Soft Computing (2020) 24:1027–1043
https://doi.org/10.1007/s00500-019-03940-5

METHODOLOGIES AND APPLICATION

Deep learning for effective Android malware detection using API call graph embeddings

Abdurrahman Pektaş[1] · **Tankut Acarman**[1]

Published online: 23 March 2019

Abstract
High penetration of Android applications along with their malicious variants requires efficient and effective malware detection methods to build mobile platform security. API call sequence derived from API call graph structure can be used to model application behavior accurately. Behaviors are extracted by following the API call graph, its branching, and order of calls. But identification of similarities in graphs and graph matching algorithms for classification is slow, complicated to be adopted to a new domain, and their results may be inaccurate. In this study, the authors use the API call graph as a graph representation of all possible execution paths that a malware can track during its runtime. The embedding of API call graphs transformed into a low dimension numeric vector feature set is introduced to the deep neural network. Then, similarity detection for each binary function is trained and tested effectively. This study is also focused on maximizing the performance of the network by evaluating different embedding algorithms and tuning various network configuration parameters to assure the best combination of the hyper-parameters and to reach at the highest statistical metric value. Experimental results show that the presented malware classification is reached at 98.86% level in accuracy, 98.65% in F-measure, 98.47% in recall and 98.84% in precision, respectively.

Keywords Android malware · Deep learning · Graph embedding · Hyper-parameter tuning · API call graph

1 Introduction

1.1 Overview of Android market and malware threats

Penetration of mobile devices and applications with Internet access makes easier our daily life tasks such as online banking, shopping from e-stores, exchanging e-mails at the risk of sharing safety and liability critical information. A total of 344.3 million smartphones worldwide in the first quarter of 2017 were produced, and Android has captured a record 85% of the global market. Android is an open-source operating system, which attracts potential buyers but also malware writers to execute their malicious intents. Android applications (apps) can be self-signed, which makes difficult to track back to their developers, and furthermore, apps can be repackaged by simply adding or modifying a payload that increases the number of variants (Li et al. 2017; Tian et al. 2017). Concerning security threats, 18.4 million mobile malware variants were detected with an increase of 105% in 2016, which was largely varied from the previous year's increase of 152% (https://www.symantec.com/content-/dam/symantec/docs/reports/istr-21-2016-en.pdf). Four new mobile families and clusters of 61 distinct new mobile threats were discovered in 2016 versus the identification of 18 new families and 75 clusters in 2015. The high number of malware samples versus change in growth rate and lower number of family can be interpreted such that malware writers have been focused on refining and modifying existing malware families in order to mitigate detection and execute malicious intents instead of deploying new and unique malware threats. In consequence, Android malware needs to be detected while using a large set of apps, and during training, detection system needs to be configured at its optimal parameter values and be reached at the highest level in prediction.

Communicated by V. Loia.

✉ Abdurrahman Pektaş
apektas@yandex.com

[1] Computer Engineering Department, Galatasaray University, 34349 Ortaköy, Istanbul, Turkey

1.2 Overview of static and dynamic features toward learning of malware behavior

Scalable and automated detection systems are necessary to remove the human expert analysis and workload while fighting with a huge number of malware variants. Machine learning algorithms have been applied to detect Android malicious apps by using features extracted from the given dataset (Tam et al. 2017). Features are extracted by either static or dynamic analysis. Static features such as Application Programming Interface (API) calls, operational code (opcode) sequences, permission requests, control flows or data flows are extracted from the static source code (Pektas and Acarman 2017). Relationships between API calls are discovered by applying API call graphs. These features heavily depend on data distribution, and a large dataset enables comprehensive learning while promising reliable detection performance. Dynamic features are extracted by execution of a given Android app, and activities related to network, file system access and interaction with Android are monitored via its log files (Pektaş and Acarman 2014). Instruction call graph has been used to illustrate the activities and build a behavioral model of an Android app (Mariconti et al. 2016; Shen et al. 2017). In Fan et al. (2017), sensitive graphs are generated to present call relations among the functions captured from Android app files. In Anderson et al. (2011), dynamic analysis of instruction traces is used toward the creation of graphs representing Markov chains whose transition probabilities are extracted from a given dataset. Deep neural networks (DNNs) are emerging learning models, which are trained to approximate nonlinear functions between the inputs and the outputs, for our study, identification of malware or benign apps. A large variety of tuning options ranging from the configuration to the number of neurons, activation functions and their parameters subject to different objectives can be tuned and statistical metric values can be maximized.

1.3 Contributions of this study

The proposed malware detection method uses pseudo-dynamic analysis of Android apps and constructs API call graph for each execution path. Then, graph embedding is applied to convert call graphs into a low-dimensional feature vector capturing the structural information of the graph. A DNN is introduced to detect and extract hidden structural similarities from the graph embedding matrix via its convolution layers. A large dataset of Android apps is used to extract API call graph, and binary code similarity detection is assured by a dense layer in the network to determine whether it is a malware or benign by combining all complex local features discovered by convolutional layers in an effective and efficient way. Parallel processing supported by GPU technology is implemented to train and test learning models. Particularly, the DNN is evaluated and tested as a promising solution to reveal hidden knowledge from a large set of apps through extracted call graphs.

Contributions of this study can be summarized as follows:

- All execution paths in terms of the invoked APIs are captured and API call graph for each execution path is constructed. Pseudo-dynamic analysis is applied, i.e., apps are not executed instead all execution paths are analyzed via API call sequences.
- Graph embedding methods are evaluated to reduce the large dimension of call graphs into a low-dimensional feature vector and to capture the structural information of the graph extracted from a large set of apps in an efficient way.
- The deep learning architecture parameters are tuned and the Tree-structured Parzen Estimator is applied to seek the optimum parameters in the parameter hyper-plane.
- Different architectures are tested and compared with extensive experiments. The highest level in statistical metric values at detecting malicious apps is assured.

This paper is organized as follows: In Sect. 2, the related work is presented. In Sect. 3, deep learning is briefly presented and its main advantages toward Android malware detection are enumerated. In Sect. 4, the proposed methodology is elaborated. The experimental study, dataset and evaluation results are presented in Sect. 5. Finally, some conclusions are given.

2 Related work

2.1 Related work on call graphs toward identification of malware

Call graphs are derived from the data collected by tracking instruction traces and system call invocations of the given Android apps. These data are represented in a graph kernel to detect structural similarities. A graph-based representation is studied by augmenting a feature vector-based representation extracted by system call traces with a graph's vertex to model a process in an Android app (Xu et al. 2016). Call graph representations using the histogram, n-gram, and Markov chain system are introduced; the shortest path graph kernel algorithm (Borgwardt and Kriegel 2005) is applied to discover the similarities between each pair of graphs. In Anderson et al. (2011), the dynamic trace data is converted into a Markov chain represented by a weighted, directed graph whose vertex set is constituted by 160 instructions. To detect similarities between each pair of graphs, a Gaussian kernel and a spectral kernel is used to project the nonlin-

ear dynamic trace's data into a higher dimensional space to apply a linear classification equivalent to a nonlinear classification in the original data space. The combination of both kernels assures 96.41% subject to a dataset of 1615 malicious apps and 615 benign apps. The behaviors of an Android app are considered as the interactions among four type of components (activities, services, broadcast receivers, and content observers); an edge is created when one component starts the lifecycle of the other component (Yang et al. 2014). The created sequence of threat modalities' patterns is sought in other unknown apps to determine whether malicious or benign. In Kinable and Kostakis (2011), call graphs are extracted by static analysis of the executable file, vertices are constituted by importing external functions from the Address Import Table, and a statically linked library functions are recognized by a disassembler IDA Pro; then, edges between vertices are added. Finding minimum graph edit distance is applied to match two graphs, and k-means clustering is applied to detect a malicious app. From a large set of 1050 samples, 253 samples cannot be classified where total cluster number is computed as 50. In Wüchner et al. (2015), data communication flows between processes, sockets, files and system registers are presented by data flow graphs. Then, using quantitative data flows and their properties, malware detection heuristics were defined based on malware behavior databases. In Hashemi et al. (2017), extraction of opcodes from disassembled executable and connecting each individual opcode with other possible opcode nodes via probabilistic edge weights is presented. Then, Power Iteration to embed the given graph into a vector is proposed to transform the nonlinear data space into a linear combination of eigenvectors, which is well suited for training machine learning algorithms. A balanced dataset of 22,000 app samples and its subset of 2000 samples is evaluated; the Adaboost classifier is reached at 96.09% in accuracy and 95.98% in F-Measure. In Gascon et al. (2013), API call graphs are extracted to find similarities between samples, but graph comparison to identify similarities is a non-trivial and NP problem (Zeng et al. 2009), graph nodes are labeled according to Dalvik's 15 distinct categories of instructions. Then, for each node, neighborhood hash over all directed graphs is computed to identify the intersection of nodes labeled with the same hash. Eighteen Android malware families are classified with a level in accuracy higher than 90% and three families are identified with lower accuracy. In Xu et al. (2017), a neural network is used to convert a graph into an embedding, then a combination of networks is trained to assure whether two similar functions are close to each other or not. The number of training epochs, embedding depth, embedding size, control flow graph attributes and the number of iterations is tuned to improve the performance of similarity detection.

