Classifying Packed Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network

Yakang Hua
Faculty of Information Technology
Beijng University of Technology
Beijing, China
e-mail: hyk3826@gmail.com

Yuanzheng Du
Faculty of Information Technology
University of Ottawa
Beijing, China
e-mail: ydu016@gmail.com

Dongzhi He
Faculty of Information Technology
Beijng University of Technology
Beijing, China
e-mail: 2602515795@qq.com

***Abstract*—In recent years, the variety and quantity of malware has increased rapidly, which makes classification based on fixed features very difficult. In order to better preserve the integrity of the malware, we extract the control flow graph of malware as the feature information. However, control flow graph of a packed malware consists of both unpack control flow graph and local control flow graph. In order to obtain a clean control flow graph, we analysis the packed malware dynamically and apply an algorithm to strip the unpacking routine from the graph. With the rapid development of Graph Neural Network (GNN), we use Deep graph Convolutional Network (DGCNN) to classify control flow graph data. Experimental results show that the proposed method can achieve 96.4% accuracy.**

Keywords-malware classification; control flow graph; graph convolution

I. INTRODUCTION

In recent years, the malware detection methods mainly use static or dynamic methods to obtain the feature information of malware, including sequence information and graph structure information. With the rapid development of deep learning in the field of malware detection, researchers used different feature information as the input of deep learning network to detect malware. [1] proposed to extract API sequences and API return values as feature sequences during dynamic sample execution. [2] proposed to extract the operating system API and C library API as the input of call sequence during dynamic execution. [3] proposed the method of combining the API sequence extracted during sample dynamic execution with n-gram to analyze the homology of malware. [4] used the SPGK algorithm to calculate the similarity among malware function call graphs. [5] statically analyzed the system call diagram of malware and used the method of graph matching for classification. [6] used the graph editing distance to determine the similarity of malware through static analysis of abstract function features. [7] analyzed the homology of malware by calculating the similarity distance of the function call graph. [8] used graph embedding to map the control flow graphs extracted from disassembled binaries to low-dimensional vectors as inputs for two stacked denoising autoencoders (SDAs) that are responsible for representation learning.

Most researchers base their research on instruction sequences or API (Application Programming Interface) sequences, but sequence information can lose some spatial characteristics during the execution of malware. As the characteristic information of malware, control flow graph has both temporal and spatial characteristics. Therefore, we decided to use the control flow graph as the input of deep learning network for classification. If graph data structure data is directly used as the input of deep learning network, there will be two problems. (1) In order to prevent from being detected by static analysis of antivirus software, many malwares choose to use packing techniques to confuse the analyzer. The control flow graph generated by running malware in sandbox consists of both unpack control flow graph and local control flow graph. This unpack control flow graph should be removed before inputting into the classification model. (2) Different malware sample extracts different control flow graph sizes. Traditional convolutional neural network and recursive neural network can not directly deal with graph structure data. To solve these problems, we propose a new control flow graph generation method and use DGCNN[9] to detect malware. First, the dynamic analysis method is used to run the packed malware in the sandbox, record the disassembly instructions it has executed and extract the function call graph of the packed malware. Then we apply an algorithm to strip the sub-graph that represents the unpack function call in the function call graph of the packed malware. Finally, we perform the expansion operation on the sub-graph which only contains local function call graph to get control flow graph of the packed malware. The features of the control flow graph are extracted as the input of the DGCNN for

978-1-7281-5904-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICCEA50009.2020.00062

Authorized licensed use limited to: Hebei University of Technology. Downloaded on January 04,2021 at 06:08:38 UTC from IEEE Xplore. Restrictions apply.

training, and the classifier is obtained to detect the packed malware.

II. Classification Model Based On DGCNN

A. Related definition

A packed malware's function call graph is composed of unpack functions call graph and malware local functions call graph. For a packed malware P, we use G = (V, E) representing its function call graph. In the graph G, $V = \{ V_1, V_2, ..., V_n\}$ is the set of function nodes. There are 2 types of function nodes: (1) Unpack functions, decrypting malware location function in memory during execution. (2) Local functions, implementation of malicious behaviour defined by malware itself. $E = \{<V_i, V_j>|V_i, V_j \in V\}$ is the set of edges between function nodes. When $<V_i, V_j> \in E$, there exists a function call from V_i to V_j. We use $|V| = n$ to denote the size of G, and $|E| = m$ to denote the number of function calls in G. We define $A = [a_{ij}]^{n*n}$ as the adjacency matrix of the graph G such that an entry $a_{ij} = 1$ if there is a flow from V_i to V_j and 0 otherwise.

