International Journal of Computer Applications (0975 - 8887)
Volume 174 - No.29, April 2021

Behavioral Malware Detection using Deep Graph Convolutional Neural Networks

Angelo Schranko de Oliveira
PPGI - Universidade Nove de Julho
235/249 Vergueiro St.
São Paulo, Brazil

Renato José Sassi
PPGI - Universidade Nove de Julho
235/249 Vergueiro St.
São Paulo, Brazil

ABSTRACT

Malware behavioral graphs provide a rich source of information that can be leveraged for detection and classification tasks. In this paper, we propose a new behavioral malware detection method that extracts behavioral graphs from API call sequences and uses a Deep Graph Convolutional Neural Network (DGCNN), a state-of-the-art neural network architecture that can directly accept graphs of arbitrary structures, to learn a binary classification function able to distinguish between malware and goodware. In order to train and evaluate the models, we created a new public domain dataset of more than 40,000 API call sequences resulting from the execution of malware and goodware instances in a sandboxed environment. Experimental results show that our models achieve similar Area Under the ROC Curve (AUC-ROC), F1-Score, Precision, and Recall to Long-Short Term Memory (LSTM) networks, widely used as the base architecture for sequence learning in behavioral malware detection methods, thus indicating that the models can effectively learn to classify malicious and benign temporal patterns through convolution operations on graphs. To the best of our knowledge, this is the first paper that investigates the applicability of DGCNN to behavioral malware detection using API call sequences.

General Terms

Malware Detection, Deep Learning on graphs

Keywords

Computer Security, Deep Learning, Dynamic Analysis, Malware Detection

1. INTRODUCTION

According to a report published by AV-TEST [1], 9.74 million new malware specimens were released just in September of 2019, totaling 948 million known specimens in the wild. Dealing with the rapid increase of malware in number, complexity, and variability requires the research and development of new intelligent, automatic malware detection methods capable of scaling accordingly.

There are two main approaches to detecting malware; static malware analysis, and dynamic malware analysis [2]. On the one hand, static malware analysis can be conducted quickly by comparing a set of handcrafted features of the incoming file to previously observed malware features or signatures, which makes static analysis vulnerable to code obfuscation techniques employed by polymorphic and metamorphic malware [3], as well as to complete new specimens of malware or zero-days. Traditional signature-based malware detection methods are the cornerstone of the majority of the commercial endpoint protection systems since they are relatively fast and do not depend on any additional infrastructure to collect and analyze the data; however, they require expert knowledge to reverse engineer malware instances and produce the features that will be used for detection. Evidently, this approach does not scale as fast as malware production.

On the other hand, dynamic malware analysis or behavioral analysis is based on behavioral data such as API or system calls, which is harder to obfuscate [4]. In order to collect dynamic analysis data, it is often necessary to run the program in a sandbox environment [5]. A sandbox provides a controlled and isolated environment for the guest program to run while monitoring and tracking its activities. Once the data is collected and preprocessed, it can be used to feed behavioral detection algorithms [6].

Deep learning algorithms have shown unprecedented success in various domains such as image classification, natural language processing, and speech recognition [7]. The key advantage of deep learning is the capability of automatically learn hierarchical feature representations using a general-purpose learning procedure [7]. Deep learning algorithms are less dependent on good feature extractors that would require skill and domain expertise to perform feature engineering; therefore, they can easily take advantage of the increasing amount of data and computation available [7].

Following this trend, Deep Learning algorithms have also been applied to malware detection and classification tasks using static and dynamic analysis data exploiting its temporal [8, 9, 10], spatial [11, 12], or spatio-temporal [13, 14] structure.

In this work, we propose a new behavioral malware detection method that exploits yet another structure of the dynamic analysis data, the graph structure of the API call sequences. To accomplish this task, our method is based on a state-of-the-art Deep Learning architecture designed for graph classification; more specifically, the Deep Graph Convolutional Neural Network (DGCNN) [15]. Due to their capability of learning from non-Euclidean data such as graphs, Graph Neural Networks (GNNs) [16, 17] can be applied to problems in a vast range of domains from protein classification [18] to Materials science [19]. By defining a graph structure to represent the API call sequence of a program, we combine both the spatial and temporal information from its behavior. Then, we use the DGCNN to learn high-level representations that can be used by a

classifier to detect whether the program is malware or goodware. Experimental results show that the proposed method achieves similar AUC-ROC [20], F1-Score, Precision and Recall to specialized Deep Learning architectures for sequence learning such as LSTM networks [21], widely used as the base architecture for behavioral malware detection methods [22]. To the best of our knowledge, this is the first paper that investigates the applicability of DGCNN to behavioral malware detection using API call sequences.
The main contributions of our work are as follows.

