The ISC Int'l Journal of Information Security
ISeCure

November 2022, Volume 14, Number 3 (pp. 51–59)
http://www.isecure-journal.org

PRESENTED AT THE ISCISC'2022 IN RASHT, IRAN.

Android Malware Detection Using One-Class Graph Neural Networks ☆

Fatemeh Deldar [1], Mahdi Abadi [1,*], and Mohammad Ebrahimifard [1]
[1] *Department of Computer Engineering, Tarbiat Modares University, Tehran, Iran.*

ARTICLE INFO.

Keywords:
Android Malware Detection, Attributed Function Call Graph, Graph Convolutional Layer, One-Class Classification, Semi-Supervised Deep Learning, Stacked Graph Autoencoder.

Type:
Research Article

doi:
`10.22042/isecure.2022.14.3.6`

dor:
`20.1001.1.20082045.2022.14.3.6.0`

ABSTRACT

With the widespread use of Android smartphones, the Android platform has become an attractive target for cybersecurity attackers and malware authors. Meanwhile, the growing emergence of zero-day malware has long been a major concern for cybersecurity researchers. This is because malware that has not been seen before often exhibits new or unknown behaviors, and there is no documented defense against it. In recent years, deep learning has become the dominant machine learning technique for malware detection and could achieve outstanding achievements. Currently, most deep malware detection techniques are supervised in nature and require training on large datasets of benign and malicious samples. However, supervised techniques usually do not perform well against zero-day malware. Semi-supervised and unsupervised deep malware detection techniques have more potential to detect previously unseen malware. In this paper, we present MalGAE, a novel end-to-end deep malware detection technique that leverages one-class graph neural networks to detect Android malware in a semi-supervised manner. MalGAE represents each Android application with an attributed function call graph (AFCG) to benefit the ability of graphs to model complex relationships between data. It builds a deep one-class classifier by training a stacked graph autoencoder with graph convolutional layers on benign AFCGs. Experimental results show that MalGAE can achieve good detection performance in terms of different evaluation measures.

© 2022 ISC. All rights reserved.

1 Introduction

Malware is a malicious program designed to compromise a computer system. Malware attacks have a wide range and can interfere with the normal operation of the computer system in various ways. For example, they can lead to stealing confidential information, gaining unauthorized access to system resources, bringing down servers, and damaging files.

* Corresponding author.

☆ The ISCISC'2022 program committee effort is highly acknowledged for reviewing this paper.

Email addresses: `f.deldar@modares.ac.ir`, `abadi@modares.ac.ir`, `m.ebrahimifard@modares.ac.ir`

ISSN: 2008-2045 © 2022 ISC. All rights reserved.

There are different types of malware, such as viruses, backdoors, rootkits, and ransomware.

Android is the dominant mobile operating system worldwide, leading the market with more than 70% market share in August 2022 [1]. The popularity of Android is growing exponentially, and this, along with openness, has made it an attractive target for a large number of malware authors. According to a recent report, more than 3 million new malware samples targeting Android devices were discovered in 2021 [2]. There is an urgent demand for developing malware detection techniques to deal with the massive growth of Android malware. Traditional malware detection techniques cannot cope with this problem due to the rapid evolution of complex malware or the emergence of zero-day malware.

Generally, zero-day malware is previously unseen malware that often exploits zero-day vulnerabilities. Zero-day vulnerabilities are exploited in the wild before the patch is released or widely deployed. Among the various malware, zero-day malware is the most dangerous one that can cause considerable and irreparable damage. This is because zero-day malware commonly has a newly-encountered behavior that has not been seen before; therefore, there is no predesigned and documented defense mechanism against its malicious activities. As a result, zero-day malware detection is a critical and problematic issue and is the highest priority of malware detection techniques.

Most of the existing malware detection techniques use stored malware characteristics. For example, signature-based malware detection is a proven and widely-used technique that simply works by identifying signatures related to previously known malware. However, an obvious weakness of such techniques is that they are inherently limited to detecting already known malware. As a result, they often fail to detect zero-day exploits or advanced and complex threats that circumvent known signatures [3].

In recent years, deep learning has become a popular and dominant machine learning technique and can achieve outstanding achievements in various domains. The multi-layered and in-depth structure of deep learning models helps to capture the intrinsic properties of complex and highly nonlinear data and allows the features to be automatically learned with multiple levels of abstraction. However, traditional machine learning techniques are limited by the manually-crafted features from the raw data [4, 5]. Thus, deep learning techniques are more suitable than traditional machine learning techniques for various applications, including malware detection. In this regard, research focusing on deep learning to detect malware has attracted more attention. Meanwhile, some deep-learning studies have addressed zero-day malware detection.