B. Stripping unpacking routine from packed malware's function call graph

During the dynamic execution of the packed malware, the unpack functions do not call the malware local functions. On the other hand, the malware local functions do not call the unpacking functions. For the unpacked malware, due to the existence of function reuse, we observed a lot of mutual functions calls between unpack functions and between local functions. Such mutual calls lead to a high connectivity between function nodes.

The distribution of non-zero values in the adjacency matrix of the function call graph corresponding to each malware can directly express the connectivity between function nodes. There is a high connectivity between unpack function nodes, but there is zero connectivity from unpack function nodes to malware local function nodes. Similarly, there exists high connectivity between malware local function nodes, but the connectivity from malware local function nodes to unpack function nodes is negligible. As shown in Figure 1, a packed malware sample has two sets of points, the set of points in the upper left corner representing the unpack functions that was called first, and the set of points in the lower right corner representing the malware local functions would be called later. As shown in Figure 2, the distribution of the point set of a unpakced malware is relatively scattered.

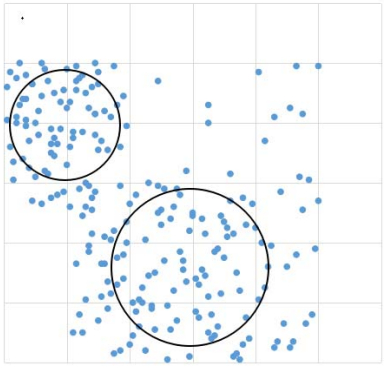

Figure 1. Schematic diagram of non-zero value distribution of the adjacency matrix of packed malware

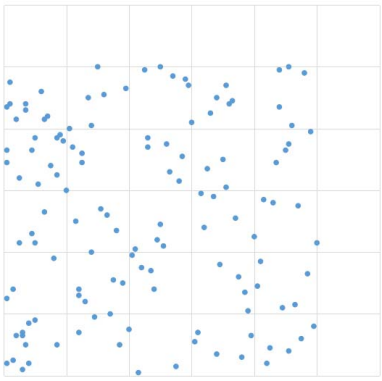

Figure 2. Schematic diagram of non-zero value distribution of the adjacency matrix of unpacked malware

The connectivity difference of the function call graph can be represented by the adjacency matrix density of the function call graph. The density of the adjacency matrix is defined as follows:

$$d(G) = \frac{m}{n*(n-1)} . \quad (1)$$

For the function call graph, there is a recursive call that causes the node to self-connect, so its density of the adjacency matrix is defined as follows:

$$d(G) = \frac{m}{n^2} . \quad (2)$$

As shown in the Figure 3, the adjacency matrix $A \in [a_{ij}]^{n*n}$ of the function call graph can be divided into four submatrices $A_0 \in [a_{ij}]^{i*(n-i)}$, $A_1 \in [a_{ij}]^{i*i}$, $A_2 \in [a_{ij}]^{(n-i)*i}$, $A_3 \in [a_{ij}]^{(n-i)*(n-i)}$. By sliding the cursor downward to the right to the vertex V_i with the diagonal as the axis and the vertex as the unit, the densities of the four matrices are $d(A_0)$, $d(A_1)$, $d(A_2)$, $d(A_3)$. Both A_0 and A_2 represent adjacency matrices for the connectivity between the unpack function and the malware local function. $d(A_0)$ and $d(A_2)$ represents the adjacency matrix density of the connectivity between the unpack function and the malware local function. A_1 represents the adjacency matrix of self-connectivity between unpack functions. $d(A_1)$ represents the adjacency matrix density of self-connectivity of unpack functions. A_3 represents

Authorized licensed use limited to: Hebei University of Technology. Downloaded on January 04,2021 at 06:08:38 UTC from IEEE Xplore. Restrictions apply.

the adjacency matrix of self-connectivity between malware local functions. $d(A_3)$ represents the adjacency matrix density of the self-connectivity of the malware local function.

