2018 17th IEEE International Conference on Machine Learning and Applications

DLGraph: Malware Detection Using Deep Learning and Graph Embedding

Haodi Jiang
Computer Science Department
New Jersey Institute of Technology
Newark, NJ 07102, USA

Turki Turki*
Computer Science Department
King Abdulaziz University
P.O. Box 80221, Jeddah 21589, Saudi Arabia

Jason T. L. Wang*
Computer Science Department
New Jersey Institute of Technology
Newark, NJ 07102, USA

***Abstract*—In this paper we present a new approach, named DLGraph, for malware detection using deep learning and graph embedding. DLGraph employs two stacked denoising autoencoders (SDAs) for representation learning, taking into consideration computer programs' function-call graphs and Windows application programming interface (API) calls. Given a program, we first use a graph embedding technique that maps the program's function-call graph to a vector in a low-dimensional feature space. One SDA in our deep learning model is used to learn a latent representation of the embedded vector of the function-call graph. The other SDA in our model is used to learn a latent representation of the given program's Windows API calls. The two learned latent representations are then merged to form a combined feature vector. Finally, we use softmax regression to classify the combined feature vector for predicting whether the given program is malware or not. Experimental results based on different datasets demonstrate the effectiveness of the proposed approach and its superiority over a related method.**

***Index Terms*—malware detection, function-call graphs, Windows API calls**

I. Introduction

Malicious software, or malware, brings harm to individual computers or an entire organization's network. Malware occurs in various forms, among which adware and Trojans are commonly seen in today's cyber threats [1]. Adware, or advertising-supported software, generates revenue for its developer. For instance, Lollipop, a family of adware, automatically shows unwanted advertisements in a user's browser and keeps track of the user's behavior in cyberspace. A Trojan, or Trojan horse, is disguised as a legitimate program that tricks users into loading the virus on their systems. For instance, Kelihos-ver3, a family of Trojans, infects a computer system by sending malicious download links to its user. Once the links are clicked, a Trojan horse is downloaded, which turns the computer system into a zombie. With the increasing number of devices connecting to a network, the malware population is rapidly growing, leading a competing race between antivirus software developers and malware producers [2].

In general, a malicious program and its capabilities can be observed either by examining its code, referred to as static analysis, or by executing it in a safe environment, referred to as dynamic analysis [3]. In static analysis, commonly used malware detection features include string signatures, byte-sequence n-grams, control flow graphs, and so on. Most antivirus programs use a signature-based approach, where a signature is a set of manually crafted rules for identifying different groups of known malware types [4]. However, the signature-based approach usually cannot recognize new malware [5]. To solve this problem, researchers proposed to use function-call graphs for malware detection [6]. For example, Hu *et al.* [7] designed a database management system for malware indexing and searching using function-call graphs. Shang *et al.* [8] presented an algorithm to calculate the similarity of function-call graphs.

In this paper, we present a new approach, named DLGraph, for malware detection. Fig. 1 illustrates how DLGraph works. DLGraph employs two stacked denoising autoencoders (SDAs) [9] for representation learning, taking into consideration computer programs' function-call graphs and Windows application programming interface (API) calls. Given a program, we first extract the function-call graph from the program. This function-call graph is then mapped to a vector in a low-dimensional feature space using the graph embedding technique, node2vec [10]. The first SDA in our deep learning model is used to learn a latent representation, v_1, of the embedded vector of the given program's function-call graph. In addition, we collect the program's Windows API calls to build an API vector. This API vector is then fed into the second SDA in our deep learning model where the second SDA is used to learn a latent representation, v_2, of the API vector of the given program's Windows API calls. We then concatenate v_1 and v_2 to create a combined feature vector v. Finally, a softmax regression layer in our deep learning model classifies v to predict whether the given program is malware or not.

The rest of this paper is organized as follows. Section II reviews related work. Section III details our proposed approach. Section IV evaluates the performance of the proposed method and reports some experimental results. Section V concludes the paper and points out some directions for future research.