(1) We propose a new behavioral malware detection method that uses graph convolution operations for sequence learning by leveraging the graph structure of the API call sequences.
(2) LSTM networks are the de-facto standard networks used for sequence learning problems, including behavioral malware detection and classification methods using API call sequences. In this work, we propose the use of an alternative state-of-the-art architecture, the DGCNN, to tackle the problem of behavioral malware detection using API call sequences and their associated behavioral graphs. Our results show that, although DGCNNs were not initially designed for sequence learning problems, our method achieves similar ROC-AUC, F1-score, Precision, and Recall to LSTM networks, thus indicating that DGCNNs are also applicable.
(3) We created a new public domain dataset of more than 40,000 API call sequences resulting from the execution of malware and goodware instances in a sandboxed environment.

The rest of the paper is organized as follows. Related work is reviewed in Section 2. A brief background on DGCNNs is introduced in Section 3. The proposed method is detailed in Section 4. Performance evaluation is described in Section 5. Results, discussion, limitations, and future work are presented in Section 6. Finally, conclusions are drawn in Section 7.

2. RELATED WORK

Classical Deep Learning algorithms such as Convolutional Neural Networks (CNNs) [23] and LSTM networks have been successfully applied to malware detection and classification problems using both static and dynamic analysis data [22]; however, Deep Learning on graphs has been mainly applied to data extracted employing static analysis methods.
Yan *et al.* [24] proposed a malware classification method (MAGIC) based on a modified version of the DGCNN to learn directly from attributed control flow graphs (ACFGs) extracted from disassembled binaries, in which each vertex summarizes code characteristics as numerical values.
Jiang *et al.* [25] introduced a malware detection approach using graph embedding to map the control flow graphs (CFGs) extracted from disassembled binaries to low-dimensional vectors as inputs for two stacked denoising autoencoders (SDAs) that are responsible for representation learning.
Zhu *et al.* [26] presented a method for Android malware detection that creates CFGs using data extracted from decompiled applications' source codes and information from their manifest files. Then, a Graph Convolutional Network (GCN) [27] was used to learn high-level representations that could be used in detection tasks.
Phan *et al.* [28] studied the effectiveness of DGCNNs in processing large-scale graphs with hundreds of thousands of nodes by conducting experiments on malware detection and software defect prediction.
Our work follows a similar approach to [24] and shares the same theoretical basis on applying DGCNNs for classification tasks [15]. However, we use the standpoint of dynamic analysis by extracting behavioral graphs from the API call sequences and using both the API call sequences and the behavioral graphs as inputs to a DGCNN. In addition, the standard LSTM network was chosen as a benchmark since it has been successfully applied as the base architecture for several behavioral malware detection and classification methods using API call sequences data [22].

3. BACKGROUND ON DEEP GRAPH CONVOLUTIONAL NEURAL NETWORKS

DGCNN is a state-of-the-art neural network architecture that can directly accept graphs of arbitrary structures to learn a graph classification function [15]. In other words, DGCNNs deal with the task of graph classification as opposed to node classification [27].
Let G be a directed graph of order $n \in \mathbb{N}$ and $A \in \mathbb{Z}^{n \times n}$ its associated adjacency matrix. Now, let us define the following: The augmented adjacency matrix of G, $\tilde{A} = A + I_n$, to ensure that the convolution operation takes into account the features of each node as well as its neighbours' features. The augmented diagonal degree matrix of G, $\tilde{D}_{i,i} = \sum_j \tilde{A}_{i,j}$ for row-wise normalization. The node feature matrix $X \in \mathbb{R}^{n \times c}$, $c \in \mathbb{N}$, where each row of X is a node "feature descriptor", and each column of X is a node "feature channel." The matrix of learning parameters $W \in \mathbb{R}^{c \times c'}$, where $c' \in \mathbb{N}$ is the number of output feature channels, with the non-linear activation function $f : \mathbb{R}^{n \times c'} \to \mathbb{R}^{n \times c'}$. Then, the graph convolution operation can be written as follows [15]:

$$Z = f(\tilde{D}^{-1}\tilde{A}XW) \tag{1}$$

The graph convolution operation defined by (1) aggregates local substructure information by considering the nodes' immediate neighborhoods. In order to extract multi-scale substructure features, (1) can be stacked to form a deep network, using the following recurrence relation [15]:

$$Z^{(t+1)} = f(\tilde{D}^{-1}\tilde{A}Z^{(t)}W^{(t)}) \tag{2}$$

where $Z^{(0)} = X$ and $W^{(t)} \in \mathbb{R}^{c_t \times c_{t+1}}$, $t \in \mathbb{N}$.
In summary, standard DGCNNs have four sequential steps [15]: 1) Graph convolutional layers generalize the convolution operation from Euclidean domains or grid-like structures such as image data to non-Euclidean domains such as graph data by generating node representations as the aggregation of their own feature descriptors and their neighbors' feature descriptors. 2) Unordered graph data from each convolutional layer are concatenated along their feature channels (or columns), resulting in matrix $Z \in \mathbb{R}^{n \times \sum_t c_t}$. 3) A SortPooling layer sorts the unordered graph data according to their feature descriptors or structural roles. This step guarantees that the nodes of different graphs will be placed in similar positions, according to their weighted feature descriptors. 4) The ordered graph data is flattened and passed to a standard 1-dimensional CNN layer followed by a fully connected layer to learn a classification function. For a more comprehensive review, please refer to [15].

4. PROPOSED METHOD

As illustrated in Fig. 1, our method has eight sequential steps from data gathering to detection. First of all, Portable Executable (PE) files (1) are fed to a Cuckoo Sandbox [29] environment (2), which in turn runs the PE files and generates raw JSON reports containing dynamic analysis data such as API call sequences, generated traffic and dropped files (3). Next, the API call sequences are extracted from the reports and post-processed in order to identify and convert

the API calls into ordinal categorical values (4). At this point, we have tracked the temporal behavioral information from the PE files and the ordered set of all possible API calls. Behavioral graphs are then generated based on both the API call sequences and the set of API calls (5), and both are passed to a graph convolutional layer to learn high-level representations of the spatial and temporal relations among the API calls (6). If multiple graph convolutional layers are stacked together to form a deep network, it is necessary to concatenate their results in order to consider multi-scale substructure features. Finally, the learned representations are passed to a fully connected layer (7), followed by a sigmoid layer (8) binary classification. In the next sections, a more in-depth description of the method is presented.

4.1 Data Collection and Post-Processing

We introduced a new public domain dataset of 42,797 malware API call sequences and 1,079 goodware API call sequences each [30]. Our motivation was twofold. On the one hand, we were motivated by the lack of public domain PE dynamic malware analysis dataset for training and evaluating our models. On the other hand, we were motivated by the desire to provide an open dataset that the research community could further utilize and extend.

Following [10], malware samples were collected from VirusShare [31], and goodware samples were collected from both portablepps.com [32] and a 32-bit Windows 7 Ultimate directory. Portableapps.com offers commonly used Windows applications such as text editors, spreadsheets, calculators, email clients, games, FTP clients, and so son, that have been specially packaged for portability. Based on [33], we assume that these goodware applications to be representative of software commonly present on the average user's computer. All the samples from VirusShare have at least one positive detection from Virustotal [34], and all the samples from portableapps.com are assumed to be goodware since the repository is actively developed and maintained. Also, notice that all the collected goodware samples can be executed in the sandbox without any installation procedure and either are standalone executables or depend only on local system Dynamic Link Libraries (DLLs).

In order to gather the API call sequences from each sample, we chose Cuckoo Sandbox, which is a largely used, open-source automated malware analysis system capable of monitoring processes behavior while running in an isolated environment. We set up a standard Cuckoo Sandbox environment with full internet access and without any hardening techniques such as anti-evasion or anti-VM. Each instance was executed only once, with the average execution and logging time of 5 minutes.

Once the data was collected, three additional post-processing steps were performed. 1) Similar to [13], it was considered the first 100 non-consecutive repeated API calls to avoid tracking loops. 2) Since in malware detection tasks, it is prominent to recognize malicious patterns as early as possible, the sequences were extracted from the parent process only. 3) We built the list of unique API calls, considering all the samples, and then converted each API call name into a unique integer identifier equal to the index of the API call name in the list.

As a result, 307 distinct API calls were identified. We produced a dataset where the first column contains the MD5 hash of the sample. The next 100 columns contain ordinal categorical values between 0 and 306, representing the API call sequence of the sample. The last column contains the label of the sample.

The total running time to collect the data was about 3000 hours, resulting in approximately 50,000 Cuckoo JSON report files and 1.5 TB of raw data. Our Cuckoo sandbox environment was based on an Intel Xeon D-1540, 8 cores, 16 threads, 2.6 GHz, 64 GB RAM, and 2 TB SSD running Ubuntu Server 16.04 Ubuntu Server 16.04 as the Cuckoo host and 8 32-bit Windows 7 Ultimate VirtualBox virtual machines running in parallel as Cuckoo analysis guests.