Generally, deep learning algorithms can be classified into three categories: supervised, semi-supervised, and unsupervised. Supervised deep learning algorithms summarize complex relationships among features in a labeled training dataset that discriminate between data samples. The main disadvantage of these algorithms is that they often require a lot of labeled data samples to train the model. Semi-supervised deep learning algorithms usually combine both labeled and unlabeled data samples to build a model from them. Unsupervised deep learning algorithms capture the high-level correlation of input data samples without knowing their labels. Gathering a large number of labeled data samples usually takes a lot of time and labor. Especially in the case of zero-day malware, which is still unknown, collecting labeled data samples does not seem possible. Hence, semi-supervised and unsupervised techniques have more potential to detect previously unknown malware. However, few deep learning techniques adopt semi-supervised and unsupervised learning for malware detection.

Graph neural networks (GNNs) [6] have emerged as a flexible and powerful model for machine learning and an effective framework for representation learning over graph-structured data. They follow a convolutional architecture that allows them to inherit the desired properties of convolutional neural networks (CNNs) while applying directly to graph-structured data. GNNs use a message-passing framework in which the representation vector (embedding) of a node is recursively calculated by aggregating the representation vectors of its neighbors. The main idea is to combine graph structure and node features to better learn how to represent graph-structured data. Due to their remarkable performance and high interpretability in various graph-based tasks at the node, edge, and graph level, GNNs have become a widely used graph analysis tool. In particular, in recent years, they have gained popularity in cybersecurity, especially in malware detection tasks [7–10]. Many malware detection models [11, 12] emphasize graph-structured data to describe the malware's behavior, thanks to the ability of graphs to model complex relationships between data. But almost all of these models rely on supervised deep learning algorithms to detect malware.

Autoencoders are neural networks that learn a new representation of the input data, usually known as the latent representation. An autoencoder is composed of an encoder and a decoder. The encoder compresses the input data by learning its latent representation,

while the decoder tries to reconstruct the original input data from the latent representation. The graph autoencoder is a GNN framework for semi-supervised and unsupervised learning on graph-structured data. Recently, graph autoencoders have attracted more attention for graph embedding due to their great potential for dimensionality reduction. However, it is not straightforward to apply the idea of autoencoder to graph-structured data because of their irregular structure. In particular, graph-structured data have rich and complex information on which content and structure are dependent. Therefore, it is challenging to effectively integrate both structure and content information into a unified framework.

In this paper, we present MalGAE, a novel end-to-end malware detection technique that leverages attributed function call graphs (AFCGs) and semi-supervised deep learning to detect previously unknown Android malware. MalGAE stacks multiple graph convolutional layers in the autoencoder architecture to distinguish between benign and malicious Android applications. This is done in a semi-supervised manner where there is no need to access the labels of malicious samples, and the training is done only with a small set of benign samples. In other words, MalGAE builds a deep one-class classifier by training a stacked graph autoencoder on benign samples and then uses it to detect Android malware.

In the following, we list the main contributions of this paper:

- To the best of our knowledge, we are the first to apply semi-supervised deep learning to graph neural networks (GNNs) for Android malware detection.
- We present MalGAE, an end-to-end malware detection technique that trains a stacked graph autoencoder with graph convolutional layers to distinguish between benign and malicious Android applications in a semi-supervised manner. We represent each Android application with an attributed function call graph (AFCG) to benefit the ability of graphs to model complex relationships between data.
- Through experiments on an Android malware dataset, we show that, despite being semi-supervised, MalGAE can achieve good detection performance in terms of different evaluation measures. Especially, MalGAE achieves a recall of 81.99 and a false positive rate of 3.27.

The rest of this paper is organized as follows. Section 2 briefly reviews related work, and Section 3 introduces some background information about Android. Section 4 introduces MalGAE and describes how it detects Android malware in a semi-supervised manner. Section 5 reports our experimental results, and finally, Section 6 summarizes and concludes the paper.

2 Related Work

The concept of GNNs was first introduced by Gori *et al.* [13] in 2005 and further elaborated by Scarselli *et al.* [14] in 2009. In recent years, GNNs have received more attention for malware detection [7–10, 15, 16]. In this section, we review the latest state-of-the-art techniques for malware detection using GNNs.

