Detecting and Categorizing Android Malware with Graph Neural Networks

Peng Xu
Technical University of Munich

Claudia Eckert
Technical University of Munich

Apostolis Zarras
Delft University of Technology

ABSTRACT

Android is the most dominant operating system in the mobile ecosystem. As expected, this trend did not go unnoticed by miscreants, and quickly enough, it became their favorite platform for discovering new victims through malicious apps. These apps have become so sophisticated that they can bypass anti-malware measures implemented to protect the users. Therefore, it is safe to admit that traditional anti-malware techniques have become cumbersome, sparking the urge to come up with an efficient way to detect Android malware. In this paper, we present a novel Natural Language Processing (NLP) inspired Android malware detection and categorization technique based on Function Call Graph Embedding. We design a graph neural network (graph embedding) based approach to convert the whole graph structure of an Android app to a vector. We then utilize the graphs' vectors to detect and categorize the malware families. Our results reveal that graph embedding yields better results as we get 99.6% accuracy on average for the malware detection and 98.7% accuracy for the malware categorization.

ACM Reference Format:
Peng Xu, Claudia Eckert, and Apostolis Zarras. 2021. Detecting and Categorizing Android Malware with Graph Neural Networks. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3412841.3442080

1 INTRODUCTION

Android is the most popular mobile operating system in the world. Unfortunately, it has also become the leading target platform for attackers. Adversaries use Android to launch millions of malicious apps that dupe victims into revealing their private data or performing malicious operations, such as spying on users' actions, propagating spam, or launching unwanted advertisements. At the same time, Android malware investigation, which includes malware detection and categorization, has become a crucial task for security investigators. As a result, numerous research works have attempted to detect Android malware [5, 7, 10]. Recently, a significant portion of the proposed approaches leverages the contextual information of Android applications. For example, Li et al. [7] presented a classifier using the Factorization Machine architecture, where they extract various Android app features from manifest files and source code. Similarly, Chen et al. [4] proposed an approach that analyzes Android malware based on its static behavior that involves the use of permissions, components, and sensitive API calls.

Although the above methods add an an extra security layer to Android, they come with limitations. For instance, the contextual information struggles against malware obfuscation procedures. Such examples are (i) the *Identifier Renaming*, which replaces the packages, classes, or methods' original values with random or encrypted labels, (ii) the *Dead-Code Insertion*, which inserts ineffectual instructions to a program to alter its appearance while maintaining its behavior, and (iii) the *Instruction Substitution*, which replaces instructions with equivalent ones. These procedures cause changes in the compiled code to evade detection. In addition, the previously mentioned malware detection techniques have limitations in learning comprehensive program semantics to characterize malware of high diversity and complexity. In terms of learning approaches, they borrow techniques from the Natural Language Processing (NLP), such as document embedding, to process the source code or the dissembled binary files. However, in contrast with document embedding, source code and dissembled binaries are more structural and logical than natural languages. Therefore, considering the graph representation of source code and binary files, such as Abstract Syntax Tree (AST), Control Flow, and Data Flow Graph, is a reasonable method to preserve the structural and logical information. Finally, adversarial machine learning can also evade some of the widely-used malware detection techniques due to the low complexity of the presented neural network structure, as most of these networks leverage the manually specific features [11, 14, 15].

In this paper, we present an NLP inspired-method based on the function call graph. It can detect obfuscated applications while maintaining an excellent performance even when it is under the influence of adversarial examples. In brief, we first design the `opcode2vec`, `function2vec`, and `graph2vec` components to represent instruction, function, and the whole program's information with vectors. We then feed the vectors of graph embedding into the classifier and train it to differentiate between benign and malicious applications, and finally identify the Android malware families. We evaluate our approach on various datasets and show that it outperforms most of the existing frameworks, as we get 99.6% accuracy for malware detection and 98.7% accuracy for malware categorization.

In summary, we make the following main contributions:

- We introduce graph embedding for Android malware detection and categorization.
- We design and implement an NLP-inspired malware detection and categorization framework that can discover obfuscated applications while defending against adversarial machine learning.
- We evaluate the accuracy of our approach using real malicious and benign datasets.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03.
https://doi.org/10.1145/3412841.3442080