4.2 API Call Sequences and Behavioral Graphs Generation

On the one hand, API call sequences represent the most important part of the program behavior through time [13]. On the other hand, graph structures encode spatial relations, such as adjacency and connectivity, between API calls. Our method leverages both temporal and spatial information for malware detection. In order to accomplish that, it is necessary to extract the graph structure from the API call sequences to generate their associated behavioral graphs. A behavioral graph is a graph representation of a program run and, in particular, of its API call sequences [35]. Fig. 2 shows the behavioral graph of a Trojan horse malware. More formally, let us define a behavioral graph as a 2-tuple directed graph $G = (N, E)$, where $N \subseteq \mathbb{N}$ is the ordered set of nodes representing the ordered set of API calls, $E \subseteq N \times N$ is the set of edges, and each edge $e = (e_i, e_j) \in E$ corresponds to the temporal relation between the two consecutive API calls e_i and e_j such that the order of execution is e_i followed by e_j. Let $x = (x_0, x_1, ..., x_{L-1}), x_i \in N$ be an API call sequence of length $L \in \mathbb{N}$. Then, the adjacency matrix $A \in \{0,1\}^{|N|\times|N|}$ of the behavioral graph G associated to x is given by: [1]

$$A_{ij} = \begin{cases} 1 & \text{if } (x_i, x_j) \subseteq x \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The above definition can be implemented using Algorithm 1.

Algorithm 1: Seq2BGraph

input : API call sequence x of length $L \in \mathbb{N}$. Cardinality of N: $|N| \in \mathbb{N}$
output: Adjacency matrix A of the behavioral graph G associated to x
Let A be a new zero-initialized $|N| \times |N|$ matrix
for $i \leftarrow 0$ **to** $L - 2$ **do**
  $A[x_i, x_{i+1}] \leftarrow 1$
end

As an example, let $N = (0, 1, 2, 3)$ be the ordered set of API calls. Fig. 3 step I shows the behavioral graph G resulting from the adjacency matrix generated by (3) applied to the API call sequence $x = (0, 1, 2, 0, 2, 3)$.

4.3 Deep Graph Convolutional Neural Networks and Graph Convolutional Layers

In order to take advantage of the DGCNN architecture, let us define the node feature matrix $X \in \{0,1\}^{|N|\times L}$ of G as the result of one-hot encoding each x_i in the API call sequence x. As an example to show how the graph convolution operation acts on X, let us again consider the ordered set of API calls $N = (0, 1, 2, 3)$, the API call sequence $x = (0, 1, 2, 0, 2, 3)$ and the adjacency matrix A,

[1] The reader may forgive a little abuse of notation here. $(x_i, x_j) \subseteq x$ if and only if x_i, x_j are two consecutive elements in the sequence x

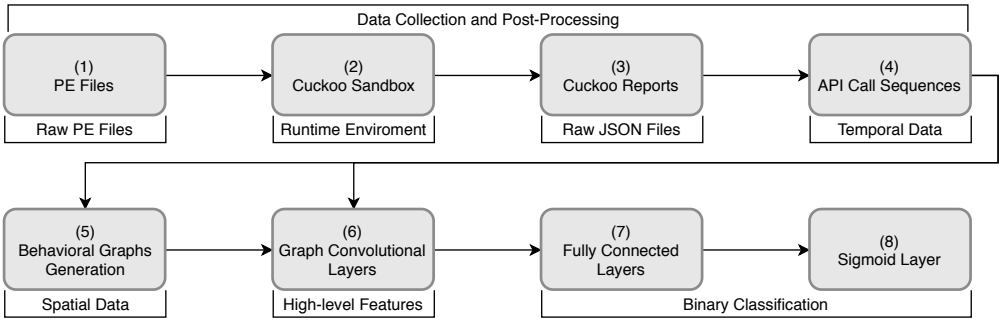

Fig. 1. High-level flow of the proposed method.

