2022 12th International Conference on Computer and Knowledge Engineering (ICCKE) | 978-1-6654-7613-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCKE57176.2022.9960076

Android Malware Detection Using Supervised Deep Graph Representation Learning

Fatemeh Deldar
Department of Computer Engineering
Tarbiat Modares University
Tehran, Iran
f.deldar@modares.ac.ir

Mahdi Abadi
Department of Computer Engineering
Tarbiat Modares University
Tehran, Iran
abadi@modares.ac.ir

Mohammad Ebrahimifard
Department of Computer Engineering
Tarbiat Modares University
Tehran, Iran
m.ebrahimifard@modares.ac.ir

***Abstract*—Despite the continuous evolution and significant improvement of cybersecurity mechanisms, malware threats remain one of the most important concerns in cyberspace. Meanwhile, Android malware plays a big role in these ever-growing threats. In recent years, deep learning has become the dominant machine learning technique for malware detection and continues to make outstanding achievements. Deep graph representation learning is the task of embedding graph-structured data into a low-dimensional space using deep learning models. Recently, autoencoders have proven to be an effective way for deep representation learning. However, it is not straightforward to apply the idea of autoencoder to graph-structured data because of their irregular structure. In this paper, we present DroidMalGNN, a novel deep learning technique that combines autoencoders with graph neural networks (GNNs) to detect Android malware in an end-to-end manner. DroidMalGNN represents each Android application with an attributed function call graph (AFCG) that allows it to model complex relationships between data. For more efficiency, DroidMalGNN performs graph representation learning in a supervised manner where two autoencoders are trained with benign and malicious AFCGs separately. In this way, it generates two informative embedding vectors for each AFCG in a low-dimensional space and feeds them into a dense neural network to classify the AFCG as benign or malicious. Our experimental results show that DroidMalGNN can achieve good detection performance in terms of different evaluation measures.**

***Index Terms*—Android application, attributed function call graph, autoencoder, graph neural network, graph representation learning, malware detection**

I. Introduction

Malware is any program intentionally designed to harm computer systems or gain illegal access to them. In recent years, malware attacks have become a serious security threat and caused a lot of damage. These attacks have a wide range and can interfere with the normal operation of the computer system in different ways, such as stealing confidential information, making some resources unavailable, and disrupting services. There are different types of malware, such as viruses, Trojans, backdoors, worms, rootkits, ransomware, and spyware. Due to the rapid growth of all types of malware, their quick and accurate detection is one of the most important concerns in cybersecurity. Today, there are various anti-malware engines to detect and remove known malware. These malware detection engines maintain a database of known malware signatures and update their database whenever unknown malware is found. The main weakness of such malware detection engines is that they are inherently limited to detecting already known malware. As a result, they often fail to detect zero-day exploits or the most advanced malware like metamorphic, polymorphic, and obfuscated malware [1].

Android is an open-source and Linux-based operating system for mobile devices such as smartphones and tablets. It is the dominant mobile operating system worldwide, leading the market with more than 70% market share in August 2022 [2]. 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 [3]. 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.

In recent years, deep learning has become a popular and powerful machine learning technique for learning different types of data in different domains. The multilayer structure of deep learning models allows them to deal with high-dimensional data and automatically learn deeper features with multiple levels of abstraction. However, traditional machine learning techniques are limited in processing data in their raw form and require a lot of human intervention. Deep learning techniques are more suitable than traditional machine learning techniques for various classification tasks, including malware detection. Hence, research focusing on deep learning for malware detection, especially Android malware, has recently received more attention.