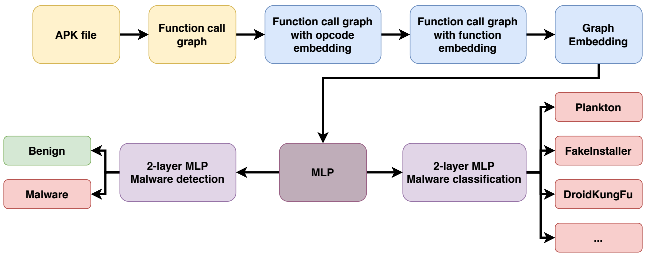


Figure 1: Framework's overall architecture

2 DESIGN

2.1 Overview

We want to develop a framework that leverages neural networks to detect malicious Android applications. To achieve this, we first need to transform the Android opcode, represented as text, to a vector. Then we must continuously convert the functions to the corresponding vectors by function embedding and the Android APK file to a vector as well. Finally, we inject the generated graph embedding into a *Multi-Layer Perception* (MLP) to perform the classification. Figure 1 displays the overall architecture of our framework.

The framework takes an Android app as input and produces the function call graph through static code analysis. The graph nodes represent the functions in an application. Each function includes several basic blocks, and each block includes various Dalvik opcodes. After retrieving the opcodes' embedding, we use the function embeddings sub-module to convert those functions into function vectors. The converted function vectors (node of the call graph) then convert to a final vector by graph embedding, representing the entire graph information. Finally, the framework detects the malicious application through a 2-layer MLP network or classify it into multiple groups such as DroidKungFu, Plankton, and FakeInstaller.

2.2 Opcode Embedding

To simplify the procedure, we replace the instruction (opcode and operands) embedding with the opcode embedding, as the opcode represents the behaviors of Dalvik's instruction and the operands represent the parameters. Dalvik's operands are virtual registers in a virtual machine. Those values are affected by the undergoing usage of Dalvik VM or ART VM. Thus, we cannot enumerate them all. Further, if several malware samples within the family use the same malicious pattern, the opcode itself can capture these behaviors. In theory, our opcode embedding method may suffer from the *operand removal* problem [6]. One significant issue with that is that all the *Invoke-Virtual* instructions[1] have the same embedding vector, no matter what are the targets of the *Invoke-Virtual* instruction.

For opcode embedding, or `opcode2vec`, we map each opcode $op_i \in OP$ (i.e., OP stands for the whole Dalvik opcodes) to a vector of the real number, using `word2vec` [9] with skip-gram. `word2vec` is an excellent feature learning method, based on continuous bag-of-word and skip-gram methods. The skip-gram uses the current opcode to predict the opcodes around it. We trained our `opcode2vec` model with a large corpus of opcodes extracted from real apps.

[1] All the calling instructions such as *invoke-super*, *invoke-direct*, *invoke-static*, and *invoke-interface* suffer from the same problem.

2.3 Function Embedding

In this work, we treat the function embedding similar to the sentence embedding. Overall, we introduce two methods to perform the function embedding, which we describe as follows.

Weighted Mean Function Embedding. We utilize the weighted mean of a non-empty finite multi-set of instruction's opcode to calculate the function embedding. Assuming the function f includes n-opcode and a l-dimensional vector represents each opcode, the weight of the corresponding non-negative weights $w_1, w_2, \ldots, w_n$ is obtained by calculating the average value. Weighted mean function embedding is an easy and straightforward way. However, this weighted method skips the sequence of opcodes. Therefore, we design a follow-up method, which considers the sequence of opcodes.

SIF-Invoked Function Embedding. For this function embedding, we utilize the SIF network [2]. We compute the function embedding $\vec{f}$ by using the sequence of opcodes' vectors, which we get from the `opcode2vec` method. Adapting from the natural language processing, given the discourse vector c_f, the probability of instruction is emitted in the function f is modeled by

$$Pr[i \notin f|\tilde{c_f}] = \alpha p(i) + (1-\alpha)\frac{exp(< c_f, v_i >)}{Z_{\tilde{c_f}}}, \qquad (1)$$

where $c_f = \beta c_0 \tilde{+} (1-\beta)c_f, c_0 \perp c_f$, α and β are scalar hyperparameters, and $Z_{\tilde{c_f}} = \sum_{i \in f} exp(< \tilde{c_f}, v_i >)$ the normalizing constant.

2.4 Graph Embedding, Malware Detection and Identification

After getting the function embedding, we take those generated function embedding as the node embedding of the function call graph; i.e., we perform graph embedding on function call graph level. This way, we convert the graph representation to a vector and take the vector as the neural network-based classifier's input.

For the graph embedding, in our case, the vectors (nodes) of graphs are functions, and the edges are connections among those functions. Each vector (node) contains a set of opcodes inside it. The function embedding constructs each node's feature. Finally, a p-dimensional vector μ_i is associated with each vertex v_i. We use adapted `structure2vec` to update the p-dimensional vector μ_i^{t+1} during the network training dynamically. The graph embedding generates the vector embedding after all iterations, and we use the average aggregation function as our last step to transform the vector embedding to the graph-based function embedding.