Yan *et al.* [7] proposed a malware classification system that uses the deep graph convolutional neural network (DGCNN) [17] to embed structural information inherent in control flow graphs (CFGs) for classifying malware programs. Then, Xu *et al.* [8] introduced a graph embedding technique for Android malware detection and categorization. They represented Android applications based on their function call graph (FCG) and designed the opcode2vec, function2vec, and graph2vec components to represent instruction, function, and the whole application's information with vectors. They then fed the obtained vectors into an MLP classifier and trained it to distinguish malicious from benign applications and finally identify the Android malware families. Later, Li *et al.* [9] developed GSFDroid, a system for familial analysis of Android malware. They first constructed an FCG for each malware sample and embedded the nodes of all FCGs into a continuous and low-dimensional space using graph convolutional networks (GCNs). They then employed a two-phase familial analysis technique to improve the overall performance. For this purpose, an MLP classifier was trained to predict the family of unlabeled malware samples, and a pre-defined threshold was set for its confidence score. The samples with a high uncertainty score were given to an unsupervised clustering algorithm that performed familial analysis to discover new malware families. The malware samples grouped in a cluster were considered the same family.

Subsequently, Gao *et al.* [15] proposed GDroid, an approach for Android malware detection and familial classification based on GNNs. They mapped Android applications and APIs into a heterogeneous graph in which the App-to-API edges were established by the invocation relationships, and the API-to-API edges were built by the API usage patterns. The heterogeneous graph was then fed into a GCN model [18], iteratively generating node embeddings that incorporate topological structure and node features. The unlabeled applications were eventually classified by their final embeddings. Then, Zhang *et al.* [10] proposed HyGNN-Mal, an Android malware detection technique based on a hybrid GNN. They analyzed

Android applications at the source code level and used abstract syntax trees (ASTs) to represent their structure information. Meanwhile, they extracted typical static features, permissions, and APIs at the program level. They also used a deep traversal tree neural network (Deep-TNN) to process ASTs and a bidirectional GRU to handle permissions and API sequences. Later, Wu *et al.* [16] proposed DeepCatra, a multi-view deep learning model for Android malware detection that consists of a bidirectional LSTM and a GNN layer. They extracted a set of critical APIs from the known vulnerability repositories. They then constructed a call graph for each Android application and computed call traces reaching the critical APIs. Based on the call traces of each application, they built the data embedding for each view of learning. They also selected the nearest opcode sequences leading to the critical API calls for the embedding of bidirectional LSTM and extended the critical edges with the edges related to inter-component communications to build the global abstract flow graph for the embedding of GNN. The output vectors of the bidirectional LSTM and GNN layers were merged with a fully connected layer to produce the classification results.

All the techniques described above use a supervised training process in which input samples are labeled as benign or malicious. However, supervised techniques usually cannot perform well in detecting previously unseen malware. In this paper, we present a semi-supervised deep learning technique for Android malware detection that utilizes the power of one-class GNNs along with AFCGs to increase detection performance. Since only benign Android applications are involved in the training process of our technique, it can be effective in detecting previously unseen Android malware as well.

3 Android

Android is an open-source mobile operating system based on the Linux kernel. One major component of the Android platform is the Android runtime (ART). It introduces an ahead-of-time (AOT) compilation that compiles entire applications into native machine code upon installation. Most Android applications are written in Java. The written code is first compiled into Java bytecode. Then, a DEX compiler converts the Java bytecode to Dalvik bytecode. Further, ART translates the Dalvik bytecode into native machine code. Each Android application is packaged into an Android package kit (APK) with the `.apk` file extension. This packed file is a zipped archive that includes the Dalvik bytecode, resources, assets, certificates, and Android configuration file (`AndroidManifest.XML`).

Each Android application includes four types of components: activity, service, broadcast receiver, and content provider. The activity component is a Java class that works in the foreground and interacts with the user. The service component is invisible to the user and makes the application run in the background. The broadcast receiver component mainly receives broadcast events from the operating system. The content provider component is a database that provides a structured access interface for data sharing across applications.

4 MalGAE

In this section, we present MalGAE, a novel deep malware detection technique that leverages one-class graph neural networks (GNNs) to detect Android malware in a semi-supervised manner. MalGAE first extracts an attributed function call graph (AFCG) for each Android application in the preprocessing step. Then, in the model construction step, it builds a deep one-class classifier by training a stacked graph autoencoder to detect malicious AFCGs. In the following, we describe the details of MalGAE.