II. Related Work

Our work is related to two areas: graph embedding and deep learning. Graph embedding maps a graph into a low-dimensional feature space in which the graph structural information and graph properties are maximumly preserved. It has

*Corresponding authors: tturki@kau.edu.sa, wangj@njit.edu

978-1-5386-6805-4/18/$31.00 ©2018 IEEE
DOI 10.1109/ICMLA.2018.00168

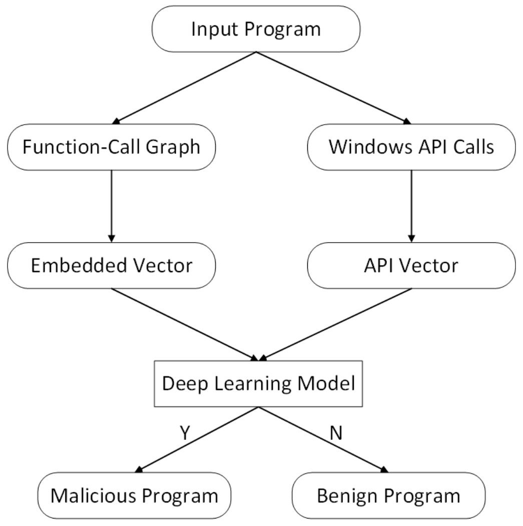

Fig. 1. Overview of our deep learning approach for malware detection.

been widely used in node classification, node recommendation, link prediction, and so on [11]. Existing graph embedding work mainly focuses on undirected graphs. For instance, stru2vec [12] uses a hierarchy to measure node similarities at different scales, and constructs a multilayer graph to encode structural similarities and generate the structural context for nodes. It is able to learn latent representations of the structural identities of nodes.

Among the existing graph embedding algorithms, node2vec [10] is most closely related to our work since it works for directed graphs, specifically function-call graphs, besides undirected graphs. It uses a random walk procedure to explore the neighborhoods of nodes, and maps the nodes to a low-dimensional feature space in such a way that it maximizes the likelihood of preserving the neighborhoods of nodes. In this paper we adopt node2vec for embedding programs' function-call graphs.

Deep learning has received increasing interest in recent years owing to its success in many applications [13]–[15]. Inspired by the success of applying neural networks to computer vision, speech recognition and natural language processing and with the increasing amount of malware data, researchers started to use deep learning techniques for malware detection. For example, Raff *et al.* [4] explored different architectures and parameter choices with raw byte-sequences of executables as input data. Gibert *et al.* [16] introduced convolutional neural networks to classify malware assembly code. Hardy *et al.* [17] developed a system, called DL4MD, which used Windows API calls extracted from malicious portable executable files in combination with a simple stacked autoencoder model for intelligent malware detection. While both our work and DL4MD employ stacked autoencoders, the architectures of these stacked autoencoders are different. Furthermore, DL4MD does not consider programs' function-call graphs. David and Netanyahu [18] presented a tool for learning malware signatures using the dynamic behavior of programs through a deep belief network. In contrast to David and Netanyahu's tool, which runs a given program in a sandbox to generate a text file containing the dynamic behavior of the program, our DLGraph approach performs static analysis by examining the program's function-call graph and Windows API calls. To the best of our knowledge, DLGraph is the first method that combines deep learning and graph embedding for malware detection.

III. Proposed Approach

A. Function-Call Graph

A function-call graph (FCG) shows the calling relationships between subroutines or functions in a computer program. It can be generated from a binary executable through static analysis of the executable code with a disassembly tool. Here we use IDA Pro [19] as the disassembly tool. In general, a function-call graph is a directed graph, in which each node represents either a local function implemented by the program designer, or an external system or library function [20]. The directed edges in the function-call graph represent the caller-callee relationships between the functions (nodes) [21].

