AFCGDroid: Deep Learning Based Android Malware Detection Using Attributed Function Call Graphs

Tong Lu, Xiaoyuan Liu, Jingwei Chen, Naitian Hu, Bo Liu
College of Computer, National University of Defense Technology, Changsha, 410003, China
lt2020@nudt.edu.cn

Abstract. With the rapid development of the mobile network, large numbers of Android malware emerge and pose a serious security threat to Android users. The function call graphs (FCG) extracted from Android application involves permissions, API calls, and structure semantics. Leveraging FCGs has great potential for Android application detection. In this paper, we propose AFCGDroid, an approach based on attributed function call graph (AFCG), to detect Android malware. The nodes of FCGs are divided into internal nodes and external nodes in our work. AFCGDroid extracts FCGs through static analysis and then generates AFCGs by labeling its internal nodes with the external nodes. To handle the graph data, graph embedding based on deep learning is proposed to embed the AFCGs into low dimension vectors. We evaluate AFCGDroid in malware detection achieving a 99.6% accuracy in a publicly available dataset, Drebin.

1. Introduction
With the widespread of the mobile network, smartphones are gradually changing people's lives. Daily needs, such as instant messages, online shopping, and bank transfer, can be easily realized through some Android applications. Despite the success of the Android system, security issues are increasing due to a large number of malicious apps flood the Android app markets, mixed in good ones. The malware generally has some malicious behavior, such as accessing personal data, remote control, forced installation, and browser hijacking. According to the mobile malware report [1], there are more than 4.18 million malicious apps detected in 2019. Therefore, constructing automated classifiers to assist Android app detection can greatly reduce time and cost, which is of critical importance to improve mobile security.

Nowadays, call graph, as a new kind of fine-grained features, is adopted for Android malware detection due to more powerful representation of the features including API calls, execution flow, code structure. However, the usage of FCG in Android malware detection remains challenging. The FCGs of Android applications usually have hundreds of thousands of nodes, which makes it difficult to assessing the similarity of FCG. Graph embedding based on random walks is widely used to solve graph representation problems, yet much structural information is lost. How to make full use of structural information to generate more representative feature vectors in a reasonable time is a problem that needs to be addressed in our work.

In this paper, we present a novel method, AFCGDroid, which conduct the Android malware detection through FCG. Compared to other work using FCGs, we generate AFCGs to reduce the scale of graph data while maintaining the original structure of the call graph. Moreover, our graph embedding method based on deep learning takes full advantage of the structured semantics of graphs...
as well as features of Android applications. Through adequate experiments, the influence of hyperparameters and the time consumption were researched. AFCGDroid has achieved a top accuracy in both Android malware detection and malicious family classification. Our contributions can be summarized as follows:

1. We propose an approach to reorganizing the FCG. By analyzing the composition of the node, we transform part of the external functions into attributes of the internal functions, thus preserving the complete structure of the FCG, reducing the size of the graph, and labeling the nodes.

2. We propose a graph embedding method based on deep learning to handle FCGs, which takes advantage of graph structure information and Android application characteristics to present a graph as a vector.

3. We implement AFCGDroid, an Android malware detection method based on AFCGs, which achieves 99.4% accuracy on Drebin dataset [2] and 96.2% accuracy on a dataset collected from VirusShare [3].

2. Related Work

Graph data contains sufficient information, and some studies employ graph structure data as a feature for Android malware detection [4-7]. Gascon et al. [6] exploit the FCG by labeling the nodes according to the instructions contained in the corresponding functions and a feature map inspired by graph kernels is applied to embed FCGs in a vector space capturing structural relationships. Hou et al. [4] construct a Linux kernel system call graph by dynamically analyzing the Android applications. The weights of all nodes and edges of the graph are used to build feature vectors containing structural semantics. Ge et al. [5] decompose each FCG into rooted subgraphs which are regarded as word sequences. A skip-gram training process of doc2vec is followed to encode each FCG into a vector. Pektacs et al. [7] construct an API call graph for all execution paths captured through pseudo-dynamic analysis. The graph embedding method based on the random walk is applied to generate low-dimensional feature vectors.