Graph neural networks (GNNs) [4] are a popular and flexible class of machine learning models that extend convolutional neural networks (CNNs) to graph-structured data by facilitating the learning of relationships between graph elements. They have recently attracted a lot of attention due to their wide range of applications in different domains, where data are naturally explained by graphs. Most GNNs use a message-passing scheme, where a node's information is iteratively updated by aggregating information from its neighbors. In fact, due to their deep nature, GNNs can have

978-1-6654-7613-3/22/$31.00 ©2022 IEEE

multiple layers, each corresponding to one of the iterations of the update process. This means that each layer allows information to be propagated from the nodes once more. In recent years, GNNs have gained popularity in cybersecurity, especially in malware detection tasks [5]–[7]. Many malware detection models emphasize graph-structured data to describe the malware's behavior, thanks to the ability of graphs to model complex relationships between data.

Due to its superior ability in feature learning through deep multilayer architectures, deep learning is feasible to learn higher-level concepts based on local feature representations [8]. The main advantage is that potentially better features can be learned than hand-crafted features. On the other hand, one of the main goals of feature learning is to discover low-dimensional features that capture some structure underlying the high-dimensional input data. Autoencoders are one of the most common deep feature learning techniques. Their main goal is to obtain an informative representation of the input data for various implications. Feature learning and dimensionality reduction are the two main goals of autoencoders [9]. More specifically, autoencoders learn efficient low-dimensional representations of high-dimensional data while trying to reconstruct the original high-dimensional data from the low-dimensional representations. Among various feature engineering techniques, autoencoders can provide more discriminative features due to learning the semantic similarity and relationship between input features. An autoencoder consists of two parts: an encoder and a decoder. The encoder compresses the input into a low-dimensional latent representation, which encodes all the important information needed to represent the input. The decoder takes this latent representation and tries to reconstruct the input.

In this paper, we present DroidMalGNN, a deep learning technique that leverages autoencoders for deep graph representation learning to detect Android malware in an end-to-end manner. DroidMalGNN extracts an attributed function call graph (AFCG) for each Android application and combines autoencoders with GNNs in a unified way to benefit their power in building end-to-end deep learning models on graph-structured data. Using an autoencoder architecture for deep graph representation learning allows it to generate low-dimensional embedding vectors for each AFCG from the high-dimensional feature vectors of its nodes. This process is performed in a supervised manner, where two autoencoders are trained with benign and malicious AFCGs separately. Both these autoencoders finally play the same role in obtaining the final embedding vector of each AFCG. In the following, we list the main contributions of this paper:

- We combine autoencoders with GNNs in a unified way and present a novel architecture for deep graph representation learning in a supervised manner.
- We present DroidMalGNN, a novel Android malware detection technique that separately trains two autoencoders with graph convolutional layers, one with benign and the other with malicious Android applications. Both these autoencoders are finally used to obtain the final embedding vector of an AFCG. By doing so, DroidMalGNN can make a good difference in representing the embedding vectors of benign and malicious Android applications for classification.
- Through experiments on an Android malware dataset, we show that DroidMalGNN can achieve good detection performance in terms of different evaluation measures. Especially, DroidMalGNN achieves a detection rate higher than 94% and a false positive rate lower than 1.5%.

The rest of this paper is organized as follows. Section II briefly reviews related work, and Section III introduces some preliminaries and background information. Section IV introduces DroidMalGNN in detail. Section V reports our experimental results, and finally, Section VI summarizes and concludes the paper.

II. Related Work

In this section, we review the latest state-of-the-art techniques for Android malware detection.

He et al. [10] proposed an Android malware detection technique based on autoencoders. They trained an autoencoder to reduce the dimension of API feature vectors extracted and converted from APK files. The logistic regression model was then applied to the reduced feature vectors to learn and classify Android applications as benign or malicious. Then, Xu et al. [5] 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 a multilayer perceptron (MLP) classifier and trained it to differentiate between benign and malicious applications. Later, Li et al. [6] 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. Next, Wu et al. [7] 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 obtained 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. Then, Kabakus [11] proposed DroidMalwareDetector, a CNN-based Android malware detection technique that extracts permissions, intents, and sensitive API calls as features and encodes them as vectors using the word2vec model [12]. They trained a CNN model consisting of Conv1D layers using the extracted feature vectors to classify Android applications as benign or malicious. Later, Shen et al. [13] proposed an Android malware detection and classification technique based on network traffic analysis. They first converted raw network flows into grayscale images and then trained a deep malware detection model with a CNN and an LSTM layer that considers both spatial and temporal characteristics of network traffic data. They also utilized self-attention weights to focus on different features and improve detection performance.