4.1 Preprocessing

We represent each Android application with an AFCG to preserve its structural and functional characteristics and examine how its various functions interact with each other. We consider an AFCG as a directed graph $G = (V, E)$, where each node in V represents a function, and each edge in E represents the calling relationship between two functions. We denote the adjacency matrix of G as $\mathbf{A} \in \mathbb{Z}^{n \times n}$, where $n = |V|$ is the number of nodes in G. The nodes in V represent internal and API functions in the application. We assume that each node in V is associated with a c-dimensional feature vector. Therefore, we use $\mathbf{X} \in \mathbb{R}^{n \times c}$ to denote the feature matrix for all the nodes in G. The feature vector of each API node (i.e., nodes corresponding to API functions) is obtained by applying one-hot encoding to its corresponding function name. All non-API nodes (i.e., nodes corresponding to internal functions written by the developer to implement specific functionality) are assigned the same one-hot encoded feature vector.

4.2 Model Construction

Among the various deep learning models, autoencoders are the most commonly used model for applications where the label information of data samples is not available. Due to their reconstruction nature, autoencoders have the potential to be used for anomaly-based malware detection.

Anomaly-based malware detection techniques typically use one-class classification that builds one-class

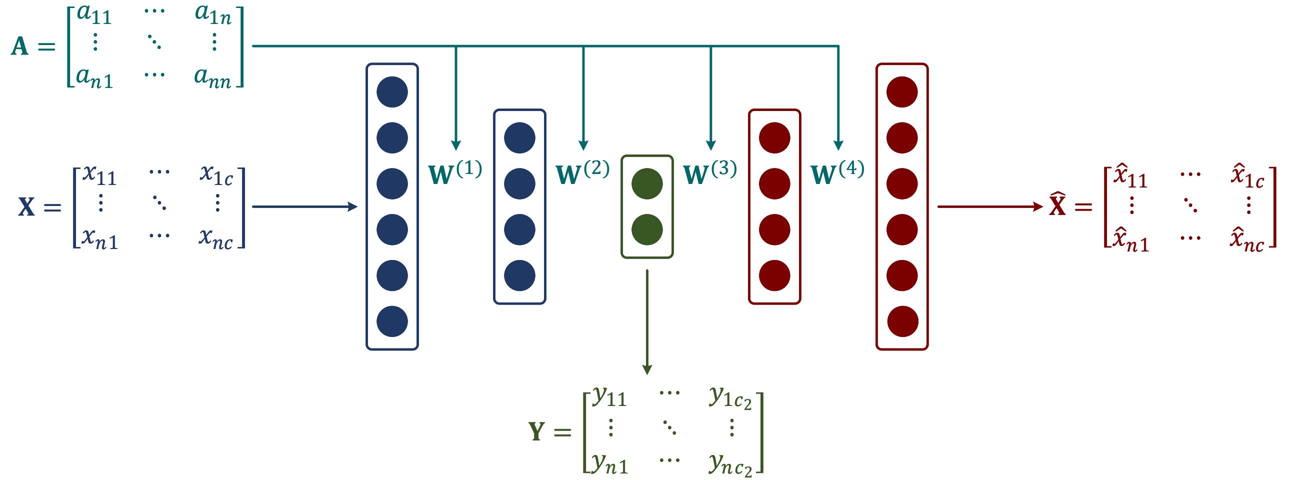

Figure 1. A stacked graph autoencoder with four graph convolutional layers

classifiers only from benign samples. Since these techniques do not require labeled malicious samples and utilize only a small set of benign samples, they are semi-supervised malware detection techniques. They mainly detect malware by finding abnormal behavior compared to the benign class. The key advantage of such malware detection techniques is their ability to detect zero-day malware.

MalGAE leverages a stacked graph autoencoder with graph convolutional layers to distinguish between benign and malware samples. It takes a set of AFCGs as input, learns their latent representations, and finally reconstructs their feature matrices. The main goal is to encode each AFCG $G = (V, E)$ into an embedding vector in such a way that the reconstructed feature matrix $\hat{\mathbf{X}}$ is as close as possible to the original feature matrix $\mathbf{X}$.

MalGAE reconstructs each input AFCG based on reconstructing its feature matrix. Graph reconstruction can also be performed by reconstructing the adjacency matrix. In this case, the decoder does not use the feature matrix at all and cannot be trained. However, this can sometimes lead to a reduction in the efficiency of graph reconstruction [19, 20]. Therefore, we reconstruct each AFCG based on reconstructing the feature matrix to overcome this limitation. In this case, the decoder does the opposite of what the encoder does.