After getting our graph embedding for function call graph, we design a two-layer multi-layer layers perception (MLP) as our malware detection and malware categorization system. In our network, malware detection is a binary classification issue. We label malware samples as "1" and benign samples as "-1" at the training step. During testing, we treat all the predictions less than zero as benign and the ones that are more than zero as malware.

$$f(G_h) =< (< g_i, w_{i1} > +b_{i1}), w_{i2} > +b_{i2} \qquad (2)$$

where $w_{i1}, w_{i2} \in R^p$ is the weight of the 2-layer MLP network and $b_{i1}, b_{i2} \in R^p$ is the offset from the origin of the vector space. In this setting, a function call graph G_h is classified as malicious if $f(G_h) > 0$ and as benign if $f(G_h) < 0$.

For the malware categorization, we divide this task into two subtasks. The first one categorizes the malware samples without pre-processing them. We label all the applications with a N-dimensional one-hot vector. The "1" in the one-hot vector stands for the index of the kinds of Android malware. We append one softmax layer, like Equation 3, at the end of MLP and classify the malware to the classification "n", which stands for the type of malicious samples. We treat malware categorization as a multi-class classification issue.

$$f(G_h) = softmax(< (< g_i, w_{i1} > +b_{i1}), w_{i2} > +b_{i2}) \tag{3}$$

Additionally, we enumerate the top-n largest malware families as a pre-processing step and retrieve the malware dataset samples. If the sample is from the indicated malware family, we label it as "1". Otherwise, we label it as "0". With this assumption, we convert the multi-class classification problem to binary classification.

3 EVALUATION

3.1 Datasets and Experimental Setup

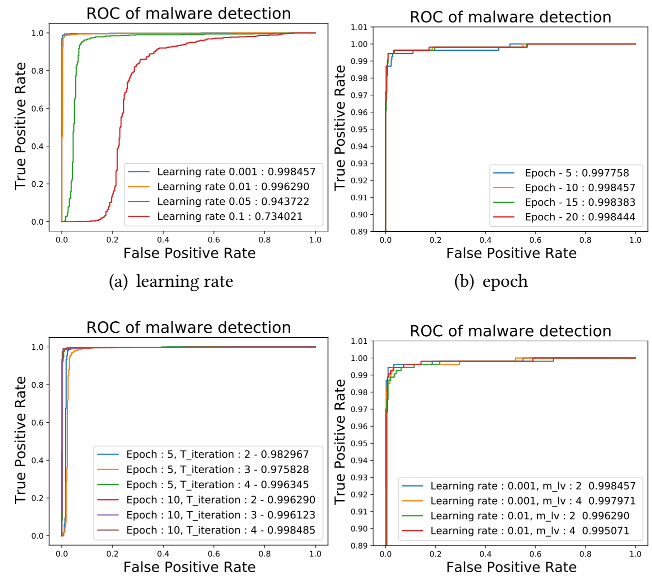
For the evaluation, we utilize four different datasets: (i) `DREBIN` [3], (ii) `AMD` [12], (iii) `PRAGuard` [8], and (iv) `AndroZoo` [1]. Our dataset includes 45,592 malware and 90,313 benign samples. We divide this dataset into training and testing sub-datasets, with 80% of those samples to be training samples and the rest 20% testing samples. For the machine learning classifier setup, we use TensorFlow[2] and scikit-learn[3]. Finally, we train the network with AdamOptimizer and squared difference cross-entropy, and use L2 loss as regularization.

3.2 Experiments

We evaluate our framework through two different tasks: malware detection and malware categorization. For malware detection, we divide our approach into three parts. First, we assess our system with different hyper-parameters. We then compare our trained system with baseline detection algorithms. Finally, we evaluate the robustness of our framework by introducing the adversarial machine learning. In the evaluation, we primarily present the results based on the 64-bit vectors due to space limitation. However, this is very easy to extend other sizes vectors such as 16, 32, 50, 100, 128. For malware categorization, we classify all malware samples into their corresponding families. Meanwhile, we list the top-6 and top-7 malware families and perform malware family classification for those top families. In the following, we provide further details.

Malware Detection. We define the malware detection problem as a binary classification task. That means we have only two types of outputs: *benign* or *malware*. Here we evaluate the malware detection performance using weighted mean function embedding, the SIF-invoked function embedding, and the graph-based methods.