III. Preliminaries

In this section, we provide a general description of the main concepts used in this paper.

A. Android

Android is an open-source mobile operating system composed of four main layers in a stacked architecture. The applications layer is the top layer of the Android architecture, containing pre-installed and third-party applications. The application framework layer provides high-level services such as Activity Manager, Resource Manager, and Notification Manager that developers are allowed to use in their applications. The native libraries layer contains a set of C/C++ and Java-based libraries. It also includes the Android runtime environment (ART), one of the most important parts of the Android platform that translates the application's bytecode into native instructions. The Linux kernel layer is the heart of the Android architecture, which is at the lowest level of the stack. It provides a level of abstraction between the device hardware and other Android architecture components and contains all the essential hardware drivers like camera, display, and Bluetooth drivers.

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, which is an archive in ZIP format. All resources of the Android application, including the Dalvik bytecode, static resources, assets, certificates, and native libraries, are stored in this archive.

B. Graph Neural Networks

Graph neural networks (GNNs) are one of the newest families of neural networks designed to operate on graph-structured data. They were first introduced by Gori et al. [14] in 2005 and further elaborated by Scarselli et al. [15] in 2009. Since their introduction, many different variants have been developed [16], [17]. Generally, GNNs take input feature vectors of nodes, edges, or graphs, representing their known attributes, and transform them into output feature vectors, i.e., embedding vectors. In more detail, an initial state is first assigned to each element of the input graph, and then these states are combined and updated by considering how the elements are connected in the graph. This is done using an iterative message-passing algorithm in which each node receives messages from all its neighbors and combines them using an aggregation function to obtain its new state. In this way, a GNN adapts its structure to the input graph and captures the complex dependencies of the graph through an iterative process of accumulating information across nodes. It allows for predicting properties of specific nodes, edges, or the graph as a whole and generalizing to unseen graphs [18].

Due to their powerful features, GNNs have shown that they could achieve good performance in several domains where data are inherently relational or can be structured as graphs [4]. One of these domains is malware detection, where executable programs and applications can be represented in various graph structures. Hence, recently GNNs have attracted more attention for malware detection [5]–[7].

IV. DroidMalGNN

In this section, we present DriodMalGNN, a novel deep learning technique that leverages autoencoders for deep graph representation learning to detect Android malware in an end-to-end manner. DriodMalGNN first extracts an attributed function call graph (AFCG) for each Android application in the preprocessing step. Then, in the deep graph representation learning step, it adopts an autoencoder architecture in combination with GNNs to learn the representations of AFCGs. Finally, in the malware detection step, DriodMalGNN generates the dual embedding vectors of AFCGs and classifies them as benign or malicious through a dense neural network. In the following, we describe the details of DriodMalGNN.

A. 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 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) are assigned the same one-hot encoded feature vector. For more efficiency, we

consider API functions commonly used by Android malware for malicious purposes as critical and then prune each AFCG to leave only nodes with at least one critical ancestor. We say that an ancestor is critical if it has at least one path to a critical API function. Some examples of critical API functions are `java.io.FileOutputStream.write()`, `java.io.Reader.read()`, `java.io.File.list()`, and `javax.crypto.Cipher.doFinal()`.