Fig. 2 shows a partial FCG of a malicious program, with a global start point in gray, local functions in white boxes, and external functions in black boxes. The directed edge from "sub_421213" to "CreateFileA" means the local function "sub_421213" calls the external function "CreateFileA". To apply deep learning to function-call graphs, we use node2vec [10] to map a function-call graph into a low-dimensional feature space to obtain an embedded vector for the function-call graph. In the study presented here, the embedded vector has about 500 dimensions.

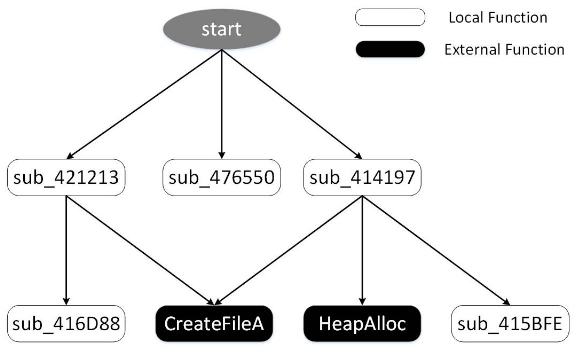

Fig. 2. A partial FCG of a malicious program.

B. Windows API Calls

We extract all the Windows APIs in the computer programs analyzed here, and sort the API names in alphabetical order. We then create an API vector for each program based on a Boolean model where if a call of a Windows API occurs in the program, then the entry in the API vector corresponding to the Windows API is set to 1; otherwise the entry is set to 0.

For example, suppose there are five Windows APIs, named API1, API2, API3, API4 and API5. Consider an API vector $[1, 0, 0, 0, 1]$ for a program. This API vector means the program calls API1 and API5 at least once, and does not call API2, API3, API4 at all. In the study presented here, there are approximately 22,000 Windows APIs in total, and therefore an API vector has about 22,000 dimensions.

C. Dual Stacked Denoising Autoencoders

An autoencoder neural network is an unsupervised learning algorithm consisting of an input layer, a hidden layer, and an output layer. The number of neurons in the input layer is equal to the number of neurons in the output layer. The hidden layer usually has few neurons than the input and output layers. The network is able to learn a hidden, or latent, representation of the input. Specifically, each input vector x of n dimensions is first mapped to the hidden layer, which produces a vector y of m dimensions where $m \leq n$, and the output layer then attempts to reconstruct x from y. The optimization goal here is to minimize the reconstruction error. Autoencoders are often trained using backpropagation with stochastic gradient descent [17], [18].

Multiple autoencoders can be stacked together, thus yielding a deep network. This deep network consists of several hidden layers. The output of a hidden layer H becomes the input to the subsequent autoencoder on top of H. A given input is passed through this deep network, and the result is a compact high-level representation of the input.

Fig. 3 illustrates how our deep learning model works. Our deep learning model consists of two stacked denoising autoencoders (SDAs) [9], named SDA1 and SDA2. SDA1 contains four layers: 500-250-100-50. SDA2 contains eight layers: 20,000-5,000-2,500-1,000-500-250-100-50. Each numeral here represents the number of neurons in a hidden layer. The activation function used by our model is ReLU (rectified linear units). ReLU often results in faster convergence and diminishes the gradient vanishing problem, and is particularly suitable for deep neural networks [18].

Given a computer program, we first extract the function-call graph from the program. This function-call graph is then mapped to a vector in a low-dimensional feature space by using the graph embedding technique, node2vec [10]. This embedded vector has about 500 dimensions. SDA1 takes this embedded vector as input and converts it into a 50-sized vector v_1. In addition, we build the API vector for the given program. The API vector has approximately 22,000 dimensions. SDA2 takes this API vector as input and also converts it into a 50-sized vector v_2. We then concatenate v_1 and v_2 to create a 100-dimensional combined vector v. We add a softmax regression layer to our deep learning model, which classifies v to predict whether the given program is malware or not. To further regularize our model and prevent overfitting, we add a dropout layer [22] to the model. SDA1 uses a dropout rate of 0.8 and SDA2 uses a dropout rate of 0.5. In addition, we use 200 training epochs for each layer. The learning rate is set to 0.001, the batch size is set to 50, and the Adam optimizer is used in our model.