Hyper-parameters: To evaluate the convergence feature of our module, we set the learning rate as {0.001, 0.01, 0.05, 0.1}, various epochs between 5 and 20, t_iteration[4] as {2, 3, 4}, and m_lv[5] as {2, 4}. Figure 2(a) illustrates the ROC of the different learning rate. We found that various learning rates during the training have a large influence on the testing data. With a learning rate of 0.1, our framework only gets 74.4% AUC. With the learning rates of 0.01 and 0.001, the AUC will be 99.62% and 99.85%, respectively. For various training epochs, Figure 2(b) shows the slight differences. Figure 2(c) shows the differences with various t_iteration, under the different epochs. In Figure 2(c), we notice that bigger t_iteration gets better performance because we collect more information from multi-hops. Finally, for the embedding depth, m_lv, our results are the same with `structure2vec`, as displayed in Figure 2(d), which indicates that the two-layers graph network is the best choice.



(a) learning rate (b) epoch

(c) t_iteration (d) m_lv

Figure 2: Hyper-parameters

Comparison: We compare our framework with similar systems. We set our hyper-parameters learning rate as 0.001, training epochs as 10, t_iteration as 2, and m_lv as 2. In more detail, we compare the performance of malware detection with Drebin [3], Droidmat [13], and Adagio [5]. The ROC curves of Drebin, Adagio, and our graph embedding are presented in Figure 3. With our graph embedding methods, we obtain nearly 99.8% AUC of our dataset. To contrast with our method, we get 96.6% AUC with the Drebin method and 89.85% with Droidmat on our mixed dataset. Additionally, Adagio gets a lower AUC value, around 89.02%.[6] Additional details can be found in Table 1. Ge-Mean, and Ge-SIF show the results within our malware detection framework.

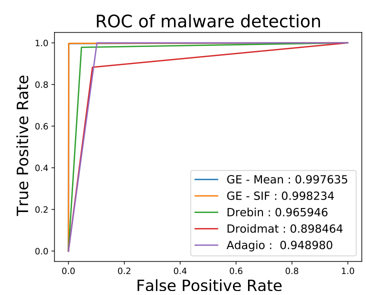


Figure 3: Comparison with related work

[2]https://www.tensorflow.org/
[3]https://scikit-learn.org/
[4]one parameter for the graph embedding, which indicates the n-hop neighbors)
[5]embedding depth, the number of layers in graph deep network

[6]The results of Adagio are a little different from the original work because of the mixed datasets.

Table 1: Comparison with other works of malware detection

Algorithm	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	FPR (%)
Ge-SIF	99.86	99.75	99.75	99.42	0.7
Ge-Mean	99.74	99.92	99.63	99.78	0.4
Drebin	96.58	95.37	97.85	96.59	2.35
Droidmat	89.87	90.89	88.28	89.56	4.36
Adagio	95.0	91.07	100	95.32	5.0

Table 2: Detection rate of obfuscated APK

	ClassEnc.	StrEnc.	Refl.	Triv.	Triv.-Str.	Triv.-Ref.-Str.	Triv.-Ref.-Str.-Class.
PRAGuard[7]	38.0	64.0	96	90.0	50.0	44.0	32.0
Drebin	99.12	98.99	86.58	98.32	98.99	99.32	96.98
Our framework	99.33	98.99	86.58	98.32	98.99	99.32	96.98

Result analysis: After getting the results of malware detection, we analyze them and reconsider those values from several standpoints. First of all, we do our evaluation under the inductive setting. That means we have never seen the test instances during training. Therefore, we do not have problems that indicate the testing dataset influences the training procedure. On the other hand, we consider the influence of the sizes of the samples. As a phenomenon, the sizes of benign samples are generally larger than malware samples. Therefore, various sizes of testing samples would influence the results of the malware detection system. For example, one sizeable benign sample may include the malicious sample's function call graph. As a consequence, our malware detection system will be confused to detect real malware samples. To demonstrate this type of influence, we split our dataset using samples' sizes and evaluated them with different sizes.

Obfuscation: PRAGuard mentions the influence of obfuscated applications on Android malware detection. More precisely, it presents seven types of obfuscation techniques and influenced performance. We evaluate our framework by the PRAGuard dataset. The ROC is illustrated in Figure 4. We compare the detection rate with PRAGuard in Table 2. From the extracted results, we identify that obfuscation does not influence our framework.

