Multi-edge directional heterogeneous graph representation of malware behavior for autonomous malware detection

Tu Minh Nguyen[1], Hung Viet Nguyen[1]
[1]Le Quy Don Technical University, Hanoi
Email: tunguyenhsgs@gmail.com, hungnv@mta.edu.vn

***Abstract* – Detecting malware using dynamic analysis techniques is an efficient method. Traditional techniques, such as signature-based detection, perform poorly when attempting to identify zero-day malwares, and it is also a tedious task to manually engineer malicious behaviors. There are several studies trying to automate this process. One of effective approaches introduced in recent years is to use graphs to represent the behavior of an executable, and learn from these graphs. However, the graphs constructed are simple and omit many important pieces of information. In this paper, we present a method to represent malware as a multi-edge directional heterogeneous graph, based on behavior collected from cuckoo sandbox, and apply a graph attention network to detect malware from the collected datasets. The experiments show that our model archived the better result than other machine learning models in both TPR and FAR score.**

Keywords – malware detection, malware dynamic analysis, deep learning, heterogeneous graph, graph representation.

I. INTRODUCTION

Malware is referred to as "any software that does something that causes harm to a user, computer, or network" [11]. Detecting malware remains a significant security challenge, predominantly due to its continued increase in sophistication. Techniques used to analyze malware are categorized in two types: static analysis and dynamic analysis. The former including signature-based is considered a simple and lightweight approach. However, malware samples that apply obfuscation techniques such as refactoring code, inserting nop-code, encryption etc. can easily bypass static analysis. The latter includes two types of behavioral data, static behavior data (or code analysis) and dynamic behavior data (or behavioral analysis). Code analysis collects data by static methods such as reverse-engineering and can give us a sight on what the software does. However, it faces the same problem of being evaded by obstruction techniques such as binary packers, polymorphism, metamorphism or anti-debugging etc. Hence, behavioral analysis becomes attractive to analysts because it can tackles it from a black box perspective, whereby only the end result on the system can be observed.. This method requires emulating a safe virtual environment and executing the malware inside it to monitor its behavior. Although there are tactics to prevent behavioral analysis, this strategy is less vulnerable to obfuscation techniques.

Manually analyzing an executable to identify malicious behavior is a highly laborious process, therefore many recent research project have focused on automating this process. Devised techniques range from extracting features using text-mining algorithms, to learning features from graphs that represent behaviors of executable files. These approaches are very inspiring and have proved their efficiency in existing literature. However, behavioral obfuscation techniques (e.g. system call reordering or bogus call injection) pose a challenge to approaches that represent behavioral data in sequences. One major limitation of current graph representation methods is that they present an abstract view of system behavior and omit important information.

This paper presents a novel approach to address this problem by building a graph that can represent multiple types of information, including API calls, connection types, and key arguments of each API. After the graph is built, malware detection is performed by using a neural network to learn the node-level embedding (which can be derived by weighted-aggregating one's neighbor nodes after weighing each edge connected between two nodes) and semantic-level embedding (which can be determined by the weighted-sum embedding of every connection type). Our main contributions in this work are as follow:

- Propose a new method to represent behavioral data (as heterogeneous graph),
- Propose several approaches of encoding data,
- Propose new model to learn features from built graphs (based on graph attention network), introducing edge-weighing layer, along with data encoding techniques, to focus on the argument of each API to weigh the importance of that call,
- Comparison with other models,
- Experiment on new constructed dataset.

The rest of the paper is organized as follows: Section II presents an overview of existing research focused on detecting malicious software automatically. Section III provides a detailed description of our proposed approach. In Section IV, experimental results are discussed. Finally, conclusions are drawn in Section V.

II. RELATED WORK

In response to the steadily increasing complexity of modern malware, much research has been conducted to find alternative malware detection strategies. One efficient way is to analyze the behavior of the software after executing it in a virtual safe environment. Many studies rely on system call traces to evaluate and identify the malicious behaviors of malware samples.

Almost all proposed methods need to depend upon behavioral data (e.g., for example API calls), which to maximize accuracy must be performed in a specific way. The difficulties of this task lie in how to represent this behavioral data efficiently, whilst reducing noise without losing any useful information. In terms of API calls being used to analyze and learn features from, there are two popular approaches used to represent such data: sequences of text, the other uses graph.

With text-based representation, features can be extracted by applying conventional algorithms, or using deep learning model such as Recurrent Neural Networks (RNN) or Convolutional Neural Networks (CNN) to extract features automatically, and a classifier would then be applied to learn from features extracted. Yu eta al. [1] gave an overview of behavioral description methods including XML-based, semantic description methods, description languages and several text-based. Hongfa et al. [2] represented system call sequences with MIST instructions and used an n-gram algorithm to extract features. In [3] Zhao et al. proposed the use of a control flow graph to generate an execution tree and form an opcode stream. N-gram is also used to generate feature set afterwards. Sequence alignment algorithms was used in [4] for common call sequence extraction. However, the complexity of sequence alignment algorithms was too large and computing time was too high. Based on NLP techniques, Tran et al. in [5] enhanced the conventional ML algorithms for API calls analysis by doc2vec, N-gram and TF-IDF methods. The n-gram analysis method archived some good results, but it faced to the optimizing the values of n and L. The current pace of malware development requires models that can seek patterns and informative features autonomously. Pascanu et al. in [6] were the first to use a hybrid model of RNN and an machine learning classifier to predict the next API call. Kolosnjaji et al. in [7] proposed a method to detect and classify malwares in series of opcodes representation, using a Convolutional Neural Network (CNN) and feed-forward layers. This model used static analysis of portable executable files so hard to detect malwares with obfuscation and detection evasion techniques. RNN and LSTM are also experimented with in various existing works but largely face the same problems [8][9][10].