Compared to the above work, our method labels the nodes as well as reduces the size of the graph data. At the same time, the structure of the FCG is well maintained. Besides, our graph embedding approach is based on the neural network that will be trained together with the classifier.

3. The Proposed Method

In this section, we introduce an overview of AFCGDroid. As shown in Fig. 1, our proposed method includes three stages. Firstly, we decompile an APK and extract functions with dependency to generate an FCG through static analysis. Secondly, the nodes representing the functions in the application are analyzed and filtered, and some of them are represented as attributes of the remaining

![Fig. 1 Overview of AFCGDroid.](image-url)
nodes to generate AFCGs. Third, we implement a deep neural network based on the graph embedding method to detect Android malware.

3.1. FCG Extraction
This section aims to decompile the APK file to generate FCGs with node attributes as the representative feature. The APK file is the installation file of an Android application. It packages all the information on the Android application, including the compiled code, request permissions, resource files, certificates, and other information. With the help of the androguard project [8], we parsed the APK file and decompiled the DEX file inside to achieve enough information about the functions. All the functions are searched for specific instructions which cause the invocations. We construct the FCG by taking all the functions as nodes and the calling relationship between them as edges.

3.2. AFCG Generation
FCGs are extracted to represent the Android application, which turns the task of malware detection into graph classification. We analyze the functions represented by the graph nodes and classify them into two categories: external functions and internal functions. An internal function whose implementation is available inside the application and can be decompiled to access instructions. The names of internal functions can be modified and are generally not repeated in different applications. An external function is a method that has a declaration but no specific code in an Android application. The names of external functions generally do not change and are used repeatedly in different applications.

We set up a small dataset containing 1,000 malicious samples and 1,000 benign samples for analysis. We come to the following conclusion:

(1) The number of FCG nodes can reach hundreds of thousands, with an average of tens of thousands.

(2) The number of nodes in a benign sample is generally more than the number of a malicious sample.

(3) The number of external functions is about 21% of the number of all functions.

| TABLE 1. The external function distribution of the small dataset. |
|---------------------------------|-------------------|----------------|----------------|----------------|
| Android API                     | Java API          | Others         |
| Ben.                            | Mal.              | Ben.           | Mal.           | Ben.           | Mal.           |
| total number                    | 51,406            | 29,558         | 9,681          | 8,935          | 552,527        | 472,688        |
| $\leq 1\%$                      | 34,166            | 17,964         | 4,852          | 4,846          | 508,701        | 461,173        |
| 1%$<p\leq10\%$                  | 9,043             | 6,168          | 2,059          | 2,184          | 40,971         | 11,231         |
| 10%$<p\leq20\%$                 | 1,604             | 2,342          | 509            | 690            | 2,208          | 152            |
| 2%$<p\leq50\%$                  | 2,446             | 2,807          | 867            | 918            | 499            | 108            |
| 50%$<p\leq100\%$                | 4,147             | 277            | 1,394          | 297            | 148            | 24             |

Since the external functions have no specific implementation in the application, we have no way of knowing which functions are invoked by the external functions, which results in the nodes representing them being located at the edge of the FCG. This means the ignoring of external functions has little effect on the main structure of the graph. Moreover, we come up with an idea to label internal functions with external functions, which preserves both the characteristics of the external functions and the structural information of the internal functions. As a result, we have reduced the number of graph nodes by approximately 21%.