2.2 Related work on deep learning applied to detect Android malware

Considering deep learning algorithms, in Nix and Zhang (2017), a convolution neural network (CNN) is built and evaluated for API call-based Android app classification. Long Short-Term Memory (LSTM) is integrated to extract knowledge from sequences. CNN results are compared with respect to n-gram SVM and naive Bayes algorithm. In McLaughlin et al. (2017), a deep CNN is built with the raw opcode sequences. Recently, particular research has been focussed on tuning the network parameters, in Yuan et al. (2016), Hou et al. (2016); different network architectures are tested while tuning the parameters to reach a higher level in terms of detection accuracy. In Nauman et al. (2017), the hyperparameters of CNNs are tuned while using a dropout of 0.2 at each convolution layer to reduce overfitting, which leads to the conclusion that the largest network gives the highest statistical metric values. In Martinelli et al. (2017), deep learning classifiers are experimentally evaluated, hyperparameters are presented. In Xiao et al. (2017), 129 different system calls are extracted from the dataset and the longest sequence is about 200,000. Lengths of system call sequences are taken between 50 and 50,000; the highest level in accuracy is reached when the length is 100; and the hidden layer number is chosen 4. In Xiao et al. (2017), leveraging the knowledge about system call's transition probabilities' difference between malware and benign apps, the Markov chains are used to extract system call subsequences; the statistically derived transition matrices are given to artificial neural network (ANN) in a vector input. The ANN design parameters, which are the number of hidden layers and nodes in each layer, are tuned by simply choosing three and four layers and varying the number of nodes with an increment of 10 in the interval of 10 and 100. In Dimjašević et al. (2016), the effect of encoding of system calls into features and the relative sizes of benign and malicious datasets are experimentally investigated, and dynamic Android malware detection methods using tracking system calls are evaluated. In Rhode et al. (2017), ransomware detection is presented by using recurrent neural networks (RNNs) that capture time series data and sequential changes. The hyper-parameters are changed subject to random search. In Martín et al. (2017), a genetic algorithm is introduced to train the parameters of DNNs where a particular network architecture is constituted by each individual subject to both accuracy and network complexity constraints. In Gharib and Ghorbani (2017), three designs of deep auto-encoder with different configurations of hidden layers are evaluated to reduce and learn new features toward ransomware detection on a real-time basis. The configuration feeding both high-level and low-level features into a deep auto-encoder generates more accurate results. In Yousefi-Azar et al. (2017), an unsupervised automatic feature

learning model is presented by using auto-encoders, which accept the feature vector as an input and extract a code vector that captures the semantic similarity between the feature vectors. During the evaluation phase, a grid search is applied while tuning the parameters of k-NN from 1 to 20, and the C and gamma parameters of SVM from 10^{-4} to 10^{4}.

3 The proposed method

The proposed Android malware detection consists of six consecutive main stages. The first stage is pseudo-dynamic analysis of Android apps, and API call graph of the samples under inspection is extracted. Particularly, all execution paths are captured in terms of the invoked APIs and API call graph for each execution path is built. The API call number of each Android app sample is checked against a threshold whether to be processed by graph embedding. If the app sample generates greater than or equal to a given threshold, it is selected to generate graph embedding; otherwise, it is discarded. In this study, the threshold for the number of generated API call is chosen 60. At the second stage, graph embeddings based on the constructed graph is calculated. At the third stage, since machine learning and deep learning methods depend on numerical values, the graph embedding features are preprocessed to be interpreted numerically. The embedding vector is normalized, and each value in the vector is scaled between 0 and 255. Additionally, because the graph embedding dimension is very high, the embedding vector is rescaled into a smaller form to work effectively at the fourth stage. At the fifth stage, the normalized and rescaled feature vectors (i.e., embedding vector) are forwarded into the DNN to build the classification model. At the sixth stage, hyper-parameter tuning is applied to search the best performing network configuration. And finally, the classification performance of the proposed approach is evaluated. The flowchart about the proposed method is plotted in Fig. 1. The rest of this section will cover the basic knowledge about Android OS and application structure, graph embedding, extracting API call graph of the Android apps and the architecture of deep learning.

3.1 Background in Android OS and application

Android apps are generally coded in Java language and run in Dalvik Java Virtual Machine. Starting with Android version 5.0, Google replaced the Dalvik Virtual Machine with another app runtime environment called Android Runtime (ART). In 2017, Google announced that the Groovy has been decided to be the main development language in their future releases. Indeed, the Groovy language is very similar to Java language and the output of the compiled app is Java virtual machine executable (exe) file. The underlying runtime structure remains the same. Android apps are packed and distributed with an archive file called Android package (APK). APK archive file includes Dalvik bytecodes of the app, all resources such as image, data files and manifest.xml file, in which app permissions are defined.

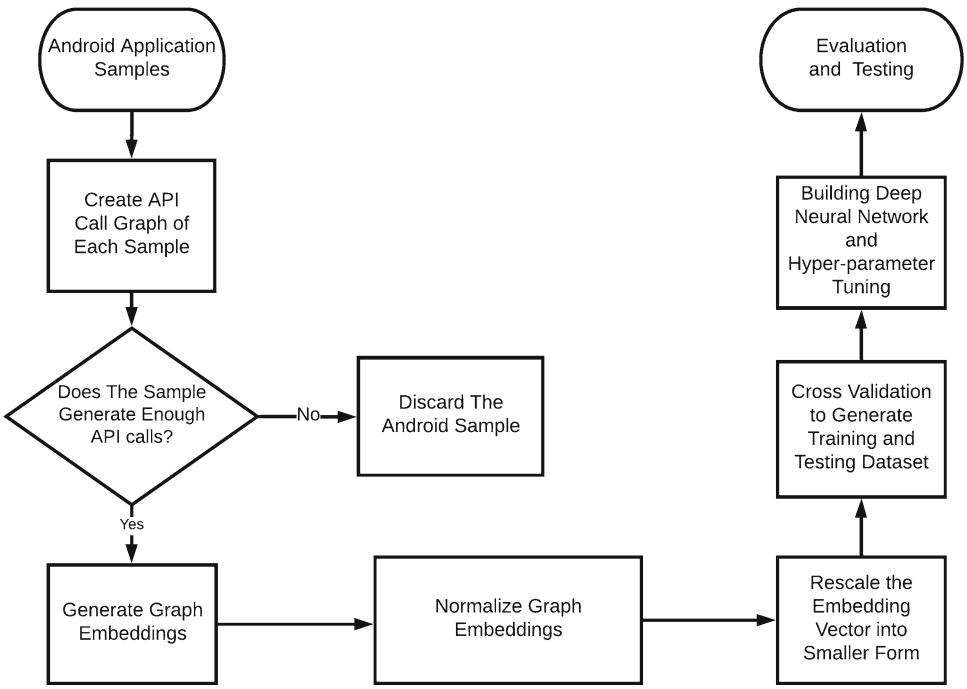

Fig. 1 The flowchart about the proposed method

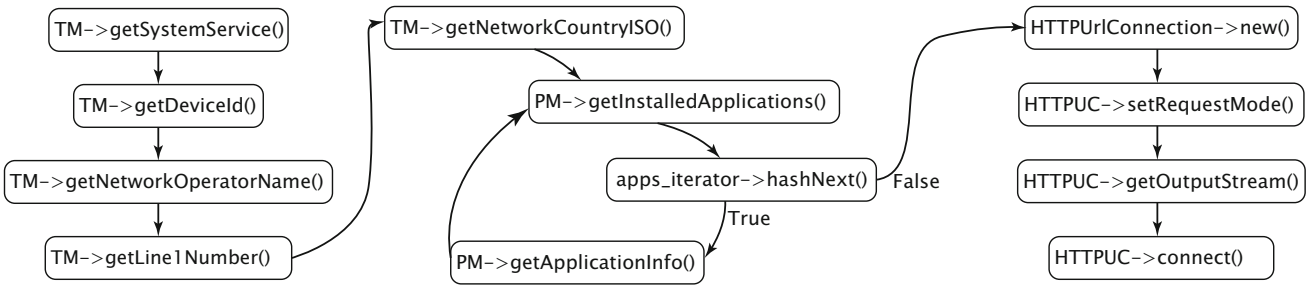

Fig. 2 Call graph of the sample malicious code given in Fig. 3

The Android app includes four essential objects to interact with the OS and other apps. While an app is running, these major objects follow different execution paths and run in harmony with each other. These objects are given as follows:

- Activity: It provides users to interact with a phone screen and any other user interface.
- Service: It enables to process background jobs without user interaction.
- Broadcast Receiver: It enables communication of different apps with each other.
- Content Provider: It enables accessing and altering data such as text messages, contact list, images, etc., on the Android platform.