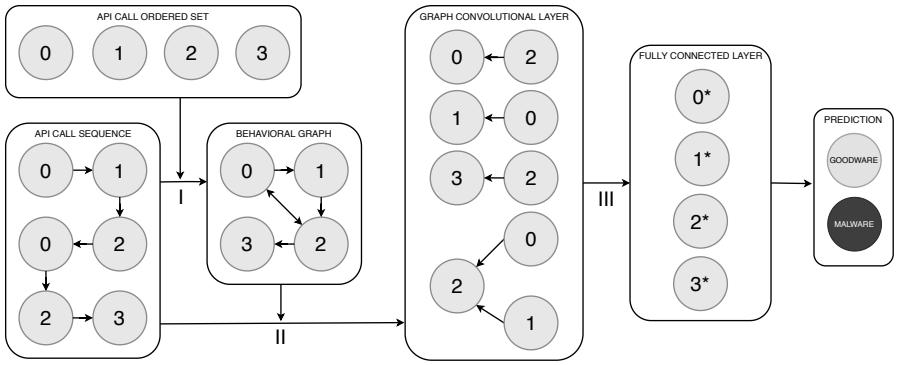

Fig. 3. Behavioral graphs generation and graph convolution operations.

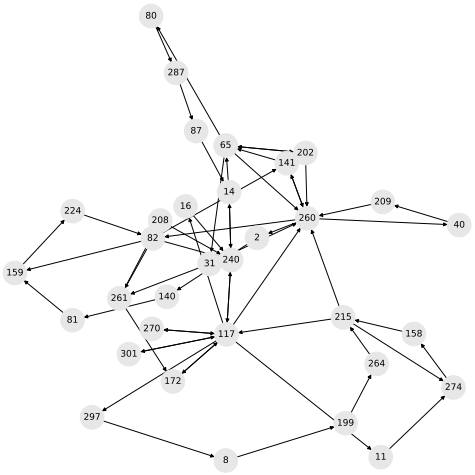

Fig. 2. Behavioral graph of the Trojan horse malware with MD5 hash ac65ce897a1f0dc273e8dc54fe3768ec.

generated by Algorithm 1, associated to the behavioral graph G of x, depicted in Fig. 3. For the sake of clarity, let us take the product AX in (4) and its visualization in step II of Fig. 3

$$AX = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 0 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \tag{4}$$

$$= \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 0 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

As we can see, the resulting product AX is a matrix representing the API call sequence and, for each API call, its immediate indegree neighbors. Also, notice that the rows of AX represent ordered nodes, and the columns of X represent the behavior of the program in time given by the API call sequence x. Moreover, since the nodes of G are already sorted by their natural order, our model does not require the SortPooling layer introduced in [15], thus reducing its execution time. Finally, the term $\tilde{D}^{-1}\tilde{A}X$ is multiplied by the weight matrix W, allowing the model to learn higher-level representations. For example, let us consider only the product AXW

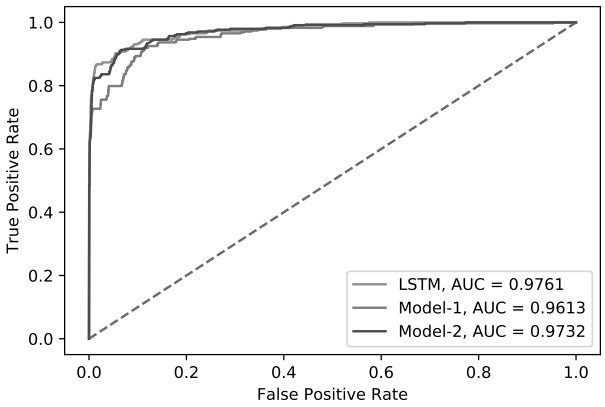

Fig. 4. ROC curves for the tuned models.

with $W \in \mathbb{R}^{6\times 2}$, then:

$$AXW = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 0 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \\ w_{51} & w_{52} \\ w_{61} & w_{62} \end{bmatrix} \tag{5}$$

$$= \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{bmatrix} w_{21} + w_{31} + w_{51} & w_{22} + w_{32} + w_{52} \\ w_{31} + w_{51} & w_{32} + w_{52} \\ w_{11} + w_{41} + w_{61} & w_{12} + w_{42} + w_{62} \\ 0 & 0 \end{bmatrix}$$

4.4 Fully Connected Layers and Sigmoid Layer

The last graph convolutional layer outputs a matrix $Z^{(T)} \in \mathbb{R}^{|N|\times c_T}$, $T \in \mathbb{N}$. Finally, in order to train a fully connected layer on it, first it is necessary to concatenate $Z^{(1)}, \ldots, Z^{(T)}$ along their columns and then reshape the result into a row vector $z \in \mathbb{R}^{|N|\sum_{t=1}^{T} c_t}$. As an example, let us consider (5). Notice that after reshaping, the high-level features will be grouped by the natural order of the nodes (6) and can be passed to a fully connected layer followed by a sigmoid layer for binary classification:

$$z = \begin{array}{c} \\ 0 \\ 0 \\ 1 \\ 1 \\ 2 \\ 2 \\ 3 \\ 3 \end{array} \begin{bmatrix} w_{21} + w_{31} + w_{51} \\ w_{22} + w_{32} + w_{52} \\ w_{31} + w_{51} \\ w_{32} + w_{52} \\ w_{11} + w_{41} + w_{61} \\ w_{12} + w_{42} + w_{62} \\ 0 \\ 0 \end{bmatrix} \tag{6}$$

Fig. 3, step III, depicts the high-level features grouped by the natural order of the nodes.

