intercepted and recorded all system call logs from the bottom of the kernel. Then the system analyzed the malicious nature of the software by using the log. Ruiz-Heras et al. [25] used dynamic methods to obtain three characteristics of Android malicious applications: interfaces usage, application-related and communication-related features. This method used these three types of features for malicious application detection.

Recent works [26–28] proposed a dynamic monitoring system that automatically monitored the transmission and storage of the data. When the data were considered to be used in a dangerous way, the system automatically recorded the content of the data and the program (using the data). Finally, they used these contents to determine the malicious nature of the application. Isohara et al. [29] proposed a detection system with a logger and an analyzer. In this system, the logger recorded the system calls used by the malware application. The analyzer filtered the specific event behavior. The system classified malicious applications by obtaining system calls and event behaviors.

In summary, the using of dynamic technology to detect Android malicious applications have the following shortcomings: (1) It consumes too many resources (Because you need to install software to detect). (2) The execution of an application often takes much time. (3) The dynamic detection method [30,31] does not cover all the behaviors and activities of the application.

Static detection is to understand the behavior of a program by using the application code without executing the program. For example, Kirubavathi et al. [19] proposed a method to detect Android malicious applications that had botnet types. This method used the botnet characteristics-related unique patterns of requested permissions and used features. Garg et al. [32] proposed a network-based detection model of Android malicious application. The method detected Android malicious applications by analyzing different network parameters of the Android application. Miao [33] proposed a way to abstract stable behavioral features from the original API. And used the SVM classifier to classify malicious applications.

Zhang et al. [5] proposed a malicious application classification method -DroidSIFT. The method was based on the API dependency graph. It classified malicious applications by calculating the similarity among API dependency graphs. There were many classification methods extracted the data flow [34–36] from applications. For example, Suarez et al. [6] proposed a malicious application classification method based on the control flow graph (the method named Dendroid). And then it selected eight basic structures from the control flow graph. The method determined the maliciousness of the application by comparing the similarity of the eight structures. Zhou et al. [7] proposed an analytical approach based on the function call graph. This method generated a function call graph by sensitive permissions and API. By calculating the similarity of the function call graph, they determined the malicious nature of the application. Fan et al. [8] proposed a method,FalDroid, based on the function call graph. The method divided the function call graphs into frequent subgraphs. And then it marked the frequent subgraphs that appeared in the various malicious families as features. Finally, it used these features to classify malicious applications.

6 Conclusion

In this paper, we propose a classification method of malicious Android application, NSDroid, based on the function call graph. This method uses the function call graph of the application as the analysis object. It abstracts the subgraph structure of Android application as the neighborhood signature of many nodes. By this method, the issue of similarity between Android applications is transformed into a histogram problem. In the experiments, NSDroid has achieved a good classification effect for most malicious families. Compared with the FalDroid [8] method, NSDroid not only enhances the classification performance, but also greatly reduces the runtime.

Acknowledgements We would like to thank anonymous reviewers for their comments. This work is partially supported by the National Natural Science Foundation of China under Grant No. 61672543, the Open Research Fund of Hunan Provincial Key Laboratory of Network Investigational Technology, Grant No. 2017WLZC002, the Fundamental Research Funds for the Central Universities of Central South University, Grant No. 2018zzts175.

Compliance with ethical standards

Ethical approval Our article does not contain any studies with human participants or animals performed by any of the authors. Every participant in our paper received informed consent.