The graph encoder stacks multiple graph convolutional layers, which can be recursively defined as

$$\mathbf{Z}^{(l)} = f^{(l)}\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{Z}^{(l-1)}\mathbf{W}^{(l)}\right), \tag{1}$$

where $\tilde{\mathbf{A}} = \mathbf{A}+\mathbf{I}_n$ is the augmented adjacency matrix of G with added self-loops, $\mathbf{I}_n$ is the identity matrix of order n, and $\tilde{\mathbf{D}}$ is the augmented diagonal degree matrix of G with $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. Also, $\mathbf{W}^{(l)} \in \mathbb{R}^{c_{l-1}\times c_l}$ is the trainable weight matrix, $f^{(l)}(\cdot)$ is the activation function, and $\mathbf{Z}^{(l)} \in \mathbb{R}^{n\times c_l}$ is the output of layer l with $\mathbf{Z}^{(0)} = \mathbf{X}$, where c_l is the output dimension of layer l.

The output of the graph encoder, called the latent representation, is given to the graph decoder to reconstruct $\mathbf{X}$. The graph decoder stacks the same number of graph convolutional layers as the graph encoder, arranged in the reverse order of the graph encoder. More specifically, the graph decoder behaves the opposite of the graph encoder as it works to reconstruct the feature matrix $\mathbf{X}$ as closely as possible. By doing so, the stacked graph autoencoder uses both graph structure and node features in the encoding and decoding steps. Assuming that the stacked graph autoencoder has k layers, the latent representation $\mathbf{Y}$ is represented as

$$\mathbf{Y} = \mathbf{Z}^{(k/2)}. \tag{2}$$

We define the reconstruction loss of the stacked graph autoencoder as

$$\mathcal{L}(G) = \frac{1}{n}\|\mathbf{X}-\hat{\mathbf{X}}\|_F^2+\lambda\sum_{i=1}^{k}\|\mathbf{W}^{(i)}\|_F^2, \tag{3}$$

where $\hat{\mathbf{X}}$ is the reconstructed feature matrix of nodes and $\|\cdot\|_F$ denotes the Frobenius norm. The second term is a weight decay regularizer with hyperparameter $\lambda > 0$. Figure 1 shows a simple stacked graph autoencoder with four graph convolutional layers. There are two layers for the graph encoder and two layers for the graph decoder.

After the training process of the stacked graph autoencoder is completed, the latent representation contains the embedding vectors of all nodes. The embedding vector of the AFCG is obtained by applying an average pooling layer to the latent representation,

which averages all embedding vectors of its nodes. Formally, the output of the average pooling layer, represented by $\mathbf{y}$, is an embedding vector of length $c_{k/2}$ calculated as

$$\mathbf{y} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{Y}_i, \tag{4}$$

where $\mathbf{Y}_i$ denotes the ith row of $\mathbf{Y}$.

4.3 Malware Detection

We use only benign AFCGs in the model construction step. Therefore, the reconstruction loss can be interpreted as an anomaly score, where AFCGs incurring a larger reconstruction loss are more likely to be malicious. In more detail, we divide the training dataset, which contains only benign AFCGs, into two parts, and use the first part for training the stacked graph autoencoder. After the training process is completed, we give the second part as input to the trained model and calculate the anomaly score of each AFCG. We then sort these anomaly scores in descending order and select the νth percentile of these values as the rejection threshold δ, where $\nu > 0$ is a hyperparameter. We refer to ν as the training rejection rate or simply the rejection rate.

The rejection threshold δ is used to decide whether an input AFCG is malicious or benign. Each AFCG is first given to the trained model, and its anomaly score is calculated. If this anomaly score is less than or equal to δ, that AFCG is labeled as benign and otherwise as malicious.

5 Experiments

In this section, we discuss the detection performance of MalGAE for different parameter settings and compare it to a baseline case that uses a simple stacked autoencoder instead of a stacked graph autoencoder.

5.1 Dataset

In our experiments, we used the dataset provided by Chew *et al.* [21]. We selected 141 Android malware samples and 498 benign Android applications from this dataset. The malware samples belonged to five different families of crypto-ransomware. We converted the APK file of each application to an FCG using Androguard [22]. We then generated an AFCG by associating a feature vector with each node. The feature vector of each API node was obtained by applying one-hot encoding to its corresponding function name. Since there were about 7200 different API functions in the Android applications of this dataset, the length of each feature vector was about 7200.