3.2 Creating API call graph for Android applications

Examining API call sequences can help effectively reveal intentions of an app. For instance, when an app wants to obtain the device ID, it has to leverage the relevant API call or series of the API calls of the Android platform. Control Flow Graph (CFG) is a graph representation of all possible execution paths that are called during its runtime. In the Android platform, each method of Java class can be represented as a CFG. And overall, the entire app can be presented by the combination of its methods' CFGs. Formally, a CFG is a directed graph $G = (N, E)$, where N is a finite set of nodes representing Android-documented API calls, and E represents a finite set of edges that create links between successive instructions, (see also Ryder 1979). An edge (n_1, n_2) states that API call denoted by n_1 is followed by call n_2 on one of the execution path (see for instance, Fig. 2).

To build the control graph of an entire application, first of all, the entry point or points need to be defined. Android apps are different from former Java apps, that have one entry point called *main* method. However, Android apps can have multiple entry points. Entry point acts as a gateway to different types of events such as launching a service, touching the screen, changing the volume of the phone, etc., invoked by the app simultaneously by their event handlers, i.e., callback mechanism. Namely, Android OS dispatches an event handler to carry out the events while apps are running. Therefore, these event handlers need to be taken into consideration as entry points to successfully extract and build a CFG. Indeed, the entry point of an app is the main function invoked by the Android platform based on occurring predefined events initiated by the user or the Android OS. These three types of entry point for Android apps are activity, broadcast receiver, and services.

When a set of main entry points provided by the Android platform is extracted, the API call graph is built based on the starting points of the app. If there is a call to the other method, a link from the caller method to the other method is created in order to include all possible API calls of the Android app. All API calls reachable from the start points are drawn according to the call order. Not only Android core API is analyzed, but also API calls of the known libraries such as Apache are analyzed. However, the authors are interested in only the basic building block of Android apps and not interested in the user-defined methods to execute a task. Since these methods' call differ subject to different apps, they might lead to misleading identification results. To construct Android call graph, FlowDroid framework [1] (Arzt et al. 2014) is used.

As an illustrative example, the API call graph of the Android trojan is built in Fig. 2. It belongs to a malicious banking app whose pseudo-code is given in Fig. 3. For the sake of simplicity, relevant lines of the code showing malicious actions are given. First, the app gathers device-specific information including IMEI number, operator name, the phone number through TelephoneManager object, which gives access to the telephony services on the phone. Then, this app checks the installed banking apps and then uploads all gathered information to the Command and Control (C&C) server (i.e., http://r0n4ld4.gdn). This malicious app ran in an infinite loop and slept for 3 seconds at the end of each loop. The API call graph provides sufficient information about the program functionality by following events, intents, and transitions between edges as plotted in Fig. 2.

[1] https://github.com/secure-software-engineering/FlowDroid.

```
public void action(){
  while (true){
    TimeUnit.SECONDS.sleep(3L);
    // Collect phone specific information
    TelephonyManager localTelephonyManager = (TelephonyManager)getSystemService("phone");
    String IMEI = localTelephonyManager.getDeviceId();
    String SIMOperator = localTelephonyManager.getNetworkOperatorName()
    String SIMNumber = localTelephonyManager.getLine1Number();
    String phoneCountry = localTelephonyManager.getNetworkCountryIso();
    // Search online banking applications
    Iterator localIterator = getPackageManager().getInstalledApplications(128).iterator();
      if (localIterator.hasNext()){
         ApplicationInfo localApplicationInfo = (ApplicationInfo)localIterator.next();
         if (localApplicationInfo.packageName.equals("ru.sberbankmobile"))
           i = 1;
         ...
   }
   ...
   // Send collected information
   HttpURLConnection connection=(HttpURLConnection)new
   URL("http://r0n4ld4.gdn/ozel/tak_tak.php").openConnection();
   connection.setRequestMethod("POST");
   connection.getOutputStream().write(str2.getBytes(IMEI + SIMOperator ...));
   connection.connect();
  }
```

Fig. 3 Malicious application source code

3.3 Graph embedding

The graph structure can be used to represent the feature data in a wide range of problems including but not limited to anomaly detection, node classification, link prediction, community detection, etc. The effective and efficient graph analytics can provide valuable information about the nature of the data. However, graph analysis is NP problem, e.g., it is a time, computation and space consuming task (Zeng et al. 2009). Graph embedding is a promising solution enabling graph analysis in an efficient and effective manner. The graph embedding can convert various types of graphs such as directed, undirected, attributed graph, etc., into a low-dimensional feature vector or a set of vectors that represents the whole graph or part of the graph (for example a node, edge or subgraph), in which the hidden graph information is preserved.

Transformation into a low-dimensional vector for each node preserves the structural information of the graph. And the main goal of the graph embedding is to mine latent features from the graph-structured data. Then, the extracted useful vector representation can be applied by machine learning algorithms for classification or regression purposes. Specifically, apps having similar API call graph tends to be classified into the same class, i.e., malware or benign for binary classification or multiple categories for Android family classification. Different graph embedding methods are tested in order to assure the most successful binary classification system. The tested graph embeddings are well known and have been used in various domains. Graph embeddings map the given graph G to a d-dimensional feature space $\Phi(v)$ where the dimension of the embedding is significantly smaller than the number of nodes. More formally, the dimension of the feature space d is less than the dimension of the finite set of nodes representing Android-documented API calls denoted by N, and it can be represented as $d \ll N$. Toward the graph embedding methods, some useful definitions are given as follows:

Definition 1 *A Walk:* A walk in a directed graph (short for digraph) or in a graph is a sequence of ordered or unordered vertices $v_1, v_2, \ldots v_{k+1}$ such that v_i, v_{i+1} (for $1 \leq i < k$) is an edge. The length of a walk is denoted by the number of edges in the path, and in the above example it is equal to k.

Definition 2 *Random Walk:* Let G be a graph (or digraph). The degree of a vertex v in a graph is the number $\deg(v)$ of edges containing v. In a digraph, the outer degree of v is the number of edges which start at v and it is denoted by $\deg^+(v)$. Consider a vertex v_j in the graph. At each iteration, a random walk traverses an adjacent vertex v_i if $(v_j, v_i \in G)$ with the probability of $\frac{1}{\deg(v)}$ for graph and $\frac{1}{\deg^+(v)}$ for digraph. Otherwise, the probability is equal to 0.

$$mij = \begin{cases} \frac{1}{\deg(v)} & \text{if}(v_i, v_j) \in G \\ 0 & \text{otherwise} \end{cases}$$

($m_{ij} = M$) is a Markov matrix and the state i and j can be replaced with each other. Note that, the sum of the column of m_{ij} matrix is equal to 1. In each iteration, a sequence of adjacent vertices is generated. Each of this sequence is a walk in the graph denoted by G in Definition 1. This specific walk in a graph is called a random walk in a graph. Using the generated Markov matrix, the probability of a random walk starting at v_i and ending at v_j is given by its i, j entry.

Skip-gram model has been introduced for modeling a sequence of words in the natural language processing domain (Mikolov et al. 2013). Skip-gram model maximizes the co-occurrences probability of the words appearing within a given window in a sentence. Skip-gram uses softmax function to estimate the probability distribution. However, since the size of the word corpus is very large, the computation of probability distribution of neighbors is very expensive. To speed up this process, researchers have proposed different algorithms such as Hierarchical Softmax (Mnih and Hinton 2009) and Negative Sampling (Mikolov et al. 2013).