In recent years, graph neural network has been a trend in related literature that has proven to be an effective format for representing linked data and extracting features. Inspired by this approach, many studies have attempted to present behavioral data in graph form. Authors of [19] generated Markov chain graphs from dynamic trace data, and applied graph kernels to acquire similarity matrix, which was sent to a Support Vector Machine (SVM). Naval et al. [12] extracted system call traces by monitoring malware execution and transforming the traces into Ordered System-Call Graphs (OSCGs). Another common type of graph that is used frequently in visualizing malware behavior data is Quantitative Data Flow Graph (QDFG) as introduced by Wüchner et al. [14], however, this work only formalizes heuristics to identify malware. Work by Hung et al. [15] outline an extended version of the traditional QDFG by subsequently applying a graph neural network (GCN). Although this graph succeeded in expressing more informative data, it still lack some details, for example each entity is only identified by its type (i.e. process, file, registry, network) but does not contain any more data such as its name, path or arguments etc.

It can be inferred that behavioral data contains different types of information, including different API categories, different objects and resources that the software influences. Therefore, this signifies that heterogeneous graph would be a suitable format in which to illustrate behavioral data. Currently, there is very little existing work investigating the use of heterogeneous graphs, and we believe our work is the first to represent behavioral data as a heterogeneous graph. Further details are outlined in Section III.

III. PROPOSED METHOD

In our paper, we use the dynamic behavioral data, or more precisely, the API calls collected from Cuckoo sandbox to construct multi-edge directional graphs. To generate graph embedding and classify malicious and benign samples, we apply an attention neural network, as inspired by the work of Wang et al. [16].

A. Graph representation

1) Entities and connections

Our graph contains 6 main types of entities or nodes: Process, File, Registry, and three for 3 types of API calls: ProcessAPI, FileAPI, RegistryAPI. There are 5 types of connection accordingly:

- Process-ProcessAPI performs connection between a process handle (process entity) to a Process API (an API that belongs to process category),
- File-FileAPI performs connection between a file handle (file entity) and a File-API,
- Registry-RegistryAPI performs connection between a registry handle (registry entity) and a RegistryAPI,
- Process-FileAPI performs connection between a process entity and a FileAPI,
- Process-RegistryAPI performs connection between a process entity and a RegistryAPI,
- Self-loop: for each node to have its own features taken into consideration.

Note that there would never be a connection between a file handle and a registry API or a registry handle and a file API. The detail of how the graphs are constructed are presented below.

These entities and connections are built on API calls that belong to 3 categories respectively: process, file and registry. Inherited from the work of Wüchner [14], processes, files, sockets, and registry keys are of much significance when identifying malicious actions. Notice that there is no restriction on the number of types of entities or nodes in our graph, the reason we limit the types of nodes to 3 is due the limitations in data collection. Therefore, with more types of nodes and edges the graph needs to represent, feature space would become bigger, and the data would be inadequate for learning in such a huge feature space. The edge data would be important arguments of each call. Note that for each API call node (entity), only the name of the API is used to encode as features, all arguments are placed in edge data. Therefore, there might be multiple connections to one single API call node. The graph is also directional. The principles to determine the direction of each connection is similar to the work of Hung et al. in [15], where all API calls that perform opening, creating, writing, or any modifying actions towards a file or registry would be the source nodes, and the destination nodes are the file or registry themselves . In other cases, this will be reversed (i.e.the file or registry are the source node and the API calls are destination node). Below is an example of the behavior from a malware sample collected from a cuckoo report.

List I. An example of a behavior generated by cuckoo

```
{
   "category": "process",
   "status": 1,
   "stacktrace": [],
   "api": "WriteProcessMemory",
   "return value": 1,
   "arguments": {
      "process identifier": 768,
      "buffer":
   "MZnu0090nu0000nu0003nu0000nu0000...",
      "process handle": "0x0000007c",
      "base address": "0x01000000"
   },
   "time": 1556629733.164881,
   "tid": 3812,
   "flags":{}
},
```

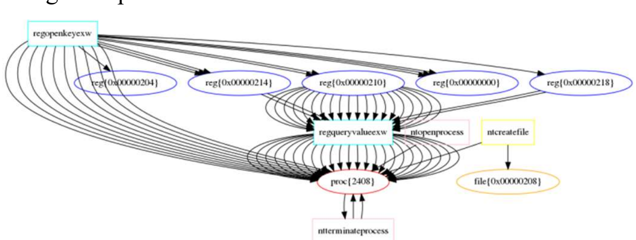

This small piece of behavior constructs 2 nodes, one is a process entity (id 768) and one is a ProcessAPI entity (`WriteProcessMemory`) (because this API belongs to category process). One edge from the API entity to the process entity, with features are generated by encoding some information. The data used for acquiring features of each node is either its API name (if it is an API entity) or the name of its type (if it is a process/file/registry entity). And for each connection, its data is obtained from the flags fields of the API that the connection links to (or from). Flags fields are generated by cuckoo giving an insight into important information about that call.

Note that in our paper, we define meta-path quite differently from the original work of Wang et al [xx]. We do not define connections between two nodes of the same type (for example movie-movie) through a middle node of different types (for example movie-actor-movie and movie-director-movie), but define a type for the connection between two nodes directly (for example Process-ProcessAPI through Process-ProcessAPI edge, or we could write Process-(Process-ProcessAPI)-ProcessAPI). After all, the importance of a heterogeneous graph is the heterogeneity of nodes and edges the graph can support..