Subsequently, we focus on the composition and characteristics of external functions and select appropriate external functions as attributes of nodes. Based on the name of the external functions, we classify them into three categories. The first category includes the Android APIs. The second is Java support libraries, and the third contains the third-party support libraries and other external functions.
that we can't recognize. We calculated the frequency of occurrence of all external functions from the small dataset, and the results are shown in the Table 1. The \( p \) in the table indicates the frequency with which a function appears in different samples. If a function has a \( p \) of 100\%, it means that the function has appeared in all samples. If a function has a \( p \) of 1\%, it means that the function has appeared 10 times in all 1,000 samples. The numbers in the second through fifth rows indicate the total number of functions with frequencies \( p \) in a certain range.

External functions with low frequencies contribute little to the classification of Android applications, and we choose to simply ignore them. We will set up \( K \) to represent the total number of features for selection and then choose among external functions from three categories in a ratio of 7:2:1 based roughly on the distribution of the three categories at high frequencies. Inspired by [12], we score each external function with Mutual Information (MI). MI is a measure of the interdependence between two variables with probability. A higher MI score means a stronger association between the two variables. We used the following formula to calculate the interdependence between external function frequency and sample labels,

\[
MI(C, X) = \sum_{c \in C} \sum_{x \in X} p(c, x) \log_2 \left( \frac{P(c, x)}{P(c)P(x)} \right).
\]

After calculating MI scores for all external functions, we divide external functions into two categories based on whether their frequency of appearance is higher in benign or malicious samples and select the same number from each category. We divided the Android API into two categories based on the number of times it appeared in benign and malicious samples, and selected 35\% of the number \( K \) in each category, for a total of 70\%. We label the internal nodes in the FCGs with filtered external functions and convert the node feature into a \( K \)-dimension vector, where \( K \) is the number of external functions chosen as features. Each value in the feature vector can be 1 or 0, indicating whether the node calls a particular external function or not.

3.3. Machine Learning Based On Graph Embedding

The reconstructed FCG is denoted as \( g = \langle V, E, X \rangle \). \( V \) and \( E \) respectively represent the set of nodes and edges. \( X \) is the set of node features. In other words, for every node \( v_i \in V \), there is a \( K \)-dimensional vector \( x_i \in X \), where \( K \) is the number of external functions filtered to represent node features. We build a graph embedding network in which we input graph data \( g \) and output a vector \( h \). Then a Multilayer Perceptron (MLP) [13] with a Softmax layer is followed to classify the representative vector \( h \) and detect malware.

![Fig. 2. Graph Embedding Network.](image-url)
Inspired by structure2vec [14], we propose a graph embedding method based on a neural network and aggregate the features of the nodes and their surroundings to generate a more representative node feature vector. After that, we aggregate the features of all nodes to represent the entire graph. The process of graph embedding is shown in Fig. 2.

Firstly, we multiply the features of each node by \( W_1 \) to initialize \( u_i \) and compress the feature from \( K \) dimension \( x_i \) to \( m \) dimension. That is \( u_i^0 = W_1 x_i \), where \( W_1 \) is a \( m \times k \) matrix.

Subsequently, we incorporate the structural semantics of the graph into the node feature representation. According to our second empirical assumption, a combination of functions indicates a more specific intention than a single one and represents a clearer behavioral characteristic. For AFCGs, the features of a node are heavily correlated with the features of itself as well as its surroundings. Since the caller and the callee of a node are equally essential to the generation of the node feature, we treat AFCG as an undirected graph. Besides, we denote \( N(v) \) as the set of neighbors of the node \( v \) in graph \( g \). And then we update the node vector representation \( u_i \) as

\[
u_i^{t+1} = F(u_i^t, \sum_{v_j \in N(v_i)} u_j^t), \forall v_i \in V.
\]

\( F \) is a nonlinear function in which the parameters in it do not change during different iterations. In one iteration, the node feature aggregates the features of the node itself and its neighbors. After the \( T \) iteration, the result combines the features of itself and those within \( T \) distance, and the nodes at a different distance have different influence on the final result. In particular, we design \( F \) to have the following form

\[
F(u_i^t, \sum_{v_j \in N(v_i)} u_j^t) = \tanh(u_i^t + \sum_{v_j \in N(v_i)} W_2 u_j^t),
\]

where \( W_2 \) is a \( m \times m \) matrix. The dimension of the feature vector remains unchanged during iteration.