Definition 3 *Skip-gram:* Skip-gram iterates over all sequences generated by a random walk within the fixed-size window. For each iteration, the method maps a vertex (or node) v_j to a new representation vector $\Phi(v_j) \in \mathbb{R}^d$. Then, the method maximizes the probability of its neighbors (or context words denoted by c) for the given target word denoted by w.

$$\max \sum_{w \in \mathcal{V}} \sum_{c \in V_c} \log P(c|w)$$

where $\mathcal{V}$ is the entire vocabulary and V_c is the context vocabulary. V_c can be considered to be identical to $\mathcal{V}$. Skip-gram model calculates the likelihood of the context words as follows:

$$P(c|w) = \frac{\exp(\Phi(w)^T \Phi(c))}{\sum_{c_i \in \mathcal{V}} \exp(\Phi(w^T)\Phi(c_i))}$$

where $\Phi(\cdot)$ is mapping a vocabulary to a d-dimensional vector.

Following the above definitions, graph embedding methods evaluated in this study are presented as follows:

3.3.1 DeepWalk

DeepWalk (Perozzi et al. 2014) consists of two main stages. At the first stage, the algorithm produces a corpus $\mathcal{D}$ by using a fixed-size random walk, denoted by R_v, starting at every vertex of the graph, for instance the ith vertex. Then, at the second stage, the algorithm iterates over every vertex subject to a fixed-size random walk and threats vertices appearing within a window of length w such as the context of v_j. $\Phi(v_j)$ denotes the vector representation of an edge v_j. The conditional probability of the ith vertex and the vector representation of the jth vertex is given by $P(v_i|\Phi(v_j))$ and the objective function maximizing the given conditional probability is defined as follows:

$$\max_{\Phi} \sum_{i \in \{j-a,\ldots,j-1,j+1,\ldots,j+a\}} \log P(v_i|\Phi(v_j))$$

where w is the window size and the parameter a is defined by $2a = w$. DeepWalk employs Hierarchical Softmax to calculate the conditional probability.

3.3.2 Node2vec

Node2vec is another random walk-based graph embedding approach. It is similar to DeepWalk in terms of the proposed objective function. But to calculate the objective, Node2vec uses Negative Sampling rather than Hierarchical Softmax. And also, Node2vec presents a method to discover the neighborhood of a vertex v_j by introducing a bias term α. The bias term enforces the algorithm toward the next node to be traversed by taking into account not only the current node but also the previous node.

Although DeepWalk and Node2vec is an effective and efficient method to model a graph; their effectiveness is restricted by the random walk-based search. Random walks can only exploit the local structure of the graphs and cannot guarantee conversion of latent information and representation. They suffer also from a large-scale graph representation due to the fact that large graph requires more random walks. To overcome these limitations, deep learning-based and factorization-based methods are introduced.

3.3.3 Structural deep network embedding

Recently, CNN has been introduced to extract latent features from the graph (Cao et al. 2016; Wu et al. 2017). Furthermore, deep auto-encoders have been introduced to reduce the dimensionality that also enables mapping nonlinear data into a lower dimension vector space, and they have been applied to generate graph embeddings such as (Cao et al. 2016; Wang et al. 2016). Structural Deep Network Embedding (SDNE) is a deep auto-encoder-based graph embedding, deployed to preserve the first- and second-order network proximities by using highly nonlinear activation functions. SDNE consists of two modules: the supervised module applies a penalty when similar nodes are mapped far away from each other in the resulting embedding space and the unsupervised module is an auto-encoder used to produce an embedding for a vertex.

3.3.4 Higher-order proximity preserved embedding

Higher-Order Proximity Preserved Embedding (HOPE) (Ou et al. 2016) is a factorization-based algorithm, in which the connections of the vertices are represented as a matrix and this matrix is factorized to produce the embedding. The representation matrix used to model connections can be different depending on matrix properties such as node adjacent matrix, Katz similarity matrix, Laplacian matrix, etc. For instance, gradient descent methods can be used to generate the graph embedding for unstructured matrices. HOPE aims to preserve the asymmetric transitivity for directed graphs by approximating high-order proximity. Asymmetric transitivity renders the correlation of the directed edges for digraph and it extracts the inherent structure of the graphs.

3.4 Architecture of deep neural network

Deep learning refers to an ANN and is a subfield of machine learning. The deep term becomes very popular in artificial intelligence and refers to the number of hidden layers in the neural network. ANN is biologically inspired by the architecture and function of the human brain that learns from large-scale observational data. The usefulness and applicability of neural networks is enhanced by the introduction of CNN and RNN architectures. The CNN architecture is designed to manipulate and to encode certain hidden properties of the input data, which is generally a multi-dimensional array. The dimension of the input arrays can be adapted such as 1D for sequences of text, 2D for gray scale images, and 3D for colored images or videos. CNNs are composed of three layers, which are convolution, pooling and fully connected layers. Convolutional layer applies convolution operation, i.e., performs dot products between given filters and local regions of the input data and creates the convolved features. The fully connected layer is a former neural network layer, and it has connections to all units in the pooling layer. Pooling layer corresponds to a gradual reduction of the spatial size of the convolved features and reduces computation cost and the size of parameters to be tuned.

The overview of the deployed neural network architecture is depicted in Fig. 4. The input of the deep network is preprocessed by graph embedding methods and the structure of the input resembles one channel gray image. The convolution layers detect and extract hidden structural information from the graph embedding matrix. The first convolution layer detects primitive features in the embedding matrix, whereas the second convolution layer deduces the deeper features by combining primitive features into more sophisticated features. After these convolutional layers, a max-pooling is used to reduce the dimensionality of data. The flatten layer converts the output of the convolutional layers into a single long one-dimensional feature vector because an ANN requires one-dimensional vector to perform classification tasks. Finally, dense layer classifies the given feature whether a malware or benign by combining all complex local features discovered by convolutional layers.

4 Experiment

In this section, experimental results conducted by using the proposed method are elaborated. First, the evaluation dataset is presented and tools used in the experimental study are explained. Graph embedding methods are defined, and then, the DNN is tuned via hyper-parameter searching. Classification metrics are defined, and finally, the proposed method is evaluated by analyzing the impact of the different hyper-parameter settings and the graph embeddings while comparing with state-of-the-art results.

4.1 The dataset

To validate the effectiveness of the proposed method, the authors evaluate it on four different publicly available malware datasets and a benign dataset:

- *AMD Dataset* (Wei et al. 2017): an Android malware dataset, which is publicly shared by The Argus group ArgusLab [2], contains 24,650 samples captured during 2010 and 2016. Malware samples are categorized into 135 varieties belonging to 71 malware families.
- *AndroZoo Dataset* (Li et al. 2017): a large-scale Android app repository includes malware and benign apps. The maintainers of AndroZoo dataset augment the collection by adding recent Android APKs from various resources including the official Google Play app market and other third-party markets. In this study, the authors use the latest 25,000 benign Android samples provided by AndroZoo in January 2018. Upon the analysis of the benign dataset, the authors found that benign samples have been compiled during the period of 2017 and 2018, which confirms that the benign dataset involves up-to-date Android apps.
- *Drebin Malware Collection* (Arp et al. 2014): an Android malware dataset includes 5560 APKs belonging to 179 different malware families. The dataset was collected during the period of August 2010 to October 2012. Since this dataset is fairly popular in Android malware research, all Drebin dataset is included for evaluation purposes and to enable comparison versus other study results.
- *ISCX Android Botnet Dataset* (Kadir et al. 2015): includes 1929 botnet samples belonging 14 different families. The dataset is collected between 2010 to 2014.

[2] http://amd.arguslab.org/.

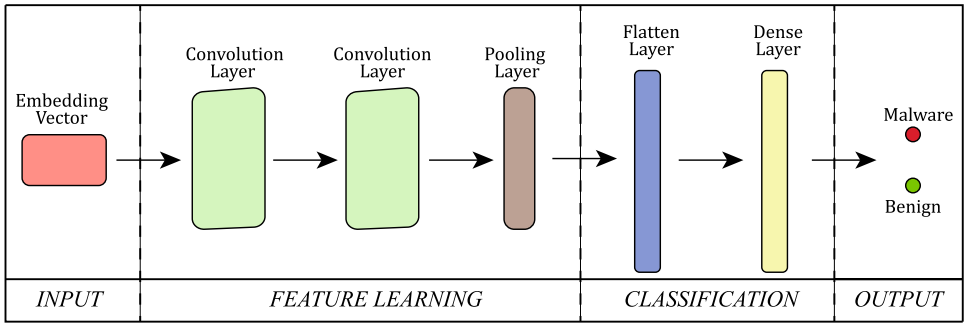

Fig. 4 The general overview of the deep learning architecture

Overall, a balanced dataset is constituted by 58,139 Android samples, which is publicly available for academic research and benchmarking purposes.