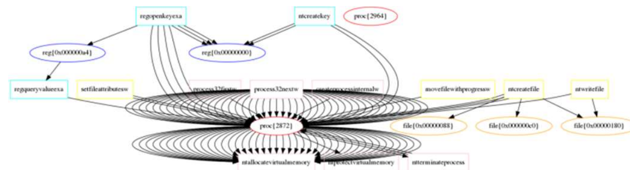
2) Embedding entities and edges arguments

As mentioned earlier, insufficient data is a big problem and has a great effect on how graphs are constructed, or more precisely what text data should be used for encoding, and how it should be encoded. In this paper, we tested on both skip-gram and TF-IDF encoding for node names (API names) and edge arguments (flags fields of an API).

Figure 1 shows a representation of a malware and a benign sample.



a) Graph representation of a benign sample



b) Graph representation of a malware sample

Fig 1. Graph representation of a malware and benign sample

B. Malware detection

In order to detect malware from a constructed graph, it becomes a graph classification task. There are two main approaches for this task: graph embedding (tries to find representations of graph nodes and edges) and graph feature extraction (e.g. using graph convolutional neural network). A recent survey in [17] in [17] classified graph deep learning models into 5 types: graph convolution, graph attention, graph generative, graph spatial-temporal networks and graph auto-encoders. [18] examined graph models in terms of the main characteristics:: graph types, training methods and propagation types; convolution and attention networks are both demonstrated to have contributed to the propagation step. [15] used GCN to extract features from graph butthis spectral approach requires training and detection on a specific graph structure, since the learned filters depend on the Laplacian eigenbasis [18]. Graph attention network, instead of statically normalizing the sum of the features using convolution operation like GCN, uses attention mechanism for weighting neighbors features with feature dependent and structure-free normalization.

For heterogeneous graph, the most distinctive feature is the heterogeneity, where each type of connection or each type of node would have a different importance in the overall consideration. [16] proposed the heterogeneous graph attention network (HAN) which utilizes node-level and semantic-level attentions and the model has the ability to consider node and meta-path importance simultaneously.

Our approach is inspired by the idea of Wang [16], our main contribution is the edge-weighing layers that can learn the importance of each connection among set of connections between two nodes, since our graphs is multi-edge. More concretely, the pipeline is as follows:

Table I. NOTATIONS AND EXPLANATIONS

Notation	Explanation
e_{ijp}	Importance of node j to node i through path p
$\alpha^{\emptyset}_{ijp}$	Weight of node pair (i,j) through path p
$\mathcal{P}_{ij}$	Set of connections between node pair (i,j) (from node j to node i)
u_p^{ij}	Importance of path p in the set of connections from node j to node i
U_{ij}	Edge-level attention vector from node j to node i
l_p	Initial edge features
l'_p	Weighted edge features
γ_p^{ij}	Weight of path p
τ_i	Weight of final node i
$\mathbf{Z}$	Graph embedding

The notations are inherited from the work of Wang, therefore the table above explains only new symbols.

1) Edge-weighing

In literature [16], embedding $\mathrm{z}_i^{\emptyset}$ of node i is computed by weighted-aggregation of the embedding of its meta-path based neighbors:

$$z_i^{\emptyset} = \sigma\left(\sum_{j\in\mathcal{N}_i^{\emptyset}} \alpha_{ij}^{\emptyset}.h_j\right) \tag{1}$$

$$\alpha_{ij}^{\emptyset} = \frac{\exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||h_j]\right)\right)}{\sum_{k\in\mathcal{N}_i^{\emptyset}} \exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||h_k]\right)\right)} \tag{2}$$

However, in our problem, each edge has features. Therefore, the importance of node j to node i should be deduced not only from the embedding of node j and node i, but alse from the connection between these two nodes. Intuitively, we would concatenate the features of edge p (between node j and node i) and calculate e_{ijp} (the importance of node j to node i through path p).

The equation (2) would then become:

$$\alpha_{ij}^{\emptyset} = \frac{\exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||l_{ij}||h_j]\right)\right)}{\sum_{k\in\mathcal{N}_i^{\emptyset}} \exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||l_{ik}||h_k]\right)\right)} \tag{3}$$

This is the case when there is only one connection between node j and node i, l_p therefore is l_{ij}. However, graphs in our problem are multi-edge, which means there could be multiple connections between two nodes. For example, Figure 2 exemplifies multiple calls to `RegQueryValueExW` but with different arguments, therefore it should have different importance values.

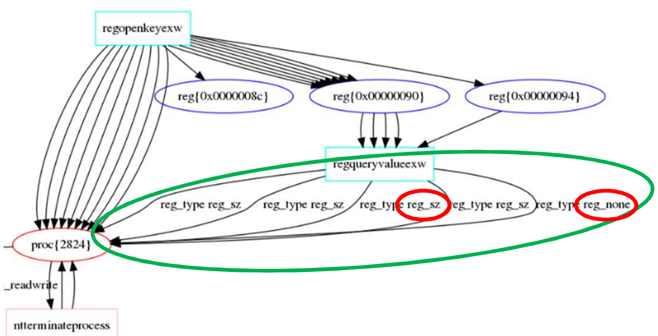


Fig 2. Same API (`RegQueryValueExW`) is called from process id 2824 with different arguments

Although we can still concatenate l_p and h_j as in equation (3), to acquire:

$$\alpha_{ijp}^{\emptyset} = \frac{\exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||l_p||h_j]\right)\right)}{\sum_{k\in\mathcal{N}_i^{\emptyset}} \sum_{m\in\mathcal{P}_{ik}} \exp\left(\sigma\left(a_{\emptyset}^{T}.[h_i||l_m||h_k]\right)\right)} \tag{4}$$

This concatenation still enables the model to learn the importance of node j to node i through path p, but note that this concatenation makes the graph become a uni-edge graph where node i has m connections to m other nodes (having features $h_j||l_p$; $p \in m$) instead of m connections to one node (having features h_j). However the purpose of building a multi-edge graph is that we expect the model could learn the importance of each edge in the set of connections between two nodes, or in other words, focus more on the edge arguments to learn the importance.