\textbf{Algorithm 1} The process of graph embedding

| Input: FCG \( g = (V, E, X) \) |
| Output: Vector \( h \) |

1. for each \( v_i \in V \) do
   2. \( u_i^0 = W_1 x_i \)
   3. end for
4. for \( t = 1 \) to \( T \) do
   5. for each \( v_i \in V \) do
     6. \( u_i^t = \tanh(u_i^{t-1} + \sum_{v_j \in N(v_i)} W_2 u_j^t) \)
   7. end for
8. end for
9. Return \( h = \text{Relu}(\sum_{v_i \in V} u_i^T) \)

Finally, we generate a graph representation vector with node features. According to our first empirical assumption, the characteristics of the graph are determined by all the node features. Thus, we aggregate the features of all the nodes to get the representative vector \( h \) of the graph as \( h = A \left( \sum_{v_i \in V} u_i^T \right) \), where \( A \) is the aggregation function and we take \( A \) as a summation followed by a rectified linear unit \( \text{Relu} \) in our work. The overall algorithm of graph embedding for each FCG is summarized in Algorithm 1.

After generating the graph representation vectors, we classify them with an MLP network. The followed MLP network contains a hidden layer and an output layer. The fully-connected structure of MLP is suitable for vector with no sequential information. A Softmax layer is followed to normalize the two-dimension output into predictive probabilities of malicious and benign.

4. Experiment

In this section, the experimental results of AFCGDroid are described in detail. To verify the effectiveness of the proposed method, we collect Android benign and malicious samples to construct
datasets and classify the datasets with AFCGDroid. The classification metrics are defined to evaluate the performance. Finally, we evaluate the time consumption of AFCGDroid and validate its performance on malware detection and malicious family classification.

4.1. Dataset
The malicious samples in the experimental dataset come from Drebin [2] and VirusShare. The Drebin dataset includes 5,560 malicious samples collected from 2010 to 2012 that were labeled as one of 179 malicious families. The 5,785 malicious samples we took from VirusShare were collected from 2018. The benign samples were collected from Apkpure [15] in 2019. We removed some of the samples that were broken or that we couldn't handle and finally we construct two datasets, one is based on the Drebin dataset, and another is based on the VirusShare data. We include enough benign samples in both datasets to ensure that there is the same number of benign and malicious samples in the datasets.

4.2. Performance metrics
As the single accuracy metric is not an excellent way to assess the proposed method, we choose four different metrics to evaluate our model's performance. They are accuracy, precision, recall, F-measure, which are based on the values of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN).

4.3. Evaluation results

**Evaluation of time consumption.** The time spent for malware detection with AFCGDroid consists of three parts, extraction of the FCGs, the AFCG generation, and the operation of the neural network. We
conducted time evaluation experiments with CPU only on a test machine, which is equipped with
Intel(R) Xeon(R) E5-2630 v4 CPU. We apply AFCGDroid using the hyper-parameters discussed
previously on 2000 samples. Experimentally, we found that time consumption is more related to
the node number of FCG than the sample size, and the performance is shown in Fig. 3. The time of the
FCG generation is about 95% of the total consumption, and the time to regenerate the AFCG is less
than one-tenth of that. Our neural network takes very little time and it can be much faster with GPUs,
which is significant for large-scale Android application detection.

**Evaluation on the Android malware family classification.** The Drebin dataset sample consists of
179 malware families. We selected the top 20 malware family samples as the dataset for the
experiment. We change the dimensions of the output layer in the model to the number of families for
multiclassification. We select 20% of the data as the validation set and train the remaining data using
10-fold cross validation. As shown in the confusion matrix in Fig. 4 we achieved great results in
family classification, achieving an overall 95.1% accuracy across 20 families.