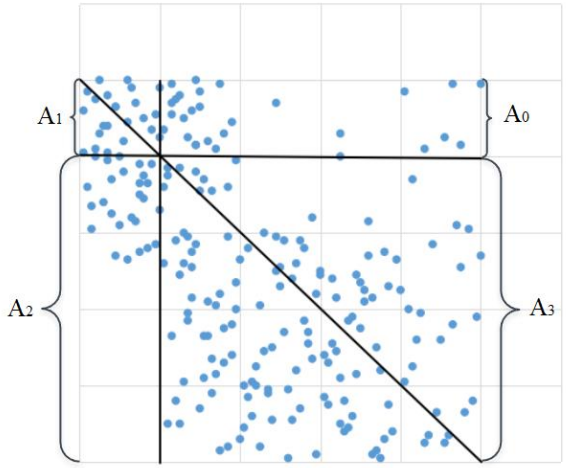

Figure 3. Schematic diagram of a packed malware adjacency matrix divided into four parts

Our goal is to find a way to minimize the sum of $d(A_0)$ and $d(A_2)$. Specifically, for a n-order adjacency matrix A with n vertices, the sliding process of the cursor is divided into three stages.

Stage 1: As shown in the Figure 4, when the cursor reaches the vertex V_i, A_0 and A_2 partially contain unpack functions.

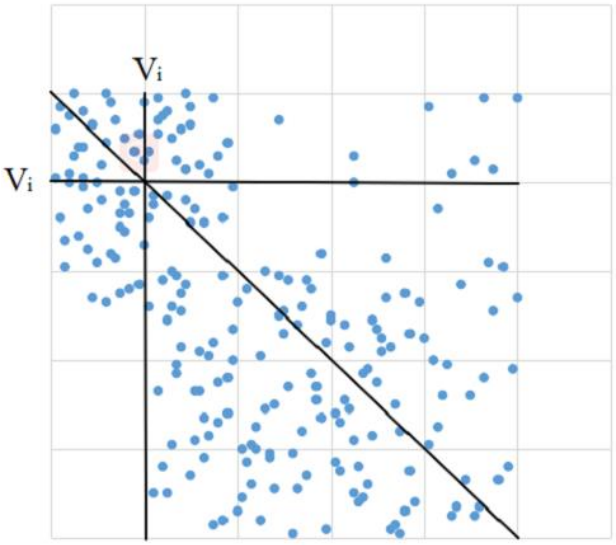

Figure 4. Schematic diagram of how the adjacency matrix is divided when cursor is at V_i

Stage 2: As shown in the Figure 5, when the cursor reaches the vertex V_j, A_0 and A_2 do not contain either unpack functions or malware local functions.

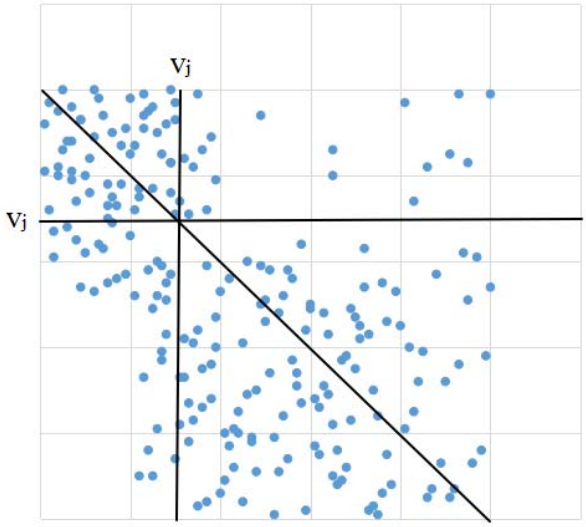

Figure 5. Schematic diagram of how the adjacency matrix is divided when cursor is at V_j

Stage 3: As shown in the Figure 6, when the cursor reaches the vertex V_k, A_0 and A_2 partially contain malware local functions.