Inspired by the idea of the attention network, we use one more attention layer to learn the importance of each edge in one set of connections:

$$\begin{aligned} u_p^{ij} &= att_p(l_p) \\ &= \sigma(U_{ij}^{T}.l_p + b) \end{aligned} \tag{5}$$

The weight coefficient of path p is the softmax of u:

$$\begin{aligned} \gamma_p^{ij} &= \text{softmax}(u_p^{ij}) \\ &= \frac{\exp\left(\sigma\left(u_{ij}^{T}.l_p + b\right)\right)}{\sum_{m\in\mathcal{P}_{ij}} \exp\left(\sigma\left(u_{ij}^{T}.l_m + b\right)\right)} \end{aligned} \tag{6}$$

And the weighted embedding of path p:

$$l'_p = \gamma_p^{ij}.l_p \tag{7}$$

2) Node-level embedding

By replacing l_p in equation (3) with l'_p in equation (7) we calculate the importance of node j to node i though path p:

$$\alpha_{ijp}^{\emptyset} = \text{softmax}\left(\sigma\left(a_{\emptyset}^{T}.[h_i||l'_p||h_j]\right)\right) \tag{8}$$

And the meta-path based embedding of node i:

$$z_i^{\emptyset} = \sigma\left(Q^{T}.\sum_{k\in\mathcal{N}_i^{\emptyset}} \sum_{m\in\mathcal{P}_{ik}} \alpha_{ijp}^{\emptyset}.[h_i||l'_p]\right) \tag{9}$$

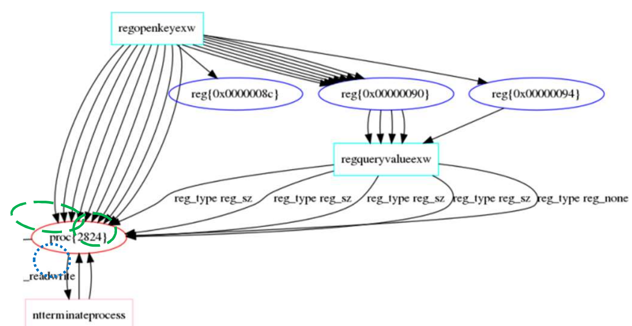


Fig 3. Aggregation of meta-path based neighbors

3) Semantic-level embedding

After having the node-level embedding, an attention network is used for learning semantic meaning:

$$\begin{aligned} w_{\emptyset_i} &= \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} q^{T}.\sigma(W.z_i^{\emptyset} + b) \\ \beta_{\emptyset_i} &= \text{softmax}(w_{\emptyset_i}) \end{aligned} \tag{10}$$

And the final embedding of node i:

$$Z_i = \sum_{k=1}^{\mathbb{P}} \beta_{\emptyset_k}.z_i^{\emptyset_k} \tag{11}$$

4) Graph embedding

After having computed the embedding for the nodes, there are a variety of ways to obtain the graph embedding. In this work, the final graph embedding is obtained by accumulating the weighted final node embedding:

$$\mathbf{Z} = \sum_{i \in \mathcal{V}} \tau_i . Z_i \quad (12)$$

IV. EXPERIMENTS

A. Datasets

In order to practically demonstrate the advancement made by this work, we will be using two datasets. The original is directly derived from the work in [15]. The second is an enhanced version created for this work, which follows the same methodology described in [15]. The exact compositions are shown in Tables I and II.

We use the *train/test subset* for training and testing, which is the same as in [15]. This dataset includes 1088 samples in total and is composed of 655 malware and 433 benign samples. Training and testing files are exactly the same as in [15]. The same *unknown subset*, which includes 637 malware samples that ClamAV was unable to detect until 2/6/2019, is tested as well. These two subsets are also used for comparison. We also collect more data using the same strategy as described in [15] for further experiments and enhancing training set. The *benign_555 subset* consists of 555 benign files, and the *pack1 subset* comprises 4620 malware samples. These two subsets, which do not contain any sample from the Original Dataset, are used for testing purpose only.

The Original Dataset composition is shown in Table I, it is important to note, that none of the samples are duplicated in any subset.

Table I. ORIGINAL DATASET COMPOSITION

Subset	Total	Malware	Benign
train/test	1088	655	433
train	761	463	298
test	327	192	135
unknown	637	637	0

The Enhanced Dataset composition is show in Table II. It is larger than the Original Dataset but it is important to note that the same methodology and same 7:3 training:test ratio was observed.

Table II. ENHANCED DATASET COMPOSITION

Subset	Total Samples	No. Malicious Samples	No. Benign Samples	Purpose
enhanced train/test	2379	1391	998	-
train	1665	954	711	Training
test	714	437	227	Testing
unknown	637	637	0	Testing
pack1	4620	4620	0	Testing
benign_555	555	0	555	Testing

The *enhanced train/test subset* consists of 2379 items, 987 of which are benign, the rest are malware. This enhanced subset is made up by combining the original *train/test subset* (described in Table I), *benign_555 subset*, (555 benign samples), and additional 741 malware samples, which are not previously seen in any set (original *train/test*, *unknown* or *pack1 subset*). The train/test ratio is 7:3, the same as that of the *train/test subset* in the Original Dataset. The experiment schemes are as follow:

- Evaluate on Original Dataset. We train on train set of *train/test subset* and experiment on test set (of *train/test subset*) and *unknown subset*.
- Evaluate on Enhanced Dataset. We train on train set of *enhanced train/test subset* and experiment on test set (of *enhanced train/test subset*), *unknown*, *pack1* and *benign_555 subset*.