5.2 Evaluation Measures

We evaluated the detection performance of MalGAE using five common measures: precision (PR), recall (RE), accuracy (ACC), F1-score (FS), and false positive rate (FPR). These measures are derived from the true positive (TP), false positive (FP), true negative (TN), and false negative (FN) metrics. TP denotes the number of samples correctly detected as malicious. FP denotes the number of samples incorrectly detected as malicious. TN denotes the number of samples correctly detected as benign. FN denotes the number of samples incorrectly detected as benign. Formally, the above measures are calculated as

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}, \tag{5}$$

$$PR = \frac{TP}{TP + FP}, \tag{6}$$

$$RE = \frac{TP}{TP + FN}, \tag{7}$$

$$FS = 2 \times \frac{PR \times RE}{PR + RE}, \tag{8}$$

$$FPR = \frac{FP}{FP + TN}. \tag{9}$$

Obviously, for ACC, PR, RE, and FS being close to one and for FPR being close to zero indicates better detection performance.

5.3 Experimental Results

We conducted a 5-fold cross-validation to measure the detection performance of MalGAE. For this purpose, we shuffled the 498 benign applications, selected 400 of them for training, and left the rest for testing. So the testing dataset contained 239 Android applications, 98 of which were benign, and the rest were malicious. Then, we divided the training dataset into five equal-sized subsets. To provide better generalization, MalGAE was trained and tested in five rounds. In each round, four subsets were treated as the training set, and one subset was considered as the validation set for selecting the rejection threshold. The training of each round was done in 50 epochs, and after the training process was completed, the detection performance of the trained model was measured using the testing dataset.

In the basic experimental setup, we trained a stacked graph autoencoder with four layers (a 2-layer graph encoder followed by a 2-layer graph decoder). The hidden layer sizes were 128, 32, 32, and 128. We added a dropout layer of rate 0.1 and a ReLU activation function after each graph convolutional layer.

Table 1. Evaluation measures of MalGAE (in percent) for different values of ν (The results are averaged over five folds)

ν	ACC	PR	RE	FS	FPR
0.01	78.33	78.72	64.40	70.72	1.63
0.03	88.03	97.35	81.99	88.87	3.27
0.05	87.70	96.15	82.55	88.68	4.90
0.10	87.78	91.73	87.23	89.39	11.43
0.20	84.35	84.37	90.21	87.18	24.08

The trained model was optimized by the AdamW optimizer [23] with a learning rate of 0.001 and a weight decay of 0.0005. We also set the rejection rate ν to 0.03. It should be noted that when we studied the impact of one parameter, we fixed the others to be the same as in the basic experimental setup.

Impact of rejection rate. In the first set of experiments, we evaluated the impact of the rejection rate ν on the detection performance of MalGAE. Table 1 shows the obtained results under different values of ν. From the table, we see that increasing ν increases RE at the expense of increasing FPR, which may be undesirable. This is because the higher the value of ν, the more Android applications are detected as malicious. Among these applications, some have been correctly detected as malicious, which increases RE, and others have been incorrectly detected as malicious, which increases FPR. We also observe that MalGAE achieves a good trade-off between different evaluation measures by choosing $\nu = 0.03$. This setting, while keeping FPR at about 3%, can achieve an RE of about 82% and a PR as high as 97.35%.

Impact of graph convolutional layers. In the second set of experiments, we evaluated the impact of the number of convolutional layers k on the detection performance of MalGAE. Table 2 shows the obtained results under different values of k. The first row of this table shows the results for a 2-layer model (one layer for the graph encoder and one layer for the graph decoder) with hidden layer sizes of 32 and 32. In fact, in this case, the stacked graph autoencoder is converted to a graph autoencoder. As can be seen, the 2-layer model has lower detection performance than the 4-layer model. This is because a stack of graph autoencoders is usually better than a simple graph autoencoder in reducing the feature vector dimension and extracting the latent representation. The third row of Table 2 shows the results for a 6-layer model (three layers for the graph encoder and three layers for the graph decoder) with the hidden layer sizes of 256, 128, 32, 32, 128, and 256. As can be seen, the 6-layer model does not cause a noteworthy increase in detection performance compared to the 4-layer model. However, it requires more training time. Therefore,

Table 2. Evaluation measures of MalGAE (in percent) for different values of k (The results are averaged over five folds)

k	ACC	PR	RE	FS	FPR
2	72.55	93.14	57.02	69.23	5.10
4	88.03	97.35	81.99	88.87	3.27
6	88.62	97.02	83.26	89.61	3.67

we conclude that the 4-layer model can provide a better trade-off between detection performance and training time than other models.