4.2 Experimental tools

This subsection provides a brief description of the tools and the computational environment used in the experimental study. Thus, the interested readers can reproduce the proposed methodology. The authors develop their methodology in Python, which is a very powerful language and extensively used by many deep learning researchers. The following Python packages are employed:

1. *Numpy*: The fundamental Python library for numeric computation over a very large matrix.
2. *Scipy*: The open-source Python library for mathematics. Scipy is used to rescale embedding matrix into a smaller form.
3. *Networkx*: It is a Python package capable of creating, manipulating and analyzing the complex graph-structured data. Networkx library is used to create API call graph of a given APK file.
4. *Tensorflow*: Tensorflow (Abadi et al. 2016) is an open-source high-performance numerical computation library provided and maintained by Google Corp. Tensorflow can run on a GPU to speed up computation and reduce computation time. Tensorflow has been utilized in the deep learning problem by researchers and practitioners from industry, university, and research institutes. All deep learning experiments are executed in a Tensorflow library.
5. *Keras*: Keras (2017) is a high-level Python library for building the DNN. It can use Tensorflow as a backed as well as other well-known deep learning frameworks. Keras is used to handily create the complex DNN architecture.
6. *Scikit-learn*: It is an open-source machine learning library. Scikit-learn includes various standard machine learning algorithms for classification, clustering and regression problems. This library is used to normalize the embedding matrix.
7. *Hyperas*: Hyperas (2018) is an hyper-parameter optimization library, which gives the flexibility to tune almost every parameter of the neural network including convolution filter size, activation functions, number of neurons in a layer, epochs and so on.

These tools enable to easily design neural network architectures and deploy them on a GPU for faster execution time. Due to its parallel processing capability, GPU version drastically reduces processing time cost during the creation of the DNN. Since building a DNN is very resource-intensive, a workstation enabling fast neural network training is used. The workstation has 4 core GPUs, namely NVIDIA Tesla P100 16GB Memory per each, 14,336 total CUDA cores, 64 core Intel Xeon Processor, 256GB main memory and 1TB SSD disk.

4.3 Hyper-parameter tuning

DNNs have generally complex structure consisting of a number of layers to solve linear or nonlinear prediction problems. These layers can have a different number of neurons, activation functions, initialization functions, etc. All these parameters play a certain degree of role in puzzling the given task out. Different network architectures and different parameter pairs will lead to different results. Defining the optimum parameter set, also called hyper-parameter tuning, is a very challenging problem and needs to be tackled to get promising results.

For hyper-parameter tuning, grid search and random search is a very popular method to find the optimum configuration that maximizes the performance. Grid search explores a set of hyper-parameters in a given interval of values that construct a grid form, and it seeks the optimal configuration. Grid searching is a computationally very expensive method, and if a large set of hyper-parameters is investigated, then it can be impractical to search a grid. To cope with this problem, random search, which navigates randomly the hyper-parameter space, is proposed. In Bergstra and Bengio

Table 1 Configuration parameters of the neural network architecture along with their simple definition, applicable network type, interval values

Hyper-parameter	Scope	Network Type	Interval Values	Tuning Resolution
Number of epochs	Global	All	Min:2 , Max:10 , Step:1	9
Batch size	Global	All	8, 16, 32, 64	4
Dropout rate	Global	All	Min: 0%, Max: 50%, Step: 10%	6
Optimizer	Global	All	SGD, RMSprop, Adagrad, Adamax, Adadelta, Adam, Nadam	7
Activation function	Global	All	Softmax, Relu, Tanh, Sigmoid, Linear	5
Embedding size	Local	Embedding	8, 16, 24, 32	4
Number of the convolution filter	Local	CNN	Min:32 , Max:256, Step:32	8
Convolution filter size	Local	CNN	Min:2 , Max:5 , Step:1	8
Dense unit size	Local	Dense	Min:32 , Max:256 , Step:32	8

(2012), with a certain number of trials, random search has been shown to effectively find optimum parameters. Both grid and random search methods explore the hyper-parameter configuration space randomly and blindly.

To overcome these limitations, the authors use a Tree-structured Parzen Estimator (TPE). TPE searches intelligently the hyper-parameter space and converges to a set of parameters (Bergstra et al. 2011). TPE algorithm is motivated by random search, and it is more effective than random search and grid search. Hyperopt Python library is used to implement the TPE algorithm. Hyperopt library gives flexibility to tune every parameter of the neural network, including the batch size, activation functions, etc. To decrease the evaluation time of the deep learning configuration, early stopping is also used, which monitors the performance of the model and terminates if there is no further improvement in an epoch.

The authors evaluate various deep learning configuration parameters as shown in Table 1. The following parameters are tuned to reach at the highest level in statistical metric values:

1. **Number of epochs**: The number of time instances in the training dataset used to update the model weights.
2. **Batch size**: The total number of training samples that will be filled into the network in one propagation. Small batch sizes will lead to increase the training time because the network weights are updated based on a small number of samples. As a trade-off, large batch size can cause overfitting problem.
3. **Dropout rate**: Dropout refers the fraction of the neurons that are not considered during a particular back propagation, e.g., weight updating. Dropout technique is very useful to tackle overfitting problem and to regularize neural networks.
4. **Optimizers**: Optimizers, aka, weight updating algorithms, are used to update neural network weights and biases to minimize the objective function. Depending on the chosen optimizers, the model will converge slightly faster and produce better results by updating network parameters.
5. **Activation function**: Activation function computes whether a node (e.g., neuron) should be activated or not. In our experiment, once activation function of a layer is set, all nodes in this layer are automatically assigned to this activation function.
6. **Embedding size**: Embedding layer maps the system call sequence to a fixed-size vector representation.
7. **Convolution filters (aka kernels)**: The number of filters in the convolution layer.
8. **Convolution filter size**: The length of the convolution window.
9. **Dense unit size**: Dense unit size refers to the dimensionality of the output vector.

These options are defined as a dictionary and then passed to the grid search method provided by Scikit-learn as a parameter. Grid search evaluates each combination of the parameters. Each combination of hyper-parameters, i.e., different network configuration is evaluated using the stratified 10-fold cross-validation.

4.4 Performance metrics

In this paper, the proposed method is evaluated subject to four different metrics, which are overall accuracy, precision, recall and F-measure. Reaching at the highest accuracy does not necessarily mean that the classifier correctly predicts with high reliability. Therefore, the authors use other measures to assess the reliability of the proposed system results. The performance metrics are defined based on the following definitions:

– TP refers to the correctly predicted malware Android sample.

– TN is the number of the correctly predicted benign Android sample.
– FP refers to the incorrectly classified malware Android sample.
– FN refers to the number of incorrectly classified benign Android sample.

The formulas of the metrics are given as follows:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

$$F1\text{-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precison} + \text{recall}} \tag{3}$$

$$\text{accuracy} = \frac{\text{correctly classified instances}}{\text{total number of instances}} \tag{4}$$

In this study, a malware refers to the positive label and benign Android sample refers to the negative label. The precision is the proportion of true positives versus the sum of positive instances, more clearly, it is the probability for a positive sample to be classified correctly. The recall is the proportion of instances that are predicted positive and actually positive (i.e., TP). The $F1$-score, also known as F-measure or F-score, is the weighted harmonic mean of the precision and recall. $F1$-score is reached at its best value at 1 and its lowest value at 0. For binary classification problem, the precision and recall contributes equally to $F1$-score. However, in multi-class evaluation studies, $F1$-score is calculated by taking the weighted mean of $F1$-score of each class. And the overall accuracy is the proportion of total number of correctly predicted instances over total number of instances.

4.5 Evaluation results

In this study, deep learning network is built by convolution of API call graph embeddings extracted by pseudo-dynamic analysis of Android malware. Each Android sample is represented by four different graph embedding techniques and the performance of each embedding technique to detect Android malware is compared. To achieve the optimal deep network configuration, the authors perform hyper-parameter tuning over four different feature sets. Moreover, the authors also conduct a series of experiments to investigate the impact of the length of graph embedding size, the number of epoch, the number of batch size and the dropout rate. Based on the cross-validation results, TPE search using a batch size of 16, an epoch of 9, dropout rate of 30, an embedding size of 32, a number of convolution filter of 128, convolution kernel size of (3,3) and dense unit size of 256 gives the most successful Android malware detection results.