B. Results

For the evaluation we utilized two types of encoding for node names and edges arguments: skip-gram and TF-IDF. For nodes names, since we only consider 3 types of API to construct nodes, the vocabulary size for node names is not relatively small. It contains 31 words, 28 of which are APIs (from the three considered categories), the 4 remain words are: proc (for process entities), file (for file entities), reg (for registry entities), and other (just in case a non-standard entry occurs in the dataset, though this would be rare). The vocabulary size for edge arguments is bigger, containing 138 words, one for each of the 137 case covered, and a "null" entry for potentially unseen words.

When using TF-IDF encoding, we use three max elements and one second-min element to construct a 4-dimensional feature vector of each edge. For skip-gram encoding, input is the whole argument string sequence and the output is a 10-dimensional feature vector. Table III shows evaluation of different model (trained on *train/test subset* in Original Dataset) with different ways of encoding node and edge data on the Original Dataset. It can be seen that using edge-weighing gives the best performance on *train/test subset*. And using edge-weighing layers outperformed original GAT model for heterogeneous graph proposed by Wang et al.

Table III. RESULTS DIFFERENT TEXT ENCODING AND MODEL ON THE ORIGINAL DATASET

	Acc	TPR	FAR	Acc	TPR	FAR	Acc	TPR	FAR
	Train/test (1088)						Unknown (637 malware)		
	Train (761) 298 benign 463 malware			Test (327) 135 benign 192 malware					
Skip-gram + TF-IDF (1st)	**96.19%**	**96.98%**	**5.03%**	**92.66%**	**92.19%**	**6.67%**	--	89.64%	--
Skip-gram (2nd)	93.82%	95.90%	9.40%	88.69%	89.06%	11.85%	--	**96.55%**	--
TF-IDF (3rd)	90.41%	92.44%	12.75%	91.74%	**92.19%**	8.89%	--	96.23%	--
Skip-gram (no edge-weighing) (4th)	88.04%	86.39%	10.07%	85.63%	83.33%	11.11%	--	85.22%	--
TF-IDF (no edge-weighing) (5th)	80.81%	79.05%	16.44%	84.40%	80.21%	9.63%	--	84.46%	--

The Original Dataset is the same as literature [15], therefore we conduct a comparison between our best model (using skip gram encoding for node names and TF-IDF encoding for edge arguments) and others on this dataset. Table IV and V show comparison results on two subsets: test set from *train/test subset* and *unknown subset*. The results of other methods are inherited from literature [15]. Our model outperformed in both cases.

Table IV. EVALUATION RESULT COMPARISON OF OUR MODEL

	Acc	TPR	FAR
Our model (1st model)	**92.66%**	**92.19%**	**6.67%**
MalGCN	86.22%	88.02%	9.66%
QDFG-GCN	74.31%	87.05%	44.04%
QDFG-KNN	62.37%	49.59%	15.49%

Table V. COMPARISON OF OUR MODEL AND OTHERS ON UNKNOWN SUBSET

Engine	**Accuracy**	**Engine**	**Accuracy**
Our model (1st model)	**89.64%**	K7AntiVirus	73.95%
MalGCN	84.03%	Invincea	73.43%
McAfee-GW631	82.59%	CrowdStrike	72.38%
Fortinet	82.59%	Sophos	70.29%
Microsoft	78.93%	AVG	69.63%
MccAfee	77.75%	GData	69.24%
ESET-NOD32	77.75%	Rising	68.06%
K7GW	74.21%	Avira	67.54%
Endgame	74.08%	VBA32	67.28%

We have implemented a simple classifier on embedding sequences using these two encoders to investigate the performance of each encoding method, the results from this are shown in Table VII. It is noticeable that the classification performance on the TF-IDF encoded data is quite poor on *benign_555*. Additionally, encoding node data using skip-gram for *benign_555* results in an even worse performance. This is because sequences of nodes names only, do not convey much meaning, in a sense that there is not much difference between the sequences of API called by benign and malware samples. As mentioned in Section II, differences usually lie within the arguments of each call. Also, note that this is just to help us understand how the encoding method may affect our model, hence we just simply apply a classifier on encoded sequences of API called (ordered by the appearance of that call in the report generated by cuckoo). TF-IDF on the other hand considers the frequency of separate words, and the way words are chosen from each sequence is the same in every circumstance, (3 max and 1 second-min elements), therefore can detect from an early stage which calls seem to be abnormal.

Table VI. CLASSIFYING BASED ON ENCODING EDGE ARGUMENTS AND NODES NAMES ONLY

	Train/test (1088)				Unknown (637 malware)		Pack1 (4620 malware)		Benign_555 (555 benign)	
	Train (761) 298 benign 463 malware		Test (327) 135 benign 192 malware							
	Edge	Node	Edge	Node	Edge	Node	Edge	Node	Edge	Node
Skip-gram	77.1%	82.3%	78.9%	75.8%	90.9%	97.5%	74.1%	59.3%	14.3%	0.0%
TF-IDF	98.2%	97.4%	87.3%	90.9%	81.3%	92.8%	64.2%	51.7%	26.4%	73.2%

When we experiment these models (trained on *train/test subset* from Original Dataset) on *pack1* and *benign_555* subsets, all models still give high True Positive Rate (TPR) on the *pack1 subset*, however, they achieve worse False Alarm Rate (FAR) on the *benign_555 subset*. The results are shown in Table VII. It can also be inferred from this table that combining skip-gram and TF-IDF encoder (using skip-gram for encoding the edge arguments and TF-IDF for encoding the nodes names) gives more promising results, and is superior in stability as well.

