Using Deep Graph Learning to Improve Dynamic Analysis-based Malware Detection

Tu Nguyen Minh[1], Hung Nguyen Viet[1*] and Nathan Shone[2]

[1]Faculty of Information Technology, LeQuyDon Technical University, 236 Hoang Quoc Viet, Hanoi, Vietnam.
[2]School of Computer Science & Mathematics, Liverpool John Moores University, Byrom Street, Liverpool, L3 3AF, UK.

*Corresponding author(s). E-mail(s): hungnv@lqdtu.edu.vn;
Contributing authors: tunguyenhsgs@gmail.com;
n.shone@ljmu.ac.uk;

Abstract

Detecting zero-day malware using dynamic analysis techniques has proven to be far more effective than traditional signature-based methods. One specific approach that has emerged in recent years is the use of graphs to represent executable behavior, which can be subsequently used to learn patterns. However, many current graph representations omit key parameter information, meaning that the behavioral impact of variable changes cannot be reliably understood. To combat these shortcomings, we present a new method for malware detection by applying a graph attention network on multi-edge directional heterogeneous graphs constructed from API calls*. The experiments show the TPR and FAR scores demonstrated by our model, achieve better performance than those from other related works.

Keywords: malware detection, dynamic analysis, deep learning, graph representation.

*Source code and dataset: https://github.com/miamor/HAN-sec-new

1 Introduction

The COVID-19 pandemic has led to rapid shifts in working patterns, with remote working becoming the new norm globally. Unfortunately, this has exposed a myriad of flaws with access technologies, security technologies and user education. One of the threats that has thrived during the pandemic is malware [1]. Despite the many cloud, network and endpoint based defensive technologies available, it has re-emphasized its position as a continual and significant security challenge [2].

The global pandemic has provided an ideal situation to demonstrate the adaptability and rapid evolutionary capabilities of malware. COVID-19 has been extensively leveraged as successful bait for many social engineering led malware campaigns. This has resulted in an estimated 667% spike in phishing emails, 105% growth in ransomware samples and 128% growth in new Trojans over the past 12 months [1]. This in part helps to explain why the development of effective detection techniques presents such a challenge. Furthermore, it is estimated that 560,000 [2][3] new pieces of malware are detected every day, and new evasion techniques are helping to drive this growth. Therefore, the need to identify and categorize malware (both new and existing), in an effective way, is clear.

Existing malware analysis techniques can typically be categorized into two main types: static and dynamic [**?**]. Static techniques such as binary fingerprinting, packer detection and debugging, focus on the identification of structural patterns and signatures within the binary without requiring its execution. However, malware samples that employ obfuscation such as code refactoring, NOP code insertion and encoding can easily circumvent such techniques. Dynamic techniques can overcome the majority of such countermeasures, as they primarily focus on the tracing and identification of behavioral patterns exhibited by the binary during runtime [**?**]. Despite being less vulnerable to obfuscation, such techniques are not immune. Similarly, dynamic approaches cannot replicate all possible environmental conditions to capture an exhaustive list of possible behavioral patterns.

Manually analyzing a binary to identify malicious behavior is a highly laborious and time-consuming process. Consequently, recent research projects have aspired to increase the level of automation. One of the current research areas of interest is the application of visualization approaches to better represent the vast and complex behavior of executing binaries. As a dynamic technique, such approaches are able to construct behavioral representations with a good level of accuracy. There have been various different approaches proposed such as the use of greyscale images [4], RGB images [5], heat maps [6], histograms [7] and most commonly call flow graphs [8]. The results achieved in existing works have demonstrated promising levels of improvement over more traditional approaches.

However, despite these improvements, behavioral obfuscation still poses a significant technical challenge to these approaches. Simply, obfuscation (commonly used by metamorphic and polymorphic malware) introduces varying

degrees of change. Without understanding the context behind these changes, it is difficult to ascertain the reasoning (e.g. are additional calls designed to render existing representations obsolete or do they serve a specific functional purpose). One major limitation of current graph-based methods [9][10][11] is the data used to generate the representations is usually abstract and frequently omits important information, which may provide much needed context.

To address these limitations, this paper proposes a new graph-based representation for binaries, which embeds various types of useful information typically omitted by existing techniques. This information includes APIs used, API calls, connection types and key arguments supplied to each API. The benefit of this is the increased level of contextual understanding it offers. To complement this method, a bespoke neural network model has been devised, which has been trained on node-level and semantic-level embedding. The main contributions of this work are as follows:

- A new method to represent a binary as a Multi-edge Directional Heterogeneous Graph (MDHG), which unlike existing methods, can retain more important behavioral characteristics.
- A bespoke deep learning model to learn features from the constructed graphs. This introduces an edge-weighing layer, along with data encoding techniques, to focus on the arguments of each API, thus weighting the importance of each call.

The remainder of this paper is organized as follows: Section 2 provides an overview of related research focused on automated malware detection. Section 3 provides a detailed description of our proposed approach. Section 4 presents and discusses the experiment configuration and the results achieved. Finally, conclusions regarding the findings of this paper are drawn in Section 5.

2 Related Work

In an attempt to address the challenge of effective identification of malware, there are many ongoing research projects seeking to develop new and improved techniques. This section highlights some of the cutting-edge work pertinent to the focus of this paper. Currently, the two main general categories of approach for representing behavioral data are text-based and graph-based.