4.5 The Method

In summary, without considering the data collection and post-processing steps, our method can be implemented using Algorithm 2. According to the principles of Deep Learning [7], Algorithm 2 can be extended by stacking the graph convolutional layers or fully connected layers followed by the sigmoid layer for binary classification or a softmax layer multi-class classification. Furthermore, we included a Dropout [36] layer after each graph convolutional layer in order to prevent overfitting and used ReLU [37] as the activation function to perform non-linear transformations while preventing the vanishing gradient problem. Notice that our method does not include the SortPooling layer introduced in the original DGCNN architecture [15] since the nodes of G are already sorted by their natural order. Moreover, our method also does not include the 1-dimensional CNN layer preceding the fully connected layer since its addition did not show performance improvements.
DGCNN uses a constant amount of memory for model parameters, and its runtime complexity is proportional to the number of vertices and vertex degrees [28], in particular, our method runtime complexity is $\mathcal{O}(L\times|N|)$, since the maximum number of vertices is defined by the API call sequence length L and the maximum vertex degree is defined by the cardinality of the ordered set of API calls N.

Algorithm 2: The Model

input : API call sequence x. Cardinality of N: $|N|$
output: The probability of x was generated by a malware
$\tilde{A} \leftarrow Seq2BGraph(x, |N|) + I_N$
$\tilde{D} \leftarrow DiagonalDegree(\tilde{A})$
$X \leftarrow OneHotEncoding(x, |N|)$
$Z^{(1)} \leftarrow Dropout(ReLU(\tilde{D}^{-1}\tilde{A}XW^{(0)}))$
...
$Z^{(T)} \leftarrow Dropout(ReLU(\tilde{D}^{-1}\tilde{A}Z^{(T-1)}W^{(T-1)}))$
$Z \leftarrow Concatenate(Z^{(1)}, \ldots, Z^{(T)})$
$\hat{Y} \leftarrow \sigma(FullyConnected(Flatten(Z)))$

Algorithm 3: LSTM network

input : API call sequence x
output: The probability of x was generated by a malware
$X \leftarrow OneHotEncoding(x, |N|)$
$Y \leftarrow LSTM(X)$
$\hat{Y} \leftarrow \sigma(FullyConnected(Dropout(Y)))$

5. PERFORMANCE EVALUATION

The goal of our experiment was to establish a fair performance comparison between our models and LSTM networks on the same task. LSTM networks were chosen as the baseline architecture for performance comparison since they are the de-facto standard architecture used in sequence learning problems, including behavioral malware detection using API call sequences [8, 9, 10, 13, 14]. Taking that into account, let us consider the dataset introduced in 4.1 and the following models: Model-1, a 1-layer DGCNN, according to Algorithm 2. Model-2, a 2-layer DGCNN, according to Algorithm 2. LSTM, an LSTM network followed by a fully connected layer that receives the last LSTM network hidden state vector and

Table 1. Hyperparameter optimization search space. $W^{(0)}$ and $W^{(1)}$ are the sizes of the output channels for Model-1 or Model-2 for each layer. H is the size of the hidden layer for the LSTM network.

Hyperparameter / Model	Model-1	Model-2	LSTM
Dropout Rate	{0.4, 0.5, 0.6}	{0.4, 0.5, 0.6}	{0.4, 0.5, 0.6}
Mini-Batch Size	{32, 64, 128}	{32, 64, 128}	{32, 64, 128}
Number of Epochs	{10, 20, 30}	{10, 20, 30}	{10, 20, 30}
$W^{(0)}$ **or** H	{31, 137, 261, 402}	{17, 71, 126, 182}	{10, 40, 70, 100}
$W^{(1)}$	{}	{17, 71, 126, 182}	{}

Table 2. Evaluation metrics for the tuned models.