B. Deep Graph Representation Learning

An autoencoder is a deep neural network that is basically designed to encode the input into a compact representation and then decode it so that the reconstructed input is as similar to the original as possible. We adopt an autoencoder architecture for deep graph representation learning in a supervised manner. The main goal is to generate two low-dimensional embedding vectors for each AFCG from the high-dimensional one-hot encoded feature vectors of its nodes. For more efficiency, the process of generating these embedding vectors is performed in a supervised manner. For this purpose, we leverage two autoencoders trained with benign and malicious AFCGs separately. The first autoencoder, known as the benign autoencoder, takes a set of benign AFCGs as input. The second autoencoder, known as the malware autoencoder, receives a set of malicious AFCGs as input. Both benign and malware autoencoders learn the latent representation of their input AFCGs by reconstructing their feature matrix. The main goal is to encode each benign or malicious AFCG G 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 $\mathbf{X}$.

Due to the power of GNNs in building end-to-end deep learning models on graph-structured data, we strengthen DroidMalGNN by building the encoder and decoder parts of the benign and malware autoencoders using a GNN model. In more detail, each autoencoder consists of multiple graph convolutional layers. The lth graph convolutional layer can be 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 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 $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. In addition, $\mathbf{W}^{(l)} \in \mathbb{R}^{c_{l-1} \times c_l}$ 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.

We have chosen that the benign and malware autoencoders reconstruct the feature matrices of AFCGs instead of their adjacency matrices. The reason is that in the case of reconstructing the adjacency matrices, the decoder does not use the feature matrices at all and cannot be trained. However, this can sometimes lead to a reduction in the efficiency of graph reconstruction [19]. Meanwhile, in the process of reconstructing the feature matrices by each autoencoder, the decoder is also a trainable GNN model, which does the opposite of what the encoder does, using both the adjacency and feature matrices. Therefore, both structure and node features of AFCGs contribute to reconstructing the feature matrices.

In each autoencoder, the graph convolutional layers of the decoder are arranged in reverse order of the encoder. The decoder behaves the opposite of the encoder by taking the output of the encoder (i.e., the latent representation) as input and reconstructing the original feature matrix as closely as possible. Assuming that each 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 each 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 newly reconstructed feature matrix of nodes and $\|\cdot\|_F$ denotes the Frobenius norm. The second term is a weight decay regularizer with hyperparameter $\lambda > 0$ that penalizes larger weights.

C. Malware Detection

After completing the deep graph representation learning step, we have two autoencoders: benign autoencoder and malware autoencoder. These autoencoders are trained with benign and malicious AFCGs separately. In this step, we obtain a dual embedding vector for each given AFCG G, either benign or malicious. For this, we feed G into both benign and malware autoencoders and extract two latent representations, each containing an embedding vector for each node. We then apply a mean pooling operation to each latent representation to compute the average of all its node embedding vectors. Formally, let $\mathbf{Y}^+$ and $\mathbf{Y}^-$ be the latent representations of the benign and malware autoencoders, respectively. The output of the mean pooling operation, denoted by $\mathbf{y}^+$ and $\mathbf{y}^-$, is calculated as

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

$$\mathbf{y}^- = \frac{1}{n}\sum_{i=1}^{n} \mathbf{Y}_i^-, \tag{5}$$

where $\mathbf{Y}_i^+$ and $\mathbf{Y}_i^-$ denotes the ith row of $\mathbf{Y}^+$ and $\mathbf{Y}^-$, respectively. We refer to $\mathbf{y}^+$ and $\mathbf{y}^-$ as the positive and negative embedding vectors of G, respectively. By concatenating these two embedding vectors together, the dual embedding vector of the AFCG is obtained; formally, $\mathbf{y} = \mathbf{y}^+ \oplus \mathbf{y}^-$, where $\oplus$ is the concatenation operator.