Table VII. RESULTS ON PACK1 AND BENIGN_355 SUBSET

	Acc	TPR	FAR	Acc	TPR	FAR
	Pack1 (4620 malware)			Benign_555 (555 benign)		
Skip-gram + TF-IDF (1st)	--	81.28%	--	--	--	21.29%
Skip-gram (2nd)	--	**95.30%**	--	--	--	29.53%
TF-IDF (3rd)	--	82.42%	--	--	--	21.77%
Skip-gram (no edge-weighing) (4th)	--	59.65%	--	--	--	26.24%
TF-IDF (no edge-weighing) (5th)	--	49.65%	--	--	--	**16.25%**

However, all experiments result in higher FAR on *benign_555* when compared with *train/test subset*. This might be due to the difference in DLL usage between the *benign_555* and *train/test* subsets. More specially, *benign_555* samples all require external DLLs loaded to be able to execute, whereas none of the samples in *train/test subset* require any DLLs, because all benign samples in *train/test subset* are Windows system files.

Therefore, we train and evaluate our model (the 1st model) on the Enhanced Dataset. The results are shown in Table VII.

Table VIII. EVALUATION WHEN TRAINING WITH ENHANCED DATASET

Dataset/testset		**Acc**	**TPR**	**FAR**
Enhanced train/test	Train	96.22%	96.02%	3.52%
	Test	93.00%	92.68%	6.45%
Unknown		--	88.23%	--
Pack1		--	90.77%	--

It can be seen from Table VII and V that our model (when trained on *enhanced train/test subset*) cause a slight decline in TPR (or accuracy) on *unknown subset*, however, it is still a better result than other models.

C. Discussions

1) Edge-weighing

For a more intuitive evaluation and deeper understanding, we have visualized the weights of each edge produced by our model. List II shows an example of a signature for malicious activity of a malware sample. The signature is generated along with cuckoo report by applying YARA rules which are contributed by the open community.

The call to `NtAllocateVirtualMemory` API is indicated as malicious when it requires not only read, write but also execute permissions, and its allocation type is

MEM_COMMIT and MEM_RESERVE. The graph of this malware after edge-weighing layers is illustrated in figure 4.

List II. A signature for malicious activity according to yara rules


```
{
  "markcount": 2,
  "families": [],
  "description": "Allocates execute permission to another process indicative of possible code injection",
  "severity": 3,
  "marks": [
    {
      "call": {
        "category": "process",
        "status": 1,
        "stacktrace": [],
        "api": "NtAllocateVirtualMemory",
        "return_value": 0,
        "arguments": {
          "process_identifier": 2508,
          "region_size": 36864,
          […]
        },
        "time": 1556598469.154953,
        "tid": 2468,
        "flags": {
          "protection": "PAGE_EXECUTE_READWRITE",
          "allocation_type": "MEM_COMMIT|MEM_RESERVE"
        }
      },
      […]
    },
    ...
  ]
}
```

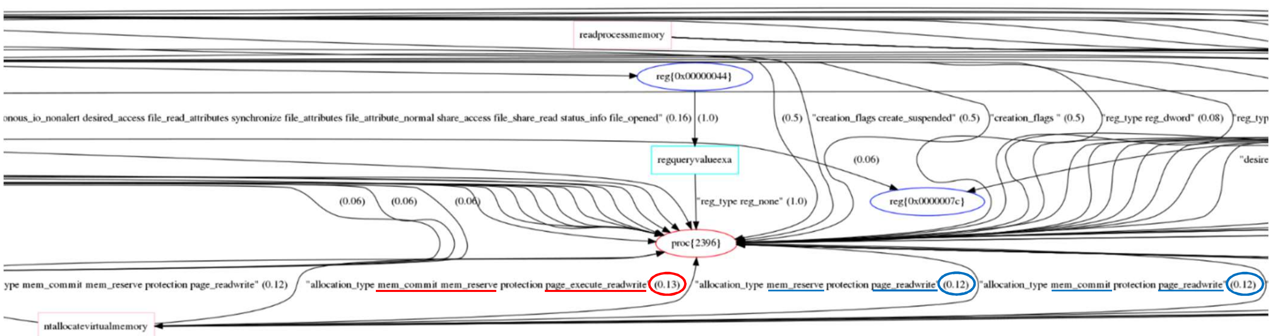



Fig 4. Visualization of edge after weighted

As can be seen from figure 4, our model has been able to learn the importance of the API call using the parameters protection and allocation_type, similar to the signature from the cuckoo report. It can be inferred that distinctive behaviors that humans can manually analyze and label as malicious activities, could be learned automatically using this approach. However, we expect the model could learn not only behaviors that human can explicitly see but also those that are more abstract that prove difficult or impossible for humans to manually analyze.

2) Information used for embedding

For now, only three types of API are represented in our graph, therefore, some important information might be ignored. For example, the two behaviors shown in List III and List IV are considered malicious activities:

List III. A query for the computer name


```
"call": {
  "category": "misc",
  "status": 1,
  "api": "GetComputerNameA",
  "return_value": 1,
  "arguments": {
    "computer_name": "WIN7X86-PC"
  },
  "flags": {}
},
```


List IV. Check for the Locally Unique Identifier on the system for a suspicious privilege


```
{
  "call": {
    "category": "system",
    "status": 1,
    "api": "LookupPrivilegeValueW",
    "return_value": 1,
    "arguments": {
      "system_name": "",
      "privilege_name": "SeDebugPrivilege"
    },
    "flags": {}
  },
}
```


The above two calls belong to category misc and system. Our model is unable to take these calls into consideration. To evaluate the effects of each API category on our model's malware detection ability, we have leveraged malware analyzing expertise to narrow down the most distinctive APIs for detecting malicious behaviors. The list of these APIs with their corresponding category is described in Table IX. The number of those APIs grouped by category is presented in Table X. Note that these categories are organized by Cuckoo, of which there are 16 in total: certificate, crypto, exception, file, iexplore, misc, netapi, network, ole, process, registry, resource, services, synchronization, system and ui. Other sandboxes might have different methods to group APIs.