Metric / Model	Model-1	Model-2	LSTM
AUC-ROC	0.9613	0.9732	0.9761
F1-Score	0.7523	0.7613	0.7389
Precision	0.8039	0.7247	0.6466
Recall	0.7069	0.8017	0.8621

outputs the result to a sigmoid layer for binary classification, according to Algorithm 3. Three experiments were set up to perform model selection, training, and evaluation. In total, 648 models were defined, trained, and evaluated, resulting in three optimized models for malware detection using API call sequences. The total running time for the hyperparameter optimization was about 40 hours on an Intel Xeon D-1540, 8 cores, 16 threads, 2.6 GHz, 64 GB RAM, 2 TB SSD, and a GPU card NVIDIA RTX 2080 Ti. We used Jupyter Notebook, PyTorch , Pandas , scikit-learn, imbalanced-learn, skorch and Matplotlib for the implementations, which are available in [38].

5.1 Model Selection

First of all, a stratified train-test split was performed, where the size of the test set was 0.3 of the total, resulting in a training set of 731 goodware samples (class 1) and 29982 malware samples (class 0), and a test set of 348 goodware samples and 12815 malware samples. Next, to perform model selection (i.e., hyperparameters optimization), an exhaustive grid search was executed on the training set using stratified 5-fold cross-validation with AUC-ROC as the validation metric. In an exhaustive grid search, the model is trained and evaluated with all the hyperparameters combinations. The stratified k-fold cross-validation ensures that each training set split contains a similar proportion of positive and negative samples. Then, the model is trained with $k-1$ folds and its performance is evaluated using the fold that was left out of the training process. This process is repeated k times, each time considering a different set of folds. The average of the evaluation performances is an estimate of the model's performance on unseen data. The number of folds was set to 5, considering the computational resources needed to perform the grid search and the degree of bias and variance on the resulting metric [39]. In addition, AUC-ROC was chosen as the validation metric because it provides a summary of performance across all possible classification thresholds and can be used in balanced and imbalanced datasets [40].
In order to reduce the bias of the classifiers for the majority class, a random oversampling of the minority class was performed on the training set before each training session. Random oversampling was chosen based on the assumption presented in Section 4.1 that our goodware dataset is representative of software usually found on the average user's computer and because the training examples are composed by API call sequences represented by ordinal categorical values only; therefore, more advanced oversampling techniques such as Synthetic Minority Over-sampling Technique (SMOTE) [41] and its variations are not applicable.
Taking into account that we desired to compare conceptually different models, the following hyperparameters were considered for the grid search: Number of parameters per layer, dropout rate, mini-batch size, and the number of epochs. In order to define the sets of parameters per layer for the search, we first defined 4 possible values for the size of the LSTM hidden layer, then, considering Algorithm 2, it is possible to show that the total number of parameters for a 2-layer DGCNN with one fully connected layer followed by a sigmoid layer is given by:

$$P = L \times P_1 + P_1 \times P_2 + |N| \times (P_1 + P_2) + 1 \qquad (7)$$

where $P_i \in \mathbb{N}$ is the number of parameters of layer i. Equation (7) was used to build sets of numbers of parameters per layer, where the maximum number of parameters for each DGCNN model is less than the maximum number of parameters of the LSTM network, thus ensuring that all models are limited to a maximum complexity regarding the number of parameters while having the same degree of freedom for tuning.
Finally, Binary Cross-Entropy function was set as the loss function, and Adam optimizer [42] was used on a learning rate of 0.001, beta1 of 0.9, and beta2 of 0.999. Table 1 summarizes the hyperparameters optimization search space. Once the best sets of hyperparameters were found, the models were trained on the entire training set using those hyperparameters and tested against the test set.

6. RESULTS, DISCUSSION, LIMITATIONS, AND FUTURE WORK

Our results show that our method is indeed applicable to the problem of behavioral malware detection using API call sequences, and its performance is similar to LSTM networks, which are widely used as the base architecture for behavioral malware detection methods, thus indicating that DGCNNs can also be useful as the base architecture for more complex and deeper models for the same purpose. Fig. 4 depicts the ROC curves of Model-1, Model-2, and the LSTM network trained and evaluated using the tuned hyperparameters presented in Table 3.
As we can see in Table 2, our models achieve the highest F1-score and precision. Taking into account that the minority class (goodware) is usually defined as the positive class, a particularly important performance metric when evaluating malware detectors is the precision, since a high precision implies a low number of false positives, which can be interpreted as a high malware detection rate.

Table 3. Hyperparameters resulting from the model selection process.

Model	Best Configuration	# Parameters
Model-1	(0.6, 128, 20, 137)	55,760
Model-2	(0.6, 128, 10, 71, 17)	35,324
LSTM	(0.5, 64, 20, 10)	12,771

High recall implies a low number of false negatives, which is less critical but is desired for malware detectors. Ideally, both precision and recall should be high, implying a high F1-score.