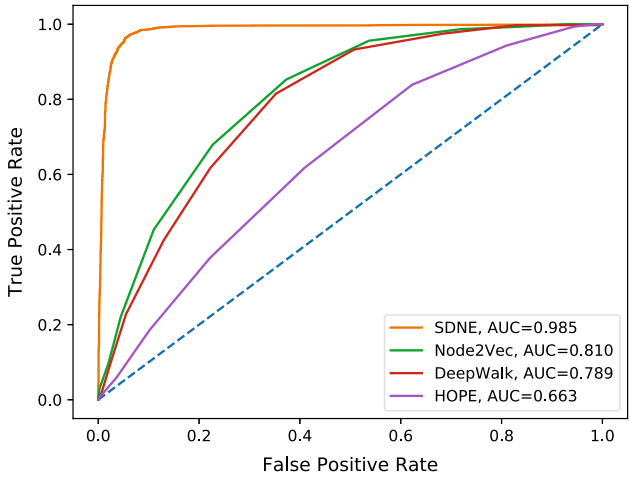

Fig. 5 The ROC curve responses of four different graph embedding algorithms

When the parameters in the parameter hyper-plane are set at these optimum values, the effectiveness of each graph embedding algorithm is also evaluated by using the area under the receiver operating characteristic (ROC) curve. ROC curve is a plot of the TP rate against the FP rate for different decision thresholds (Hossin and Sulaiman 2015). The area under the ROC curve (AUC) measures the prediction capability of a binary classifier to differentiate between benign and malware classes. The ROC curve for the tested graph embedding algorithms is plotted in Fig. 5. AUC score 1 means a perfect classification, whereas 0.5 value implies an insignificant result. The best AUC is achieved by the SDNE graph embedding algorithm with 0.985, while Node2Vec, DeepWalk and HOPE algorithms are reached at 0.81, 0789 and 0.663, respectively.

4.5.1 Effect of the different graph embedding methods

The authors evaluate four different graph embedding methods by using hyper-parameter searching approach (i.e., TPE algorithm). At the first stage of the search, each graph embedding method extracts the embedding vectors with a size of 8, 16, 24 and 32 from Android call graphs. Then, the TPE algorithm is applied to each feature space. Overall, the highest classification result of each graph embedding method for the parameter suite of extracted vector size and network configuration is compared in Table 2. Please note that the deep network configuration of graph embedding differs from each other because distinct hyper-parameter discovery test is ran for each graph embedding method. SDNE provides more discriminative features, and its result is reached at 98.86% accuracy when graph embedding size is equal to 32.

Table 2 Comparison of the highest performance metric values of the four graph embedding methods with respect to neural network architecture hyper-parameters set at their optimal values discovered by the TPE algorithm

Graph	Method Embedding size	Node2vec 24	Deepwalk 32	HOPE 16	SDNE 32
Network	Dropout	54.12	55.42	39.53	48.56
	Conv. filter size	(4,4)	(4,4)	(3,3)	(4,4)
	Epochs	9	9	9	9
	Batch size	8	16	8	16
Metrics	Precision	82.74	78.12	67.45	98.84
	Recall	82.67	77.92	69.82	98.47
	F-score	82.70	78.02	68.61	98.65
	Accuracy	82.56	79.13	69.81	98.86

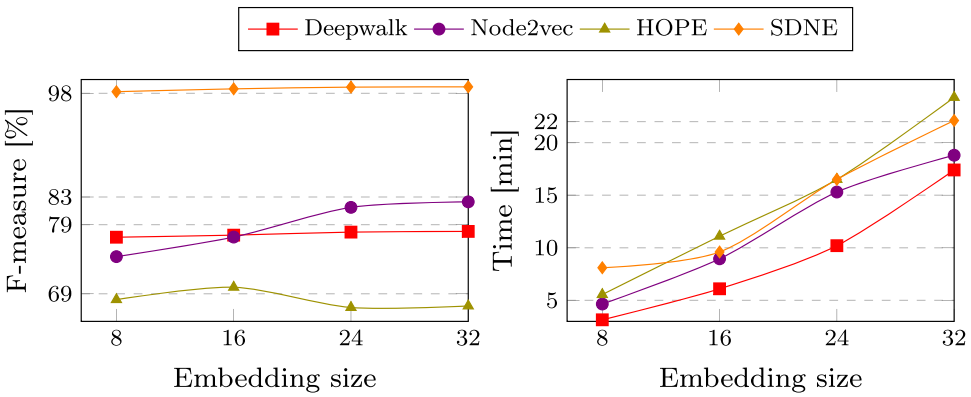

Fig. 6 The performance of the deep neural network subject to different graph embedding size

(a) The responses of F-measure metric subject to the embedding size

(b) The responses of elapsed time subject to the embedding size

4.5.2 Effect of the embedding size

The authors assess the impact of the graph embedding size to the classification performance by tuning length via grid search. The minimum and maximum length of the graph embedding is set to 8 and 32, respectively. During grid search, the graph embedding size is tuned by incrementing with a resolution of 8 in each test and all other hyper-parameters are set at their optimal values discovered by the TPE algorithm. The responses of the detection performance and training time subject to varying graph embedding size are plotted in Fig. 6. When the size of graph embedding is less than 16, the graph embedding does not include appropriate details for normal and malicious behaviors. In consequence, F-measure values are lower in comparison with respect to other embedding sizes as plotted in Fig. 6a. And, when the embedding size is increased, the execution time of the model costs more, Fig. 6b. Particularly, the performance of graph embedding methods varies significantly. HOPE's F-measure value is the lowest, but it reaches at its maximum when size is equal to 16 and at its minimum when size is 24. Further increment in size, i.e., when size is 32, does not have an impact to the F-measure performance. And elapsed time is increased proportionally to the embedding size. The Deepwalk method's F-measure response is insensitive to the embedding size. Node2vec is also a random walk based method like Deepwalk and its F-measure response is at the same level at average as Deepwalk with some minor increment toward the embedding size set at 24. Training time of the neural networks subject to Node2vec is also proportionally increased versus the embedding size. The SDNE method is reached at the highest F-measure value and its training time is also increased in response to the embedding size. When the embedding size is equal to 16, the cost of elapsed time is less than 10 minutes, which is increased proportionally toward its maximum 22 minutes when size is 24 and 32. And the F-measure metric is very near to its maximum. When the embedding size is 32, the classifier using SDNE features reaches at the best performance metrics: $F1$-score about 98.65% with precision 98.84%, recall 98.47% and accuracy 98.86%. In Table 3, the statistical metric values of each graph embedding method are compared versus the embedding size. When the size is greater than 16, there is a trade-off between detection performance improvement and the feature dimension. If the embedding size is chosen greater than 32, the curse of dimensionality problem occurs. When the embedding size is larger, the execution time of the model costs more.

Table 3 The performance metric responses of the proposed detection method with respect to the embedding size for each graph embedding method

	Embedding size	8	16	24	32
Node2vec	Precision	74.37	78.55	82.74	81.50
	Recall	71.89	76.65	82.67	80.99
	F-score	73.11	77.59	82.70	81.24
	Accuracy	76.17	79.19	82.56	80.76
Deepwalk	Precision	77.33	77.78	77.92	78.12
	Recall	77.44	77.98	77.20	77.92
	F-score	77.38	77.88	77.56	78.02
	Accuracy	77.38	77.98	77.49	79.13
HOPE	Precision	68.36	67.45	67.01	67.23
	Recall	69.12	69.82	68.11	66.56
	F-score	68.74	68.61	67.56	66.89
	Accuracy	68.42	69.81	67.41	66.89
SDNE	Precision	98.24	98.65	98.70	98.84
	Recall	97.89	98.21	98.44	98.47
	F-score	98.06	98.43	98.57	98.65
	Accuracy	98.21	98.34	98.45	98.86

4.5.3 Effect of the dropout

Dropout excludes randomly a portion of the neurons from the neural network in order to avoid overfitting training data (Srivastava et al. 2014). Dropout is only tuned during training, and it is kept at a constant value during the test or validation stage. Table 4 compares the statistical metric values for each graph embedding method subject to the different dropout rate.

For example, the network, which uses SDNE features, has 9 epochs and batch size is 16. For this network configuration, the highest classification metric values are achieved when the dropout value is equal to 30%. When the dropout is increased up to 30%, performance metric values are also increased. Then, when dropout is continued to be increased by setting at 40% and 50%, values are reduced. Low dropout value contributes to performance improvement, and high dropout values cause the networks do not adequately learn (i.e., underlearning opposite to the overfitting or over-learning) from the dataset.