**Comparison with related works.** Our proposed method exhibits an accuracy of 99.6% in
experiments based on the Drebin dataset, and we focus our attention on experiments with the
VirusShare dataset. To further demonstrate the effectiveness of our proposed method, we apply several
related methods on the same dataset. We compare our approach to some advanced work, including
Arp et al. [2], McLaughlin et al. [16], Nix et al. [17]. As is shown in Table 2, our approach achieves
the highest scores in almost four metrics. Overall, our approach shows sufficient effectiveness in
Android malware detection.

| Methods        | Accuracy | Precision | Recall | F-measure |
|----------------|----------|-----------|--------|-----------|
| Arp [2]        | 94.3     | 95.1      | 93.7   | 94.4      |
| McLaughlin [16]| 92.2     | 93.1      | 91.6   | 92.3      |
| Nix [17]       | 94.8     | 95.6      | 94.3   | 95.0      |
| Proposed Methods | 96.3  | 96.4      | 95.9   | 96.2      |

5. Conclusion
In this paper, we propose AFCGDroid, a method for Android malware detection. The approach
extracts the FCGs from APKs and generates node attributes with filtered external functions. We build
a deep learning model with the graph embedding process and an MLP network, in which AFCGs are
input to learn a representative vector and trained with deep learning techniques to identify Android
malware.

We carry out experiments on the dataset collected from Drebin, VirusShare, and Apkpure to
evaluate AFCGDroid. In the experiments on malware detection, we achieved 99.6% accuracy on the
Drebin dataset and 96.2% accuracy on the VirusShare dataset. We achieved an overall accuracy of
95.1% in the multi-classification on Android malicious family applications. Our work demonstrates
the potential of graph learning in Android malware detection.

Acknowledgments
This paper was supported by the National Natural Science Foundation of China under grant
No.61572513.

References
[1] G data mobile malware report 2019. https://www.gdata.pt/news/g-data-mobile-malware-report-
2019-new-high-for-malicious-android-apps.
[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT
Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In
Ndss, volume 14, pages 23–26, 2014.
[3] VirusShare. https://virusshare.com/.
[4] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW), pages 104–111. IEEE, 2016.

[5] Xiuting Ge, Ya Pan, Yong Fan, and Chunrong Fang. AMDroid: Android malware detection using function call graphs. In 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pages 71–77. IEEE, 2019.

[6] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In Proceedings of the 2013 ACM workshop on Artificial intelligence and security, pages 45–54, 2013.

[7] Abdurrahman Pektaş, and Tankut Acraper. Deep learning for effective android malware detection using api call graph embeddings. Soft Computing, 24(2):1027–1043, 2020.

[8] Androguard. http://code.google.com/p/androguard/.

[9] Marko Dimjasević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakar. Evaluation of android malware detection based on system calls. In Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pages 1–8, 2016.

[10] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and KuoPing Wu. Droidmat: Android malware detection through manifest and api calls tracing. In 2012 Seventh Asia Joint Conference on Information Security, pages 62–69. IEEE, 2012.

[11] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In 2013 IEEE 25th international conference on tools with artificial intelligence, pages 300–305. IEEE, 2013.

[12] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. Analysis of bayesian classification-based approaches for android malware detection. IET Information Security, 8(1):25–36, 2014.

[13] Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. Neural networks, 2(5):359–366, 1989.

[14] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In International conference on machine learning, pages 2702–2711, 2016.

[15] Apkpure. https://apkpure.com/.

[16] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickle, Ziming Zhao, Adam Doupe, et al. Deep android malware detection. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pages 301–308, 2017.

[17] Robin Nix and Jian Zhang. Classification of android apps and malware using deep neural networks. In 2017 International joint conference on neural networks (IJCNN), pages 1871–1878. IEEE, 2017.