5.4 Comparison

Here we compare MalGAE with a baseline model that uses a simple stacked autoencoder instead of a stacked graph autoencoder. In other words, the architecture of this baseline model (hereafter referred to as SAE) is similar to MalGAE, except that graph convolutional layers are not used. Figure 2 shows this comparison for different evaluation measures under two rejection rates of 0.03 and 0.05. From the figure, we observe that MalGAE significantly outperforms SAE for all evaluation measures. The reason is that MalGAE leverages graph convolutional layers in which each node collects information from its neighbors; thus, the graph structure and node features are combined to better learn the representation of AFCGs. Therefore, autoencoders with graph convolutional layers can better reconstruct AFCGs due to the power of GNNs in building end-to-end deep learning models on graph-structured data.

6 Conclusion

In this paper, we have presented MalGAE, a semi-supervised Android malware detection technique that does not require access to the labels of malicious samples during the training process. Such techniques have good potential for detecting previously unseen or zero-day malware. MalGAE uses graph convolutional layers to build a stacked graph autoencoder. This allows it to work directly on the graph-structured data in an end-to-end manner. Thanks to the ability of graphs to model complex relationships between data, graph-structured data have become increasingly popular in recent years to represent malware. MalGAE represents each Android application with an AFCG, where each node represents a function, and each edge represents the calling relationship between two functions. The nodes represent internal and API functions in the application and are associated with a feature matrix. The feature vector of each API node is obtained by applying one-hot encoding to its corresponding function name. All non-API nodes are assigned the same one-hot encoded feature vector.

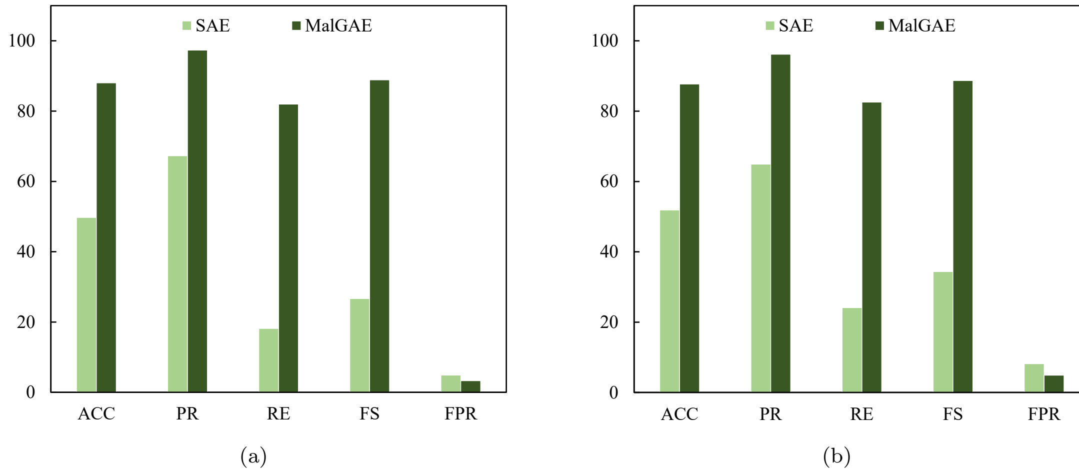

Figure 2. Comparison of SAE and MalGAE for different evaluation measures under two rejection rates: (a) $\nu = 0.03$, (b) $\nu = 0.05$

We have performed empirical experiments to evaluate the detection performance of MalGAE in terms of different evaluation measures. Experimental results have shown that, despite being semi-supervised, MalGAE can achieve good detection performance. In the future, we plan to explore more graph structures and node features to improve the detection performance of semi-supervised and unsupervised deep malware detection techniques.

Acknowledgment

This work was jointly supported by the Iran National Science Foundation (INSF) and Iran's National Elites Foundation (INEF) under Grant 4000822.

References

[1] Federica Laricchia. Market share of mobile operating systems worldwide 2012-2022. https://www.statista.com/statistics/272698/, August 2022.

[2] AV-TEST. Malware statistics & trends report. https://www.av-test.org/en/statistics/malware, 2022.

[3] Asghar Tajoddin and Mahdi Abadi. RAMD: Registry-based anomaly malware detection using one-class ensemble classifiers. *Applied Intelligence*, 49(7):2641–2658, July 2019.