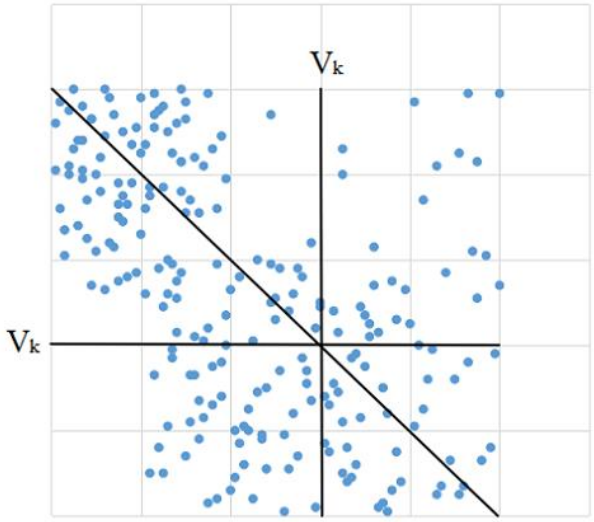

Figure 6. Schematic diagram of how the adjacency matrix is divided when cursor is at V_k

When the cursor reaches V_{n-1}, the sliding progress ends. The adjacency matrix $A \in [a_{ij}]^{n*n}$ results in a total of n-1 segmentation methods. We look for one of these segmentation methods that minimizes the sum of $d(A_0)$ and $d(A_2)$. At this moment, A_3 only contains the malware local functions and their call relationship. Therefore, when the function call graph is

Authorized licensed use limited to: Hebei University of Technology. Downloaded on January 04,2021 at 06:08:38 UTC from IEEE Xplore. Restrictions apply.

expanded, we only expand the adjacency matrix A_3 which representing the malware local function call graph to obtain the control flow graph.

C. Malware Control Flow Graph Generation

We get the control flow graph of the malware by expanding the malware local function call graph. During the expansion process, each node of the local function call graph is replaced with the control flow graph for each local function, and is connected by adding a directed edge according to its call relationship.

For example, when the base block V contains a CALL instruction that calls a malware local function, remove the CALL instruction and divide the base block V into V_{up} and V_{down}. Add 2 directed edges. The first edge starts with V_{up}, it then points to the block which containing the first instruction of the function called by block V. The second edge starts with the block containing the return instruction of the function call by block V, and then points to V_{down}. The connection method is shown in Figure 6.

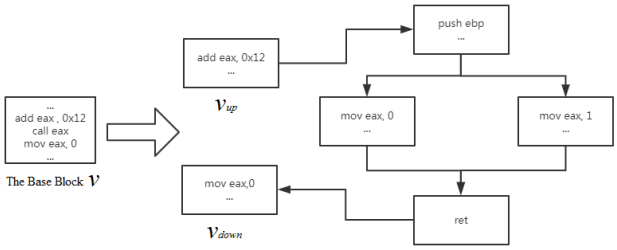

Figure 7. Basic block extension

We apply this expansion method to all function nodes in the function call graph to generate the malware control flow graph.

D. Classification Model

After the control flow graph is built, the malware is now represented as a set of connected base blocks, with each vertex representing a base block. To convert the control flow graph into input suitable for deep learning, we define properties at each vertex and summarize code characteristics as values. The characteristics of each base block are extracted as attributes of the node. The base block represented by each node has two attributes: (1) The maximum machine code instruction length of each base block. (2) The total number of machine code instructions per base block.

Since the graph is generally a spatial structure with variable length, the translation invariance of ordinary convolutional neural network is not applicable to non-matrix structured data, so the graph structure data cannot be directly processed. To solve this problem, DGCNN is used to classify malware families represented by control flow graph. DGCNN is a state-of-the-art neural network architecture that can directly accept graphs of arbitrary structures to learn a graph classification function.

As shown in the Figure 8, the standard DGCNNs have three sequential steps : 1) Graph convolutional layers generalize the convolution operation from Euclidean domains or grid-like structures such as image data to non-Euclidean domains such as graph data by generating node representations as the aggregation of their own feature descriptors and their neighbors feature descriptors. 2) A SortPooling layer sorts the unordered graph data according to their feature descriptors or structural roles. This step guarantees that the nodes of different graphs will be placed in similar positions, according to their weighted feature descriptors. 3) The ordered graph data is flattened and passed to a standard 1-dimensional CNN layer followed by a fully connected layer to learn a classification function.