Therefore, the dual embedding vector of each AFCG consists of two parts, the first part is obtained from the benign autoencoder, and the second part is obtained from the malware autoencoder. After generating the dual embedding vectors of all AFCGs, we utilize them to train an MLP classifier to classify given AFCGs as benign or malicious in a supervised manner. We finally use the whole trained model to detect Android malware. More precisely, for each given Android

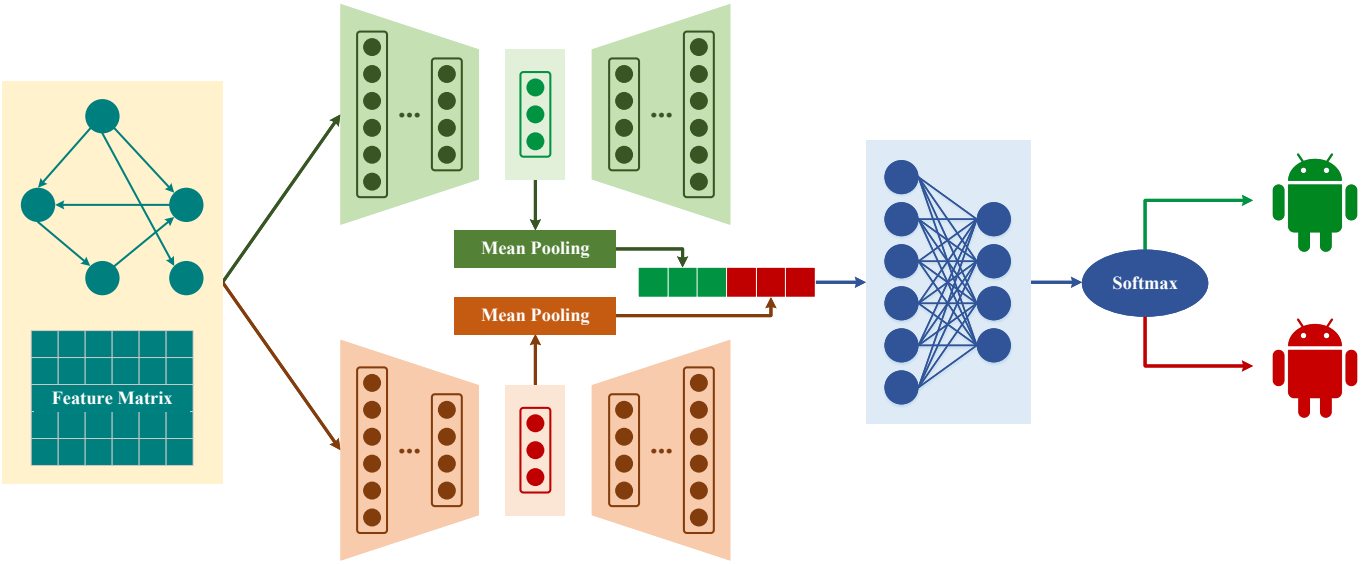


Fig. 1. An overview of the malware detection step of DriodMalGNN.

application, the dual embedding vector of its AFCG is obtained using both benign and malware autoencoders. The obtained dual embedding vector is then fed into the trained MLP classifier to decide whether the Android application is malicious or not. Fig. 1 shows an overview of the malware detection step of DriodMalGNN.

V. Experiments

In this section, we discuss the detection performance of DroidMalGNN for different parameter settings and compare it to a baseline case that uses an autoencoder for graph representation learning instead of two benign and malware autoencoders.

A. Dataset

In our experiments, we used the dataset provided by Chew et al. [20]. We selected 639 benign and malicious Android applications from this dataset. The malicious applications belonged to five different families of crypto-ransomware. We converted the APK file of each application to an FCG using Androguard [21]. We then generated an AFCG by associating a feature vector to 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.

B. Evaluation Measures

We evaluated the detection performance of DroidMalGNN using three common measures: detection rate (DR), false alarm rate (FAR), and accuracy (ACC). 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.

C. Experimental Results

In the basic experimental setup, the benign and malware autoencoders were trained with six graph convolutional layers (a three-layer encoder followed by a three-layer decoder). Moreover, the MLP classifier was trained with two hidden layers. We added a dropout layer of rate 0.1 and a ReLU activation function after each layer. The trained model was optimized by the AdamW optimizer [22] with a learning rate of 0.001 and a weight decay of 0.0005. 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.