[4] Jun Zhang, Yang Xiang, Yu Wang, Wanlei Zhou, Yong Xiang, and Yong Guan. Network traffic classification using correlation information. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):104–117, January 2013.

[5] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. A survey of Android malware detection with deep neural models. *ACM Computing Surveys*, 53(6):1–36, November 2021.

[6] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021.

[7] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 52–63, Portland, OR, USA, June 2019. IEEE.

[8] Peng Xu, Claudia Eckert, and Apostolis Zarras. Detecting and categorizing Android malware with graph neural networks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 409–412, Virtual Event, Republic of Korea, March 2021. ACM.

[9] Qian Li, Qingyuan Hu, Yong Qi, Saiyu Qi, Xinxing Liu, and Pengfei Gao. Semi-supervised two-phase familial analysis of Android malware with normalized graph embedding. *Knowledge-Based Systems*, 218:106802, April 2021.

[10] Chunyan Zhang, Qinglei Zhou, Yizhao Huang, Ke Tang, Hairen Gui, and Fudong Liu. Automatic detection of Android malware via hybrid graph neural network. *Wireless Communications and Mobile Computing*, 2022:7245403, May 2022.

[11] Xinjun Pei, Long Yu, and Shengwei Tian. AMalNet: A deep learning framework based on graph convolutional networks for malware detection. *Computers & Security*, 93:101792, June 2020.

[12] Xiao-Wang Wu, Yan Wang, Yong Fang, and Peng Jia. Embedding vector generation based on function call graph for effective malware detection and classification. *Neural Computing and Applications*, 34(11):8643–8656, June 2022.

[13] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings of the 2005 IEEE Inter-*

national Joint Conference on Neural Networks, pages 729–734, Montreal, QC, Canada, July 2005. IEEE.

[14] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

[15] Han Gao, Shaoyin Cheng, and Weiming Zhang. GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, 106:102264, July 2021.

[16] Yafei Wu, Jian Shi, Peicheng Wang, Dongrui Zeng, and Cong Sun. DeepCatra: Learning flow- and graph-based behaviors for Android malware detection. *arXiv preprint arXiv:2201.12876*, pages 1–12, July 2022.

[17] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the 32th AAAI Conference on Artificial Intelligence*, pages 4438–4445, New Orleans, LA, USA, February 2018. AAAI Press.

[18] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations*, pages 1–14, Toulon, France, April 2017.

[19] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2609–2615, Stockholm, Sweden, July 2018. AAAI Press.

[20] Yunhao Ge, Yunkui Pang, Linwei Li, and Laurent Itti. Graph autoencoder for graph compression and representation learning. In *Proceedings of the 9th International Conference on Learning Representations*, pages 1–9, Vienna, Austria, May 2021. OpenReview.

[21] Christopher Jun-Wen Chew, Vimal Kumar, Panos Patros, and Robi Malik. ESCAPADE: Encryption-type-ransomware: System call based pattern detection. In Mirosław Kutyłowski, Jun Zhang, and Chao Chen, editors, *Network and System Security*, Lecture Notes in Computer Science, pages 388–407. Springer International Publishing, Cham, Switzerland, 2020.

[22] Anthony Desnos. Androguard. https://github.com/androguard/androguard, 2022.

[23] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Proceedings of the 7th International Conference on Learning Representations*, pages 1–18, New Orleans, LA, USA, May 2019.

Fatemeh Deldar received a B.Sc. degree in computer engineering from Kharazmi University in 2007. She also received her M.Sc. and Ph.D. degrees from Ferdowsi University of Mashhad and Tarbiat Modares University in 2010 and 2019, respectively. She is currently a postdoctoral researcher in the Department of Computer Engineering at Tarbiat Modares University. Her main research interests are malware detection, deep learning, and data privacy.

Mahdi Abadi received a B.Sc. degree in computer engineering from the Ferdowsi University of Mashhad in 1998. He also received his M.Sc. and Ph.D. degrees from Tarbiat Modares University in 2001 and 2008, respectively. He is currently an associate professor in the Department of Computer Engineering at Tarbiat Modares University. His main research interests are malware detection, deep anomaly detection, and network security.

Mohammad Ebrahimifard received a B.Sc. degree in computer engineering from the Shahid Bahonar University of Kerman in 2015. He is currently an M.Sc. student in the Department of Computer Engineering at Tarbiat Modares University. His main research interests are ransomware detection and deep learning.