In general, our models achieved similar performances to LSTM networks on the proposed task. As Table 2 shows, Model-1 and Model-2 dropout rates are the highest as opposed to the number of parameters. In fact, our models overfitted the training set just after ten epochs on average, indicating that additional dropout layers or L2 regularization, as well as the addition of more examples, could further improve their performance. We plan to investigate deeper architectures in future work.

The addition of one graph convolutional layer did not greatly improve the models' performance. Further improvement could be possible by introducing a larger dataset containing more goodware samples to reduce the imbalance instead of random oversampling the minority class.

Although in this work, it was considered only the task of malware detection or binary classification, DGCNNs can also be applied to the problem of multiclass classification [15], suggesting that our method can also be extended to multiclass behavioral malware classification. Future work will investigate this possibility. In addition, notice that our work only took into account one kind of execution trace, the API call sequences. In future work, we plan to augment our datasets to include other dynamic traces such as the API call parameters, return values, and execution status.

7. CONCLUSION

In this paper, we propose a new behavioral malware detection method based on DGCNNs to learn directly from API call sequences. In order to train, evaluate, and test the models, we introduced a new public domain dynamic analysis dataset of more than 40k API call sequences of malware and goodware. By extracting behavioral graphs from the API calls sequences and using both as inputs for a simplified version of the DGCNN, our method achieves similar performance to LSTM networks, which are largely applied as the base architecture for malware detection and classification methods using dynamic analysis data, including API call sequences, thus indicating that our method can also be useful as the base architecture for more complex and deeper models for the same purpose. Even though DGCNNs are memory-less networks, as opposed to LSTM networks, our results show that the graph structure of the API call sequences plays an essential role in the problem of detecting whether a program is malware. Future work will explore deeper architectures as well as the problem of multiclass malware classification using API call sequences and their associated behavioral graphs.

8. ACKNOWLEDGMENTS

We would like to thank Universidade Nove de Julho and the Coordination for the Improvement of Higher Education Personnel (CAPES) for supporting this research.

9. REFERENCES

[1] AV-TEST. Malware statistics & trends report, 2019.

[2] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

[3] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[4] Gérard Wagener, Alexandre Dulaunoy, et al. Malware behaviour analysis. *Journal in computer virology*, 4(4):279–287, 2008.

[5] Jayant Shukla. Application sandbox to detect, remove, and prevent malware, January 17 2008. US Patent App. 11/769,297.

[6] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3):251–266, 2008.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[8] Ben Athiwaratkun and Jack W Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486. IEEE, 2017.

[9] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE, 2015.

[10] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 77:578–594, 2018.

[11] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[12] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[14] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582. IEEE, 2016.

[15] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[16] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[17] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[18] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

[19] Tian Xie and Jeffrey C Grossman. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Physical review letters*, 120(14):145301, 2018.

[20] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[22] Daniel S Berman, Anna L Buczak, Jeffrey S Chavis, and Cherita L Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4):122, 2019.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–63. IEEE, 2019.

[25] Haodi Jiang, Turki Turki, and Jason TL Wang. Dlgraph: Malware detection using deep learning and graph embedding. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1029–1033. IEEE, 2018.

[26] Rui Zhu, Chenglin Li, Di Niu, Hongwen Zhang, and Husam Kinawi. Android malware detection using large-scale network representation learning. *arXiv preprint arXiv:1806.04847*, 2018.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[28] Anh Viet Phan, Minh Le Nguyen, Yen Lam Hoang Nguyen, and Lam Thu Bui. Dgcnn: A convolutional neural network over large-scale labeled graphs. *Neural Networks*, 108:533–543, 2018.

[29] cuckoosandbox.org. Automated malware analysis, 2019.

[30] Angelo Oliveira. Malware analysis datasets: Api call sequences, 2019.

[31] VirusShare. Virusshare, 2019.

[32] portableapps.com. portableapps.com, 2019.

[33] Domagoj Babić, Daniel Reynaud, and Dawn Song. Malware analysis with tree automata inference. In *International Conference on Computer Aided Verification*, pages 116–131. Springer, 2011.

[34] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www.virustotal.com/en*, 2012.

[35] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[37] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[38] Angelo Oliveira. Behavioral malware detection using deep graph convolutional neural networks. `https://github.com/ang3loliveira/behavioral_malware_detection_dgcnn_v2`, 2019.

[39] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[40] Qiong Gu, Li Zhu, and Zhihua Cai. Evaluation measures of the classification performance of imbalanced data sets. In *International symposium on intelligence computation and applications*, pages 461–471. Springer, 2009.

[41] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority oversampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[42] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.