We conducted 5-fold cross-validation to measure the detection performance of DroidMalGNN. For this purpose, we shuffled all 639 Android applications, selected 70% of them for training, and left 30% for testing. Therefore, the training and testing datasets contained 450 and 189 Android applications, respectively. Then, we divided the training dataset into five subsets of equal size. To provide better generalization, DroidMalGNN was trained and tested in five rounds. In each round, four subsets were considered as the training set and one subset as the validation set for selecting the hyperparameters. After completing the training process, the detection performance of the trained model was measured using the testing dataset. We repeated the above process five times to get more reliable results. Table I shows the detection performance of DroidMalGNN for these five experiments. We can see that DroidMalGNN consistently achieves a high detection rate and accuracy while keeping the false alarm rate low. For brevity, from now on, we only report the average results of the five experiments.

TABLE I
DETECTION PERFORMANCE OF DROIDMALGNN IN DIFFERENT EXPERIMENTS.

Experiment	DR	FAR	ACC
1	93.95	1.23	97.67
2	98.24	1.42	98.52
3	93.91	1.54	97.35
4	89.23	1.20	96.83
5	96.74	1.64	97.99

TABLE II
IMPACT OF THE NUMBER OF GRAPH CONVOLUTIONAL LAYERS OF THE BENIGN AND MALWARE AUTOENCODERS ON THE DETECTION PERFORMANCE OF DROIDMALGNN.

Number of Convolutional Layers	DR	FAR	ACC
2	79.22	1.07	94.54
6	94.42	1.41	97.67
10	92.40	2.74	96.17

TABLE III
IMPACT OF THE NUMBER OF HIDDEN LAYERS OF THE MLP CLASSIFIER ON THE DETECTION PERFORMANCE OF DROIDMALGNN.

Number of Hidden Layers	DR	FAR	ACC
1	88.92	1.18	96.59
2	94.42	1.41	97.67
3	93.58	1.41	97.48

Impact of the number of graph convolutional layers. In the first set of experiments, we evaluated the impact of the number of graph convolutional layers of the benign and malware autoencoders on the detection performance of DroidMalGNN. Table II shows the obtained results for two, six, and ten graph convolutional layers. We observe that the autoencoders with two graph convolutional layers (one layer for the encoder and one layer for the decoder) have a lower detection rate than those with six graph convolutional layers. We also observe that the autoencoders with ten graph convolutional layers (five layers for the encoder and five layers for the decoder) have a lower detection rate and a higher false alarm rate than those with six graph convolutional layers. Therefore, we conclude that the autoencoders with six graph convolutional layers can provide a better trade-off between detection performance and training time than the others. While keeping the false alarm rate at about 1%, this setting can achieve a detection rate of 94.42% and an accuracy of about 98%.

Impact of the number of hidden layers. In the second set of experiments, we evaluated the impact of the number of hidden layers of the MLP classifier on the detection performance of DroidMalGNN. Table III shows the obtained results for one, two, and three hidden layers. We observe that the MLP classifier with two hidden layers can achieve a better detection rate and provide a suitable trade-off among different evaluation measures than the others.

D. Comparison

We compared DroidMalGNN with a baseline model having only one autoencoder for deep graph representation learning instead of two benign and malware autoencoders. In other words, this baseline model (hereafter referred to as GAE-MLP) consists of an autoencoder with graph convolutional layers (GAE) followed by an MLP classifier. Fig. 2 shows this comparison for different evaluation measures. We observe that while the false alarm rate remains almost the same, the accuracy and detection rate in DroidMalGNN are improved compared to GAE-MLP. The reason is that DroidMalGNN leverages two autoencoders trained with benign and malicious AFCGs separately. As a result, these autoencoders can feed more discriminative embedding vectors into the dense neural network to distinguish between benign and malicious AFCGs.