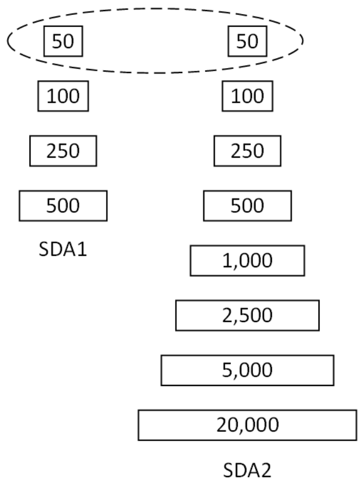

Fig. 3. The dual SDA architecture used by our approach.

IV. Experiments and Results

A. Datasets

Our datasets contained two types of samples: malicious programs and benign programs. The sources from which we collected malicious samples included the Microsoft Malware Classification Challenge [23], the VirusShare website [24], and the KafanBBS website [25]. The sources from which we collected benign samples included Windows 7 system files such as *.exe files, *.dll files and *.sys files. We submitted these collected samples to the VirusTotal [26] web service where the collected samples were scanned by 60 antivirus scanners. Samples that were detected by the scanners were labeled as "malicious" and the other samples were labeled as "benign". Among the malicious samples, we selected two families of malware: Lollipop and Kelihos-ver3. The function-call graphs of some *.dll files had only one node; these files were removed from our datasets. Overall, we collected 5,018 malicious samples and 2,177 benign samples.

Table I gives a breakdown analysis of the collected data. Based on the collected data, we created three datasets. Dataset 1 contained 2,434 Lollipop samples and 2,177 benign samples. Dataset 2 contained 2,584 Kelihos-ver3 samples and 2,177 benign samples. Dataset 3 contained all the 5,018 malicious samples, including both Lollipop and Kelihos-ver3, as well as 2,177 benign samples. In each dataset, 80 percent of the data were used for training and the other 20 percent were used for testing.

B. Performance Measure

We define a true positive to be a malicious program that is correctly classified as a malicious program, a false positive to be a benign program that is misclassified as a malicious program, a true negative to be a benign program that is

TABLE I
BREAKDOWN ANALYSIS OF COLLECTED DATA SAMPLES

Type of Samples	**Number of Samples**
Lollipop (malicious)	2,434
Kelihos-ver3 (malicious)	2,584
*.exe files (benign)	631
*.dll files (benign)	1,178
*.sys files (benign)	368

correctly classified as a benign program, and a false negative to be a malicious program that is misclassified as a benign program.

Let TP (FP, TN, FN, respectively) denote the number of true positives (false positives, true negatives, false negatives, respectively). P is the sum of TP and FN. N is the sum of TN and FP. We use Accuracy (Acc), defined in (1), to evaluate the performance of a malware detection method.

$$\mathrm{Acc} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}} \tag{1}$$

C. Experimental Results

Table II shows the accuracy values of our proposed DLGraph method and compares it with a closely related method, DL4MD [17] on the three datasets we constructed. Both DL4MD and DLGraph use programs' Windows API calls for malware detection. However, DL4MD does not consider programs' function-call graphs whereas DLGraph does. Furthermore, in [17] DL4MD only considers 9,648 Windows API calls, which are fewer than the approximately 22,000 Windows API calls considered by DLGraph here.

TABLE II
PERFORMANCE EVALUATION OF MALWARE DETECTION METHODS

Dataset	**Method**	
	DL4MD	DLGraph
Dataset 1	0.9838	0.9914
Dataset 2	0.9912	0.9936
Dataset 3	0.9875	0.9931

It can be seen from Table II that the accuracy values of our DLGraph method are high on all the three datasets studied here. Furthermore, DLGraph outperforms the existing DL4MD method. Since DLGraph considers program's function-call graphs whereas DL4MD does not, the superiority of DLGraph over DL4MD suggests effectiveness of function-call graphs for malware detection.