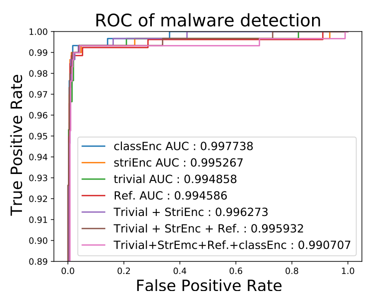


Figure 4: ROC of obfuscated APK

Malware Categorization. In contrast to the malware detection, we divide the malware categorization into two subtasks: a multi-class task and a binary classification issue. On the one hand, we implement a pre-processing step to discover the top-6 largest malware families of the DREBIN dataset (with the limited space, we did not put the AMD results of this sub-task (Table 3) to train and classify those samples. Details about those top-N malware families are shown in the second and third columns of Table 3. All samples from the six largest malware families in the DREBIN set form the first dataset.

Table 3: Family classification results

			5-epoch					10-epoch				
	Family	**Samples**	**Accuracy**	**Precision**	**Recall**	**F1**	**FPR**	**Accuracy**	**Precision**	**Recall**	**F1**	**FPR**
Mean	FakeInstaller	925	99.61	98.78	98.90	99.39	0.57	99.21	98.78	98.78	99.39	0.58
	DroidKungFu	667	99.60	98.10	98.10	99.04	0.50	99.20	98.10	98.06	99.04	0.5
	Plankton	624	99.65	92.31	92.31	96.00	0.37	99.29	92.31	92.31	96.0	0.37
	Opfake	613	99.35	97.21	97.05	98.50	0.82	99.08	97.87	97.86	98.92	0.58
	GinMaster	339	99.64	95.92	95.92	97.91	0.38	99.29	92.31	95.92	97.92	0.39
	BseBridge	330	99.61	96.62	96.62	98.28	0.44	99.14	96.31	96.31	98.12	0.48
SIF	FakeInstaller	925	99.5	98.45	98.45	99.22	0.74	99.0	98.45	97.59	97.60	0.74
	DroidKungFu	667	99.53	97.76	97.76	98.87	0.59	99.06	97.76	98.21	97.16	0.59
	Plankton	624	99.64	92.59	92.59	96.15	0.37	99.29	92.59	97.18	97.30	0.37
	Opfake	613	99.44	97.38	97.38	98.67	0.72	99.22	98.20	97.16	97.16	0.49
	GinMaster	339	99.50	94.8	94.4	97.32	0.55	98.76	92.8	97.87	97.86	0.70
	BseBridge	330	99.47	95.07	95.38	97.63	0.6	99.01	95.69	96.85	96.89	0.56

4 CONCLUSION

In this work, we present a graph embedding-based approach to detect and categorize Android malware. Our method makes use of natural language processing concepts, namely, word2vec, sentence2vec, and document2vec. We represent Android applications based on their function call graph. We train the graph embedding model with a large dataset to differentiate between benign and malicious applications and to identify the Android malware families. Graph embedding is shown to be both efficient and effective, as our framework outperforms several existing works.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 883275.

REFERENCES

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting Millions of Android Apps for the Research Community. In *IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, 2016.

[2] S. Arora, Y. Liang, and T. Ma. A Simple but Tough-To-Beat Baseline for Sentence Embeddings. 2016.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[4] C. Chen, Y. Liu, B. Shen, and J.-J. Cheng. Android Malware Detection Based on Static Behavior Feature Analysis. *Journal of Computers*, 29(6):243–253, 2018.

[5] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *ACM workshop on Artificial intelligence and security*, 2013.

[6] I. U. Haq and J. Caballero. A Survey of Binary Code Similarity, 2019.

[7] C. Li, R. Zhu, D. Niu, K. Mills, H. Zhang, and H. Kinawi. Android Malware Detection Based on Factorization Machine. *arXiv preprint arXiv:1805.11843*, 2018.

[8] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth Attacks: An Extended Insight Into the Obfuscation Effects on Android Malware. *Computers & Security*, 51:16–31, 2015.

[9] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *Advances in neural information processing systems*, 2013.

[10] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. *ACM Transactions on Privacy and Security*, 22(2):1–34, 2019.

[11] Y. Shen and G. Stringhini. Attack2vec: Leveraging Temporal Word Embeddings to Understand the Evolution of Cyberattacks. In *USENIX Security Symposium*, 2019.

[12] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[13] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android Malware Detection Through Manifest and Api Calls Tracing. In *Asia Joint Conference on Information Security*, 2012.

[14] P. Xu, B. Kolosnjaji, C. Eckert, and A. Zarras. MANIS: Evading Malware Detection System on Graph Structure. In *ACM Symposium on Applied Computing*, 2020.

[15] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack Tree Based Android Malware Detection With Hybrid Analysis. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2014.