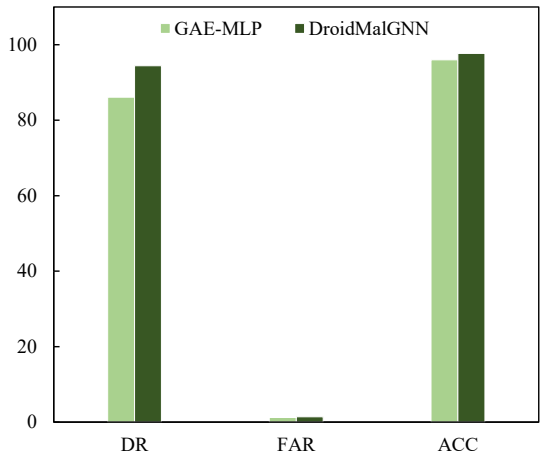
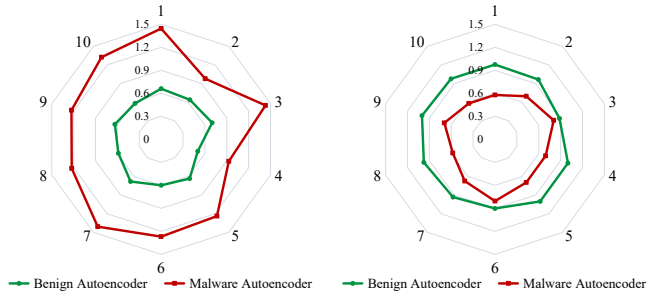


Fig. 2. Comparison of GAE-MLP and DroidMalGNN for different evaluation measures.

To better demonstrate the strength of the benign and malware autoencoders in generating discriminative embedding vectors, we compared the loss of these two autoencoders for benign and malicious AFCGs. For this purpose, we selected 10 benign and 10 malicious AFCGs from the testing dataset, fed them into each of the two trained autoencoders, and compared their loss. Figs. 3a and 3b compare the loss of the benign and malware autoencoders for benign and malicious AFCGs, respectively. Also, Figs. 4a and 4b compare the loss of benign and malicious AFCGs in the benign and malware autoencoders, respectively. From the figures, we observe that the benign/malware autoencoder produces less loss for benign/malicious AFCGs than for malicious/benign ones. Furthermore, benign/malicious AFCGs have less loss in the benign/malware autoencoder.

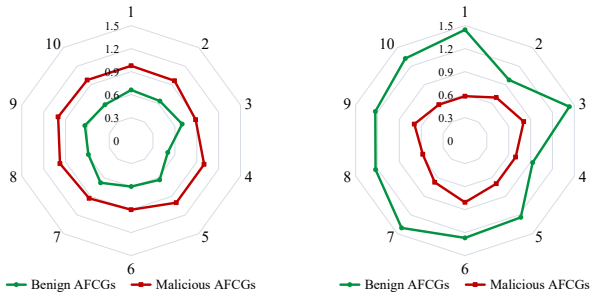
VI. CONCLUSION

In this paper, we have presented DroidMalGNN, an Android malware detection technique that leverages deep graph representation learning in a supervised manner. By doing so, we aim to generate graph representation vectors (i.e., embedding vectors) that accurately capture the structure of graphs and



(a) (b)

Fig. 3. Comparison of the loss of the benign and malware autoencoders for benign and malicious AFCGs. (a) Benign AFCGs. (b) Malicious AFCGs.