Table IX. THE MOST DISTINCTIVE API FOR DETECTING MALICIOUS BEHAVIORS

API	category	API	category	API	category
NtDuplicateObject	system	URLDownloadToFileW	network	Process32NextW	process
DeviceIoControl	file	GetUserNameA	misc	RegSetValueExA	registry
MoveFileWithProgressTransactedW	file	NtCreateFile	file	CreateServiceA	service
OpenServiceA	service	GetComputerNameA	misc	RegOpenKeyExW	registry
NtSetValueKey	registry	NtLoadDriver	system	NtDelayExecution	synchronisation
RegQueryValueExA	registry	NtCreateProcess	process	NtDeviceIoControlFile	file
NtSetInformationFile	file	NtProtectVirtualMemory	process	NtAllocateVirtualMemory	process
NtCreateProcessEx	process	EnumServicesStatusA	service	ReadProcessMemory	process
NtCreateKey	registry	RegSetValueExW	registry	RegOpenKeyExA	registry
RtlCreateUserProcess	process	InternetSetOptionA	network	NtOpenFile	file
MoveFileWithProgressW	file	SetWindowsHookExA	system	InternetReadFile	network
CryptExportKey	crypto	LdrGetProcedureAddress	system	ObtainUserAgentString	network
OpenServiceW	service	SetWindowsHookExW	system	CryptGenKey	crypto
NtOpenProcess	process	EnumServicesStatusW	service	CreateServiceW	service
ControlService	service	Process32FirstW	process	GetComputerNameW	misc
CryptEncrypt	crypto	SetFileAttributesW	file	ShellExecuteExW	process
NtTerminateProcess	process	InternetOpenA	network	NtWriteFile	file
NtClose	system	LdrLoadDll	system	LdrGetDllHandle	system
GetAdaptersAddresses	network	NtCreateUserProcess	process	URLDownloadToFileW	network
CryptHashData	crypto	InternetOpenW	network	CreateProcessInternalW	process
RegQueryValueExW	registry				

Table X. NUMBER OF INTERESTING APIS BY CATEGORY

Category	Total API	Category	Total API	Category	Total API
crypto	4	network	8	service	7
file	8	process	13	synchronisation	1
misc	3	registry	8	system	8

Not only the API is the model missing out, but the flags field also conveys limited information. For example, the action demonstrated in List V would highly be a suspicious behavior since it is trying to register itself to execute whenever Windows starts, which is a common covert activity of malware:

List V. An activity of a malware trying to install itself for auto-run at Windows startup

```
{
  "category": "registry",
  "status": 1,
  "stacktrace": [],
  "api": "RegSetValueExA",
  "return_value": 0,
  "arguments": {
    "key_handle": "0x00000078",
    "value":
"c:\\windows\\system32\\mssrv32.exe",
    "regkey_r": "ImagePath",
    "reg_type": 1,
    "regkey":
"HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\services\\msupdate\\ImagePath"
  },
  "time": 1556598470.626408,
  "tid": 2512,
  "flags": {
    "reg_type": "REG_SZ"
  }
},
```

Our graph only encodes the flag field, however, the importance does not lie within the flag field, but the regkey in the arguments section, which specifies the registry path this API is trying to modify. Similarly, when changing the content of a file, the distinctive information used to distinguish between malware and benign samples is often the path to which the API is referring, or the value the API is trying to set. With such information, we cannot simply use n-gram or similar encoding methods, since the path vary. One solution is to encode each part of the path and assign a corresponding severity level. For example, the path HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\services\\msupdate\\ImagePath, would be divided into 4 parts as follow:

1. HKEY_LOCAL_MACHINE\\
2. SYSTEM\\
3. ControlSet001\\services\\msupdate\\
4. ImagePath

Here, 1. would be the root element separated by \\, which indicates the category of the registry, (i.e. HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_CONFIG). Each value would be assigned a corresponding severity, in this case HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE would be 1 and the others 0. This is because these two root category contain paths to important registry entries that malware usually interferes with (e.g. the path to set auto-start applications).

2. would be the child element of the root registry object. This element would be assigned a severity level according to its presence on a whitelist. Any elements contained within this list would be set to 1, otherwise they would be set to 0.

3. Regular expressions would be used to detect the presence of certain words in another whitelist, or to compute the number of elements separated by \\. There is considerable diversity in the strategy to encode the path and this is just one example of a possible solution.

3) Graph embedding

As mentioned in Section III, there are multiple methods forgenerating the graph embedding. Our model now only uses the weighted-sum of all the nodes to represent the graph embedding. However, this approach would omit information about the time each API is executed, or in other word, the order of each API being called. Now, intuitively, the solution might be to concatenate the nodes' embedding in the order of time they are executed. Yet, it is intricate to determine the exact execution sequence if multiple APIs having the same `time` field value, as manifested in Figure 5.

Another hurdle is to decide whether to order the nodes just by time of execution or also by the process calling them. The first option would ignore the relationship between the caller and the node being called, and considers the time the nodes are called only. The latter groups all nodes being called by the same process, and then orders each group of nodes by the time they are called.