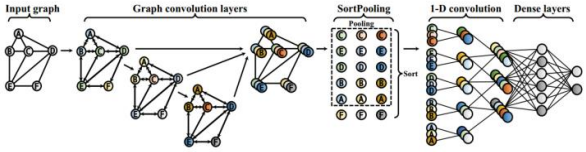

Figure 8. The overall structure of DGCNN

III. EXPERIMENT AND ANALYSIS

A. Datasets

The experimental data set in this paper was obtained through VirusShare website. This experiment chose 6 malware families, 100 samples for each family. Table 1 is the family name of experimental samples.

TABLE I. THE CHOSEN MALWARE FAMILIES

Serial number	Family name
1	Gh0stRAT
2	Nitol
3	Formbook
4	TianFaDDos
5	MyKings
6	Emotet

B. Performance Measure

We use Accuracy (Acc) to evaluate the performance of a malware classification method. The Acc is defined as follows:

$$\mathrm{Acc} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}}, \tag{3}$$

for example classes C1 and C2: TP represents the number of C1 classified as C1. FP represents the number of C2 classified as C1. FN represents the number C1 classified as C2. TN represents the number C2 classified as C2.

C. Experimental Results

In order to investigate the generalization of the classifier, we used 10-fold cross validation method. The stratified k-fold cross-validation ensures that each training set split contains a similar proportion of positive and negative samples. For a 10-fold cross-validation, the dataset is partitioned into ten different partitions. Then, the model is trained over nine partitions and tested on the remaining partition. This process is repeated ten times, each time considering a different set of folds, and the average result is reported.

Authorized licensed use limited to: Hebei University of Technology. Downloaded on January 04,2021 at 06:08:38 UTC from IEEE Xplore. Restrictions apply.

The control flow graph was extracted from the sample of the malware sample data set, and the accuracy rate was 96.4% after the training of the DGCNN.

IV. Conclusion

With the rapid development of Graph Neural Network, we use the initial DGCNN in this paper to solve the problem of input model of graphs of different sizes. However, there are other characteristics of control flow graph, which need to be tested using different neural network models. Next, we will try more neural network models more suitable for control flow graph detection through experiments to further improve the detection effect of malicious samples.

References

[1] Tobiyama S, Yamaguchi Y, Shimada H, et al. Malware Detection with Deep Neural Network Using Process Behavior[C]//Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual. IEEE, 2016, 2: 577-582.

[2] Pascanu R, Stokes J W, Sanossian H, et al. Malware classification with recurrent networks[C]//Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 2015: 1916-1920.

[3] LEE T, CHOI B, SHIN Y, et al. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient[J]. The Journal of Supercomputing, 2015: 1-15.

[4] Searles R, Xu L, Killian W, et al. Parallelization of machine learning applied to call graphs of binaries for malware detection [C7//Proceedings of the 2017 25th Euromicro International Conference on Paral lel. Distributed and Network-based Processing (PDP). Saint Petersburg: IEEE, 2017.

[5] KINABLE J, KOSTAKIS O. Malware classification based on call graph clustering [J]. Journal of Computer Virology and Hacking Techniques, 2011, 7(4):233-245.

[6] YANG E. ZHANG F G FU J M, et al. Malware Detection Based or Graph Edit Distance[J]. Wuhan: Wuhan Univ (Nat Sci Ed.) , 201359(5): 453-457.

[7] LIU X, TANG Y. Similarity analysis of malware's function-call graphs[J]. Computer Engineering & Science, 2014, 36(3):481-486.

[8] Haodi Jiang, Turki Turki, and Jason TL Wang. “DLGraph: Malware Detection Using Deep Learning and Graph Embedding”. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE. 2018, pp. 1029 1033

[9] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An End-to-End Deep Learning Architecture for Graph Classification," in Proceedings of AAAI Conference on Artificial Inteligence. 2018.

Authorized licensed use limited to: Hebei University of Technology. Downloaded on January 04,2021 at 06:08:38 UTC from IEEE Xplore. Restrictions apply.