V. CONCLUSIONS AND FUTURE WORK

In this paper we presented a new approach (DLGraph) for malware detection using deep learning and graph embedding. Our deep learning architecture consists of two stacked denoising autoencoders (SDAs). One SDA is capable of learning a latent representation of programs' function-call graphs. The other SDA is able to learn a latent representation of programs' Windows API calls. When embedding a function-call graph in a feature space, we adopted the node2vec technique. Our experimental results on three different datasets showed that this proposed DLGraph method achieves high accuracy values, and outperforms the closely related DL4MD method.

In future work, we plan to develop more sophisticated deep learning architectures, taking into consideration malware images, program behavior, function-call graphs, and Windows API calls. To mine the malware images, we plan to use convolutional neural networks (CNNs). Thus, these new deep learning architectures will contain multiple stacked denoising autoencoders and CNNs. We plan to use these new architectures to perform not only binary classification, but also multiclass classification, on a wide range of different malicious software.

REFERENCES

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, 2012.

[2] J. Aycock, *Computer Viruses and Malware*. Springer, 2006.

[3] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Information Security*, vol. 5, pp. 56–64, 2014.

[4] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," *CoRR*, vol. abs/1710.09435, 2017.

[5] P. Beaucamps and E. Filiol, "On the possibility of practically obfuscating programs towards a unified perspective of code protection," *Journal in Computer Virology*, vol. 3, no. 1, pp. 3–21, 2007.

[6] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, 2017, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029824

[7] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security*, 2009, pp. 611–620. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653736

[8] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang, "Detecting malware variants via function-call graph similarity," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, 2010, pp. 113–120. [Online]. Available: https://doi.org/10.1109/MALWARE.2010.5665787

[9] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1953039

[10] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 855–864. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939754

[11] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques and applications," *CoRR*, vol. abs/1709.07604, 2017. [Online]. Available: http://arxiv.org/abs/1709.07604

[12] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, "*struc2vec*: Learning node representations from structural identity," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 385–394. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098061

[13] J. Heaton, "Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning - the MIT Press, 2016, 800 pp, ISBN: 0262035618," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, 2018. [Online]. Available: https://doi.org/10.1007/s10710-017-9314-z

[14] Z. Hu, T. Turki, N. Phan, and J. T. L. Wang, "A 3D atrous convolutional long short-term memory network for background subtraction," *IEEE Access*, vol. 6, pp. 43 450–43 459, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2861223

[15] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[16] D. Gibert, J. Béjar, C. Mateu, J. Planes, D. Solis, and R. Vicens, "Convolutional neural networks for classification of malware assembly code," in *Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence*, 2017, pp. 221–226.

[17] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "DL4MD: A deep learning framework for intelligent malware detection," in *Proceedings of the International Conference on Data Mining*, 2016.

[18] O. David and N. S. Netanyahu, "DeepSign: Deep learning for automatic malware signature generation and classification," in *Proceedings of the 2015 International Joint Conference on Neural Networks*, 2015, pp. 1–8.

[19] "Hex-rays. The IDA Pro disassembler and debugger," https://www.hex-rays.com/products/ida/, 2018, [Online; accessed 16-April-2018].

[20] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, 2011.

[21] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, pp. 38–49. [Online]. Available: https://doi.org/10.1109/SECPRI.2001.924286

[22] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

[23] "Microsoft Malware Classification Challenge," https://www.kaggle.com/c/malware-classification, 2018, [Online; accessed 16-April-2018].

[24] "VirusShare," https://virusshare.com/, 2018, [Online; accessed 16-April-2018].

[25] "KafanBBS," https://bbs.kafan.cn/, 2018, [Online; accessed 16-April-2018].

[26] "VirusTotal," https://www.virustotal.com/, 2018, [Online; accessed 16-April-2018].