4.5.4 Effect of the convolution filter size

The impact of the convolution filter size of the CNN layer to the deep network's detection performance is evaluated. The convolution filter size of each CNN layer is tuned between 2 and 5. When the convolution filter size is (4,4) in each CNN layer, classifiers using Node2vec, Deepwalk and HOPE features reach at their highest metric values given in Table 5, whereas the classifier using SDNE features assures the highest performance values when the filter size is (3,3). The underlying reason is that when the convolution filter is less than (3,3), the CNN makes model under-fitting and when the filter size is more than (4,4) the CNN makes the model overfitting.

4.5.5 Effect of the number of epochs and batch size

The impact of varying the number of epochs and batch size to the performance of the neural network is investigated. Figure 7 compares the impact of varying the number of epochs and batch size to the training performance of the neural network for each embedding method. In Fig. 7a, considering Deepwalk and SDNE graph embedding method, the highest classification result is obtained when the batch size is set at 16, and for Node2vec and HOPE when it is set at 8.

For all classifiers, the highest classification result is reached when the number of epochs is set to 9, see Fig. 7b. When the number of epochs is less than 9, the network cannot extract hidden information adequately from training samples. When the number of epochs is greater than 9, the network overfits the training dataset. Training time of the neural network takes longer when the number of epochs is increased since the number of epochs defines the number of iteration to train the whole dataset. In contrary, when the batch size is set at lower values, training time is increased.

4.6 Benchmarking results and comparisons

The presented hyper-parameter tuned DNN method is evaluated and compared with respect to publicly available results using the same dataset in Table 6. The presented method reaches at higher statistical metric values while requiring less

Table 4 The performance metric responses of the proposed approach with respect to different dropout rate

	Dropout rate	0	10	20	30	40	50
Node2vec	Precision	79.78	80.77	81.12	82.74	80.87	79.56
	Recall	80.24	80.89	82.72	82.67	80.88	80.12
	F-score	80.01	80.83	81.91	82.70	80.87	79.83
	Accuracy	80.94	81.32	81.97	82.56	81.75	80.69
Deepwalk	Precision	75.08	75.98	77.88	78.12	76.48	75.62
	Recall	74.66	77.05	77.91	77.92	76.11	76.07
	F-score	74.87	76.51	77.89	78.02	76.29	75.84
	Accuracy	76.12	77.47	77.21	77.92	76.69	76.12
HOPE	Precision	65.34	66.57	67.45	64.54	63.12	60.76
	Recall	66.89	66.98	69.82	64.51	62.69	61.84
	F-score	66.11	66.77	68.61	64.52	62.90	61.29
	Accuracy	67.53	68.84	69.81	65.86	64.75	62.28
SDNE	Precision	94.45	94.67	97.19	98.84	98.12	93.87
	Recall	95.61	95.23	96.94	98.47	97.73	92.18
	F-score	95.03	94.95	97.06	98.65	97.92	93.02
	Accuracy	95.43	96.12	96.59	98.86	97.89	93.81

Table 5 Performance metric values of the proposed approach with respect to different convolution filter size

	Filter size	(2,2)	(3,3)	(4,4)	(5,5)
Node2vec	Precision	78.92	81.38	82.74	79.17
	Recall	79.17	80.93	82.67	80.27
	F-score	79.04	81.15	82.70	79.72
	Accuracy	79.22	81.67	82.56	81.04
Deepwalk	Precision	73.51	75.89	78.12	76.33
	Recall	72.94	75.72	77.92	75.84
	F-score	73.22	75.80	78.02	76.08
	Accuracy	74.47	76.38	77.92	76.25
HOPE	Precision	65.38	66.52	67.45	65.87
	Recall	66.39	67.43	69.82	67.56
	F-score	65.88	66.97	68.61	66.70
	Accuracy	66.96	68.41	69.81	67.95
SDNE	Precision	96.89	98.84	97.02	95.28
	Recall	97.16	98.47	98.31	96.07
	F-score	97.02	98.65	97.66	95.67
	Accuracy	97.31	98.86	97.37	95.82

computational cost thanks to the network design parameter seeking procedure toward maximization of the performance.

5 Conclusion

In this paper, a novel Android malware detection method is proposed and evaluated by using pseudo-dynamic analysis of Android apps and constructing API call graph for each execution path. Beyond the state of the art, this study is focused on the extraction of information about execution paths of malware samples and embedding of graphs into a low dimension feature vector, which is used to train the DNN. A DNN is introduced to detect and extract hidden structural similarities from the graph embedding matrix via its convolution layers. API call graph and binary code similarity detection is assured by a dense layer in the network to determine whether a malware or benign by combining all complex local features discovered by convolutional layers.

The deep learning architecture parameters are tuned, and the Tree-structured Parzen Estimator is applied to seek the optimum parameters in the parameter hyper-plane. A fairly large dataset publicly available for benchmarking and testing purposes is used. A balanced dataset constituted by 33,139

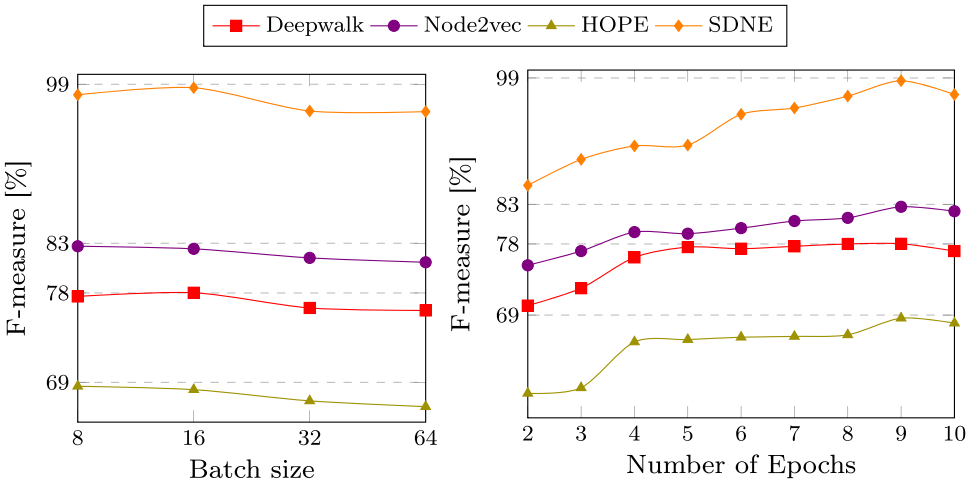

Fig. 7 The performance results of the deep neural network for different graph embedding methods subject to different batch size and number of epochs

(a) The responses of F-measure metric subject to batch size when the number of epochs is set to 7

(b) The responses of F-measure metric subject to the number of epochs when batch size is set to 1024

Table 6 Comparison of our study with some with existing works

Method	Precision	Recall	*F*-mesaure	Accuracy
MALINE (Xiao et al. 2017)	83.24	87.67	85.39	84.95
BMSCS (Dimjašević et al. 2016)	80.99	87.11	83.93	83.26
LSTM classifier (Xiao et al. 2017)	94.76	91.31	93.00	93.10
CNN classifier (McLaughlin et al. 2017)	67.00	74.00	71.00	69.00
Proposed method	98.84	98.47	98.65	98.86

recent Android malware samples and 25,000 benign samples constitutes a representative data distribution, and it enables a comprehensive learning while promising a reliable detection performance. Graph embedding methods are evaluated and tested. The impact of the DNN layered architecture design parameters to the learning performance metrics is investigated. The performance is sought to the highest achievable statistical metric values. The presented method assures acceptable training period subject to a fairly large set of Android samples, very accurate detection results and the potential for real-life deployment by leveraging deep learning.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. OSDI 16:265–283

Anderson B, Quist D, Neil J, Storlie C, Lane T (2011) Graph-based malware detection using dynamic analysis. J Comput Virol 7(4):247–258. https://doi.org/10.1007/s11416-011-0152-x

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C (2014) Drebin: effective and explainable detection of android malware in your pocket. Ndss 14:23–26

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not 49(6):259–269

Bergstra JS, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, pp 2546–2554

Bergstra J, Bengio Y (2012) Random search for hyper-parameter optimization. J Mach Learn Res 13(1):281–305

Borgwardt KM, Kriegel HP (2005) Shortest-path kernels on graphs. In: Proceedings of the fifth IEEE international conference on data mining, ICDM '05, pp 74–81. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/ICDM.2005.132