References

1. Detection report: Tencent anti virus laboratory 2017 q3 security report. https://slab.qq.com/news/authority/1744.html. Accessed 2 Nov 2018
2. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: Droidminer: automated mining and characterization of fine-grained malicious behaviors in android applications. In: Proceedings of 2014 European Symposium on Research in Computer Security (ESRCS), pp. 163–182 (2014)
3. Hou, S., Ye, Y., Song, Y., Abdulhayoglu, M.: Hindroid: an intelligent android malware detection system based on structured heterogeneous information network. In: Proceedings of 2017 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 1507–1515. ACM (2017)
4. Onwuzurike, L., Mariconti, E., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: Mamadroid: detecting android malware by building Markov chains of behavioral models (extended version). arXiv preprint arXiv:1711.07477 (2016)
5. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security(CCS), pp. 1105–1116. ACM (2014)
6. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.: Dendroid: a text mining approach to analyzing and classifying code structures in android malware families. Expert Syst. Appl. (ESA) **41**(4), 1104–1117 (2014)
7. Jiang, X., Zhou, Y.: Dissecting android malware: characterization and evolution. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE (2012)
8. Fan, M., Liu, J., Luo, X., Chen, K., Chen, T., Tian, Z., Zhang, X., Zheng, Q., Liu, T.: Frequent subgraph based familial classification of android malware. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering(ISSRE), pp. 24–35. IEEE (2016)
9. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI) **26**(10), 1367–1372 (2004)
10. Sen, A.K., Bagchi, A., Zhang, W.: Average-case analysis of best-first search in two representative directed acyclic graphs. Artif. Intell. (AI) **155**(1–2), 183–206 (2004)
11. Levin, L.A., Venkatesan, R.: An average case NP-complete graph colouring problem. Comput. Sci. **27**(5), 808–828 (2002)
12. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (DASP), pp. 317–326. ACM (2012)
13. Deshotels, L., Notani, V., Lakhotia, A.: Droidlegacy: automated familial classification of android malware. In: Proceedings of 2014 ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW), p. 3. ACM (2014)
14. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: Madam: effective and efficient behavior-based android malware detection and prevention. IEEE Trans. Dependable Secure Comput. (TDSC) **15**(1), 83–97 (2016)
15. Jang, Y., Lee, N., Kim, H., Park, S.: Design and implementation of a bloom filter-based data deduplication algorithm for efficient data management. J. Ambient Intell. Hum. Comput. (2018). https://doi.org/10.1007/s12652-018-0893-1
16. Hido, S., Kashima, H.: A linear-time graph kernel. In: Proceedings of the 9th IEEE International Conference on Data Mining (ICDM), pp. 179–188. IEEE (2009)
17. Wang, W., Gao, Z., Zhao, M., Li, Y., Liu, J., Zhang, X.: Droidensemble: detecting android malicious applications with ensemble of string and structural static features. IEEE Access **6**, 31798–31807 (2018)
18. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: CERT Siemens. Drebin: effective and explainable detection of android malware in your pocket. In: The Network and Distributed System Security Symposium (NDSS), Vol. 14, pp. 23–26. ISOC (2014)
19. Kirubavathi, G., Anitha, R.: Structural analysis and detection of android botnets using machine learning techniques. Int. J. Inf. Secur. (IJIS) **17**(2), 153–167 (2018)
20. Jang, J., Kang, H., Woo, J., Mohaisen, A., Kim, H.K.: Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. Comput. Secur. **58**, 125–138 (2016)
21. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 252–276. Springer, Berlin (2017)
22. Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software(MUS), pp. 55–62. IEEE (2010)
23. Wang, C., Li, Z., Mo, X., Yang, H., Zhao, Y.: An android malware dynamic detection method based on service call co-occurrence matrices. Ann. Telecommun. (AT) **72**(9–10), 607–615 (2017)
24. Wong, M.Y., Lie, D.: Intellidroid: a targeted input generator for the dynamic analysis of android malware. In: Proceedings of the 2016 ISOC Network and Distributed System Security Symposium (NDSS), vol. 16, pp. 21–24. ISOC (2016)
25. Ruiz-Heras, A., García-Teodoro, P., Sánchez-Casado, L.: Adroid: anomaly-based detection of malicious events in android platforms. Int. J. Inf. Secur. (IJIS) **16**(4), 371–384 (2017)
26. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
27. Bai, J., Wang, W., Qin, Y., Zhang, S., Wang, J., Pan, Y.: Bridgetaint: a bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications. IEEE Trans. Inf. Forensics Secur. (TIFS) **14**(3), 677–692 (2019)
28. Dai, S., Liu, Y., Wang, T., Wei, T., Zou, W.: Behavior-based malware detection on mobile phone. In: Proceedings of the 6th International Conference on Wireless Communications Networking and Mobile Computing (WCNMC), pp. 1–4. IEEE (2010)
29. Isohara, T., Takemori, K., Kubota, A.: Kernel-based behavior analysis for android malware detection. In: Proceedings of the 7th International Conference on Computational Intelligence and Security (CIS), pp. 1011–1015. IEEE (2011)
30. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPISM), pp. 15–26. ACM (2011)
31. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the 7th European Workshop on System Security (EWSS), pp. 1–6. ACM (2014)
32. Garg, S., Peddoju, S.K., Sarje, A.K.: Network-based detection of android malicious apps. Int. J. Inf. Secur. (IJIS) **16**(4), 385–400 (2017)
33. Miao, Q., Liu, J., Cao, Y., Song, J.: Malware detection using bilayer behavior abstraction and improved one-class support vector machines. Int. J. Inf. Secur. (IJIS) **15**(4), 361–379 (2016)
34. Wei, F., Roy, S., Ou, X., et al.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. (TOPS) **21**(3), 14 (2018)

35. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE), pp. 576–587. ACM (2014)
36. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM Sigplan Notices (SN) **49**(6), 259–269 (2014)