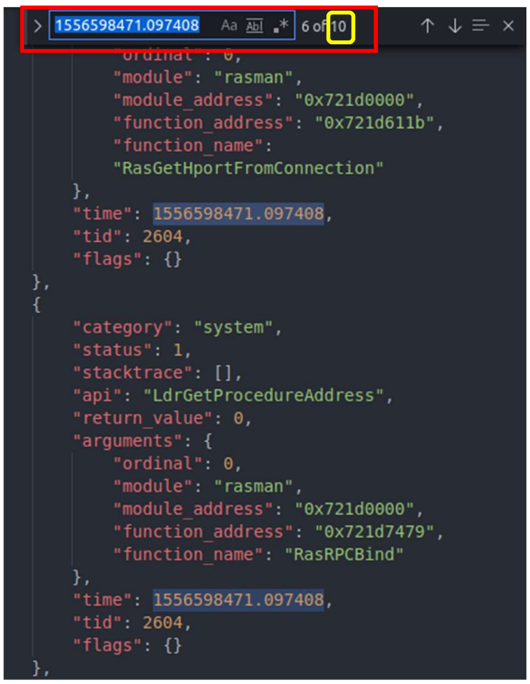


Fig 5. An example of 10 API containing the same value for time field

In previous works, there are already some efforts to represent the graph as a sequence of nodes to apply an RNN on. However, these works mostly use walking algorithms such as RandomWalk or DeepWalk to choose the order of the nodes [20][21]. He et al. proposed a modified random walk on heterogeneous graph in [22]. Yet, all these models are not either designed for, or evaluated on malware detection tasks, and the information of the nodes in these literatures does not contain time data. Nevertheless, these approaches do produce promising results and are inspiring, although in this specific task they would still overlook time data.

V. Conclusions & Future work

In conclusion, this paper outlinedseveral hallenges faced in the field of malware detection. It has also proposed a novel approach in an effort to help address the challenges. Our method has achieved comparable results with other state-of-the-art techniques. However, there are still several limitations in our strategy of representing behaviors, which we have also discussed in section IV to aim for future research.

As with other deep learning approaches, this cannot simply be a replacement for existing time-served tactics for malware detection, such as signature-based, but it could be implemented as a module to analyze more complicated or unseen samples (it might be an additional validationafter static analysis). However, this approach requires executing the sample in a virtual environment, hence it would take a while to first generate a report which contains behavioral data. Therefore, when implemented in a program, it would still be infeasible to process every file that static modules cannot detect as malware, but rather only execute suspicious files. Yet, knowing which files are suspicious might be another challenge.

Moreover, not every executable can be activated in a virtual environment due to anti-virtualisation techniques, or the fact that some executables require human interaction, especially those that are benign, which makes collecting benign samples a time-consuming task.

………………..

VI. References

[1] Yu, B., Fang, Y., Yang, Q. et al. A survey of malware behavior description and analysis. Frontiers of Information Technology & Electronic Engineering. 2018.

[2] Hongfa, X., Shaowen, S., Guru, V., Tian, L. Machine Learning-based Analysis of Program Binaries - A Comprehensive Study. IEEE Access. 2019.

[3] Yuxin, D, Wei, D., Shengli, Y., Yume, Z. Control flow-based opcode behavior analysis for Malware detection. 2014.

[4] Ki, Y., Kim, E., Kim, H.K. A novel approach to detect malware based on API call sequence analysis. Int. J. Distrib. Sens. Netw. 11, 659101. 2015.

[5] Tran, T.K., Sato, H. NLP-based approaches for malware classification from API sequences. In: 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES). 2017.

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

[7] Kolosnjaji, B., Zarras, A., Eraisha, G. et al. Empowering convolutional networks for malware classification and analysis. In: International Joint Conference on Neural Networks (IJCNN). 2017.

[8] Tobiyama, S., Yamaguchi, Y., Shimada, H. et al. Malware detection with deep neural network using process behavior. In: 40th Annual Computer Software and Applications Conference (COMPSAC). 2016.

[9] Wang, X,. Yiu, S.M. A multi-task learning model for malware classification with useful file access pattern from API call sequence. arXiv:1610.05945 [cs.SD], Cryptography and Security. 2016.

[10] Xiao, X, Zhang S, Mercaldo F, Hu G, Sangaiah A.K. Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. 2017.

[11] Sikorski, M., Honig, A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. Page xxviii.

[12] Naval, S., Rajarajan, M., Laxmi, V., Conti, M. Employing Program Semantics for Malware Detection. IEEE Transactions on Information Forensics and Security. 2015.

[13] J. Mathew, M. A. Ajay Kumara. API Call Based Malware Detection Approach Using Recurrent Neural Network – LSTM. Intelligent Systems Design and Applications. 2018.

[14] Wüchner, T., Ochoa, M., Pretschner, A. Malware Detection with Quantitative Data Flow Graphs. ASIA CCS '14: Proceedings of the 9th ACM symposium on Information, computer and communications security. 2014.

[15] Hung, N., Dung, P., Ngoc, T. et al. Malware detection based on directed multi-edge dataflow graph representation and convolutional neural network. 2019 11th International Conference on Knowledge and Systems Engineering (KSE). 2019.

[16] Wang, X., Ji, H., Shi, C. et al. Heterogeneous Graph Attention Network. arXiv:1903.07293. 2019.

[17] Wu, Z., Pan, S., Chen, F. et al. A Comprehensive Survey on Graph Neural Networks. Network Embedding and Graph Neural Networks. 2019.

[18] Zhou, J., Cui, G., Zhang, Z. Graph Neural Networks. A Review of Methods and Applications. arXiv:1812.08434. 2018.

[19] Anderson, B., Quist, D., Neil, J. et al. Graph-based malware detection using dynamic analysis. J Comput Virol 7, 247–258. 2011. https://doi.org/10.1007/s11416-011-0152-x.

[20] Jin Y., Joseph, F. JaJa, Learning Graph-Level Representations with Recurrent Neural Networks. arXiv:1805.07683. 2018.

[21] Perozzi, B., Al-Rfou, R., Skiena, S.. DeepWalk: online learning of social representations. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD ’14). Association for Computing Machinery, New York, NY, USA, 701–710. 2014

[22] He, Y., Song, Y., Li, J., Ji, C., HeteSpaceyWalk: A Heterogeneous Spacey Random Walk for Heterogeneous Information Network Embedding. 28th ACM International Conference. 2019.