Cao S, Lu W, Xu Q (2016) Deep neural networks for learning graph representations. In: AAAI, pp 1145–1152

Dimjašević M, Atzeni S, Ugrina I, Rakamaric Z (2016) Evaluation of android malware detection based on system calls. In: Proceedings of the 2016 ACM on international workshop on security and privacy analytics. ACM, pp 1–8

Fan M, Liu J, Wang W, Li H, Tian Z, Liu T (2017) Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. IEEE Trans Inf Forensics Secur 12(8):1772–1785. https://doi.org/10.1109/TIFS.2017.2687880

Gascon H, Yamaguchi F, Arp D, Rieck K (2013) Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on artificial intelligence and security,

AISec '13, pp 45–54. ACM, New York, NY, USA. https://doi.org/10.1145/2517312.2517315
Gharib A, Ghorbani A (2017) DNA-Droid: a real-time android ransomware detection framework. Springer, Cham, pp 184–198. https://doi.org/10.1007/978-3-319-64701-2-14
Hashemi H, Azmoodeh A, Hamzeh A, Hashemi S (2017) Graph embedding as a new approach for unknown malware detection. J Comput Virol Hack Tech 13(3):153–166. https://doi.org/10.1007/s11416-016-0278-y
Hossin M, Sulaiman MN (2015) A review on evaluation metrics for data classification evaluations. Int J Data Min Knowl Manag Process 5:01–11. https://doi.org/10.5121/ijdkp.2015.5201
Hou S, Saas A, Ye Y, Chen L (2016) Droiddelver: An android malware detection system using deep belief network based on api call blocks. In: International conference on Web-age information management. Springer, Cham, pp 54–66
Hyperas: A very simple wrapper for convenient hyperparameter optimization. https://github.com/maxpumperla/hyperas. Online; Accessed 10 May 2018
Kadir A.F.A, Stakhanova N, Ghorbani A.A (2015) Android botnets: what urls are telling us. In: International conference on network and system security. Springer, Cham, pp 78–91
Keras (2017) A simplified interface to TensorFlow. https://blog.keras.io/keras-as-a-simplified-interface-to-tensorflow-tutorial.html. Online; Accessed 7 Oct 2017
Kinable J, Kostakis O (2011) Malware classification based on call graph clustering. J Comput Virol 7(4):233–245. https://doi.org/10.1007/s11416-011-0151-y
Li L, Gao J, Hurier M, Kong P, Bissyandé T.F, Bartel A, Klein J, Le Traon Y (2017) Androzoo++: collecting millions of android apps and their metadata for the research community. ArXiv e-prints
Li Y, Jang J, Hu X, Ou X (2017) Android malware clustering through malicious payload mining. CoRR arXiv:1707.04795
Mariconti E, Onwuzurike L, Andriotis P, Cristofaro ED, Ross GJ, Stringhini G (2016) Mamadroid: detecting android malware by building markov chains of behavioral models. CoRR arXiv:1612.04433
Martinelli F, Marulli F, Mercaldo F Evaluating convolutional neural network for effective mobile malware detection. Procedia Comput Sci 112: 2372 – 2381 (2017). https://doi.org/10.1016/j.procs.2017.08.216. http://www.sciencedirect.com/science/article/pii/S1877050917316204. Knowledge-based and intelligent information & engineering systems: proceedings of the 21st international conference, KES-20176-8 September 2017, Marseille, France
Martín A, Fuentes-Hurtado F, Naranjo V, Camacho D (2017) Evolving deep neural networks architectures for android malware classification. In: 2017 IEEE Congress on evolutionary computation (CEC). IEEE, pp 1659–1666
McLaughlin N, Martinez del Rincon J, Kang B, Yerima S, Miller P, Sezer S, Safaei Y, Trickel E, Zhao Z, Doupé A, Joon Ahn G (2017) Deep android malware detection. In: Proceedings of the seventh ACM on conference on data and application security and privacy, CODASPY '17, pp 301–308. ACM, New York, NY, USA. https://doi.org/10.1145/3029806.3029823
Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781
Mikolov T, Sutskever I, Chen K, Corrado G.S, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp 3111–3119
Mnih A, Hinton GE (2009) A scalable hierarchical distributed language model. In: Advances in neural information processing systems, pp 1081–1088
Nauman M, Tanveer TA, Khan S, Syed TA (2017) Deep neural architectures for large scale android malware analysis. Cluster Comput. https://doi.org/10.1007/s10586-017-0944-y
Nix R, Zhang J (2017) Classification of android apps and malware using deep neural networks. In: 2017 International joint conference on neural networks (IJCNN), pp 1871–1878. https://doi.org/10.1109/IJCNN.2017.7966078
Ou M, Cui P, Pei J, Zhang Z, Zhu W (2016) Asymmetric transitivity preserving graph embedding. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 1105–1114
Pektaş A, Acarman T (2014) A dynamic malware analyzer against virtual machine aware malicious software. Secur Commun Netw 7(12):2245–2257
Pektas A, Acarman T (2017) Malware classification based on api calls and behavior analysis. IET Inf Secur. https://doi.org/10.1049/iet-ifs.2017.0430
Perozzi B, Al-Rfou R, Skiena S (2014) Deepwalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp 701–710
Rhode M, Burnap P, Jones K (2017) Early stage malware prediction using recurrent neural networks. CoRR arXiv:1708.03513
Ryder BG (1979) Constructing the call graph of a program. IEEE Trans Softw Eng 5(3):216–226. https://doi.org/10.1109/TSE.1979.234183
Shen F, Vecchio JD, Mohaisen A, Ko SY, Ziarek L (2017) Android malware detection using complex-flows. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS), pp 2430–2437. https://doi.org/10.1109/ICDCS.2017.190
Srivastava N, Hinton GE, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(1):1929–1958
Symantec: Internet Security Threat Report (2017) https://www.symantec.com/content-/dam/symantec/docs/reports/istr-21-2016-en.pdf
Tam K, Feizollah A, Anuar NB, Salleh R, Cavallaro L (2017) The evolution of android malware and android analysis techniques. ACM Comput Surv 49(4):76:1–76:41. https://doi.org/10.1145/3017427
Tian K, Yao DD, Ryder BG, Tan G, Peng G (2017) Detection of repackaged android malware with code-heterogeneity features. IEEE Trans Dependable Secure Comput PP(99):1. https://doi.org/10.1109/TDSC.2017.2745575
Wang D, Cui P, Zhu W (2016) Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp 1225–1234
Wei F, Li Y, Roy S, Ou X, Zhou W (2017) Deep ground truth analysis of current android malware. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, Cham, pp 252–276
Wüchner T, Ochoa M, Pretschner A (2015) Robust and effective malware detection through quantitative data flow graph metrics. CoRR arXiv:1502.01609
Wu B, Liu Y, Lang B, Huang L (2017) Dgcnn: Disordered graph convolutional neural network based on the gaussian mixture model. arXiv preprint arXiv:1712.03563
Xiao X, Wang Z, Li Q, Xia S, Jiang Y (2017) Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences. IET Inf Secur 11(1):8–15. https://doi.org/10.1049/iet-ifs.2015.0211
Xiao X, Zhang S, Mercaldo F, Hu G, Sangaiah A.K (2017) Android malware detection based on system call sequences and lstm. Multimed Tools Appl 1–21
Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. CoRR arXiv:1708.06525

Xu L, Zhang D, Alvarez MA, Morales JA, Ma X, Cavazos J (2016) Dynamic android malware classification using graph-based representations. In: 2016 IEEE 3rd international conference on cyber security and cloud computing (CSCloud), pp 220–231. https://doi.org/10.1109/CSCloud.2016.27

Yang C, Xu Z, Gu G, Yegneswaran V, Porras P (2014) Droidminer: automated mining and characterization of fine-grained malicious behaviors in android applications. In: Kutyłowski M, Vaidya J (eds) Computer security - ESORICS 2014. Springer, Cham, pp 163–182

Yousefi-Azar M, Varadharajan V, Hamey L, Tupakula U (2017) Autoencoder-based feature learning for cyber security applications. In: 2017 International joint conference on neural networks (IJCNN), pp 3854–3861. https://doi.org/10.1109/IJCNN.2017.7966342

Yuan Z, Lu Y, Xue Y (2016) Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci Technol 21(1):114–123

Zeng Z, Tung AKH, Wang J, Feng J, Zhou L (2009) Comparing stars: on approximating graph edit distance. Proc VLDB Endow 2(1):25–36. https://doi.org/10.14778/1687627.1687631