(a) (b)

Fig. 4. Comparison of the loss of benign and malicious AFCGs in the benign and malware autoencoders. (a) Benign autoencoder. (b) Malware autoencoder.

the content information of their nodes in a low-dimensional space. To build a deep graph representation learning model, we combine the autoencoder architecture with GNNs in a unified way, which allows us to leverage the power of GNNs on graph-structured data. We represent each Android application with an AFCG, where the feature vector of each node is obtained by applying one-hot encoding to its corresponding function name. We separately train a benign and a malware autoencoder. These autoencoders are then used to generate a dual embedding vector for each given AFCG. Each dual embedding vector consists of two parts, the first part is obtained from the benign autoencoder, and the second part is obtained from the malware autoencoder. A dense neural network follows the proposed deep graph representation learning architecture to learn a classifier for distinguishing benign and malicious AFCGs based on their dual embedding vectors. We have conducted empirical experiments to evaluate the detection performance of DroidMalGNN in terms of different evaluation measures. Experimental results have shown that DroidMalGNN can achieve good detection performance.

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] A. Tajoddin and M. Abadi, "RAMD: Registry-based anomaly malware detection using one-class ensemble classifiers," *Applied Intelligence*, vol. 49, no. 7, pp. 2641–2658, Jul. 2019.

[2] F. Laricchia, "Market share of mobile operating systems worldwide 2012-2022," https://www.statista.com/statistics/272698/, Aug. 2022.

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

[4] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[5] P. Xu, C. Eckert, and A. Zarras, "Detecting and categorizing Android malware with graph neural networks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Virtual Event, Republic of Korea, Mar. 2021, pp. 409–412.

[6] Q. Li, Q. Hu, Y. Qi, S. Qi, X. Liu, and P. Gao, "Semi-supervised two-phase familial analysis of Android malware with normalized graph embedding," *Knowledge-Based Systems*, vol. 218, p. 106802, Apr. 2021.

[7] Y. Wu, J. Shi, P. Wang, D. Zeng, and C. Sun, "DeepCatra: Learning flow- and graph-based behaviors for Android malware detection," *arXiv preprint arXiv:2201.12876*, pp. 1–12, Jul. 2022.

[8] W. L. Hamilton, *Graph Representation Learning*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer Nature Switzerland AG, Jan. 2022.

[9] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," *arXiv preprint arXiv:2003.05991*, pp. 1–22, Apr. 2021.

[10] N. He, T. Wang, P. Chen, H. Yan, and Z. Jin, "An Android malware detection method based on deep autoencoder," in *Proceedings of the 2018 Artificial Intelligence and Cloud Computing Conference*, Tokyo, Japan, Dec. 2018, pp. 88–93.

[11] A. T. Kabakus, "DroidMalwareDetector: A novel Android malware detection framework based on convolutional neural network," *Expert Systems with Applications*, vol. 206, p. 117833, Nov. 2022.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedings of the 2013 International Conference on Learning Representations*, Scottsdale, AZ, USA, May 2013, pp. 1–12.

[13] L. Shen, J. Feng, Z. Chen, Z. Sun, D. Liang, H. Li, and Y. Wang, "Self-attention based convolutional-LSTM for Android malware detection using network traffics grayscale image," *Applied Intelligence*, pp. 1–23, Apr. 2022.

[14] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, Montreal, QC, Canada, Jul. 2005, pp. 729–734.

[15] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the 5th International Conference on Learning Representations*, Toulon, France, Apr. 2017, pp. 1–14.

[17] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, LA, USA, May 2019, pp. 1–17.

[18] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Computing Surveys*, vol. 54, no. 9, pp. 1–38, Dec. 2022.

[19] S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang, "Adversarially regularized graph autoencoder for graph embedding," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, Jul. 2018, pp. 2609–2615.

[20] C. J.-W. Chew, V. Kumar, P. Patros, and R. Malik, "ESCAPADE: Encryption-type-ransomware: System call based pattern detection," in *Network and System Security*, M. Kutyłowski, J. Zhang, and C. Chen, Eds. Springer International Publishing, 2020, pp. 388–407.

[21] A. Desnos, "Androguard," https://github.com/androguard/androguard, 2022.

[22] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, LA, USA, May 2019, pp. 1–18.