2.1 Text-Based Representations

For text-based methods, most involve the use of the use of machine learning (either shallow or deep learning) for either decisions or feature extraction. Yu et al. [12] gave an overview of behavioral description methods including XML-based, semantic description methods, description languages and several text-based. Hongfa et al. [13] represented system call sequences with their MIST instructions and used an n-gram algorithm to extract features. In [14] 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 [15] 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 [16] enhanced the conventional ML algorithms for API calls analysis by doc2vec, N-gram and TF-IDF methods. The n-gram analysis method achieved some good results, but it faced the challenge of 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 [17] were the first to use a hybrid model of RNN and a machine learning classifier to predict the next API call. Kolosnjaji et al. in [18] proposed a method to detect and classify malware 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 malware with obfuscation and detection evasion techniques. RNN and LSTM are also experimented with in various existing works but largely face the same problems [9][19][20]. Homayoun et al. compared the use of LSTM and CNN when identifying ransomware families through API call sequences, concluding that LSTM offered increased accuracy [21]. Qin et al. [22] also propose an API call sequence-based detection for ransomware. Their method is based on an adapted version of the TextCNN model.

2.2 Graphical Representations

An emerging trend observed in related research is the application of graph neural networks, which has proven effective for both representing behavior and for feature extraction. Anderson et al. [23] generated Markov chain graphs from dynamic trace data, and applied graph kernels to acquire a similarity matrix, which was sent to a Support Vector Machine (SVM). Naval et al. [24] 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. [10], however, this work only formalizes heuristics to identify malware. Work by Hung et al. [11] outline an extended version of the traditional QDFG by subsequently applying a Graph Convolutional Network (GCN). Although this graph succeeded in expressing more informative data, it still lacks 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.

In [25], System-call Dependency Graphs (ScD-graphs) and constructed using traces captured through dynamic taint analysis. A set of detection and classification techniques are then applied on a weighted directed graph. Dynamic taint analysis is also used by Ding et al. [26], who use the data to construct a graph of common behavioral features of malware families.

Zhang et al. propose the use of heterogeneous graphs for the detection of malicious domains through the analysis of DNS traffic [27]. Their approach

tackles the omission of important information from graphical representation of domain names through the use of an attributed heterogeneous information network.

Jiang et al. [28] also proposed the use of heterogeneous graphs to detect Android malware. The graphs are used to represent the relationships between code regions in Android apps, which enables the structural and semantic features to be included.

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, especially in malware detection problem. Shen Wang et al. [37] proposed the use of heterogeneous graph for malware detection. The authors used invariant graphs to capture the interactions between different pairs of system entities. They used an adjacency matrix to store the structure of graph. In this way, this graph can represent the relation between two entities by a single edge but cannot store other important information between them such as the times that a process called another process, or the quantity of information transferred. We believe our work is the first to represent the most important behavioral data of a program, as a heterogeneous graph.

3 Proposed Method

Throughout the remainder of this paper, we will use the term "dynamic behavioral data" in reference to the primary source of data being utilized. In the context of this paper, this specifically refers to the sequence of API calls (and their associated data) observed from an executing binary. The data used in this proposed method will be generated through the use of Cuckoo, the automated malware analysis sandbox. The methods presented in this paper are based heavily upon those featured in our previously published work [11][30].

3.1 Graph Representation

The dynamic behavioral data captured is used to construct the Multi-edge Directional Heterogeneous Graphs (MDHGs), subsequently an attention neural network (inspired by the work of Wang et al. [31]) is used to differentiate between malicious and benign binaries.

Entities and Connections: The MDHGs are constructed using six types of entities (graph nodes): Process, File, Registry, ProcessAPI, FileAPI and RegistryAPI. There are also six types of connections that can be established between the entities:

- Process-ProcessAPI: connection between a process handle (process entity) and a Process API.

- File-FileAPI: connection between a file handle (file entity) and a File API.
- Registry-RegistryAPI: connection between a registry handle (registry entity) and a Registry API.
- Process-FileAPI: connection between a process handle and a File API.
- Process-RegistryAPI: connection between a process handle and a Registry API
- Self-loop: A node can connect to itself, in order for its own features to be taken into consideration.

It is important to note that some potential connections were not listed previously, as they would never normally be made e.g. connections between a file handle and a registry API, or between a registry handle and a file API.

All of the entities and connections used fall into 3 main categories: process, file, and registry. The use of these categories was inspired by the work of Wüchner et.al. [10], who determined that processes, files, sockets, and registry keys are of great importance when identifying malicious behavior. There is no restriction on the number of entity/node categories that can be used in the generation of the MDHG. However, the restriction to 3 categories is used due to limitations in the data being collected. If more categories of entities and connections need to be represented, the feature space would increase significantly, and therefore the limited amount of data available would be inadequate for learning in a feature space of such size.

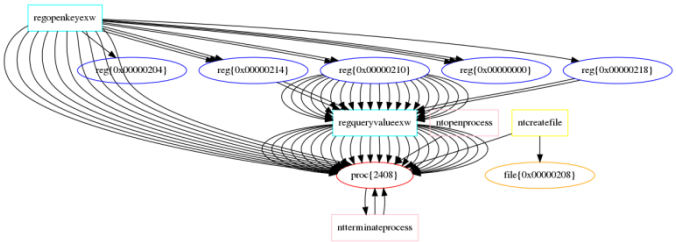
For each API call node (entity), only the name of the API is used for feature encoding, and all arguments are placed in the edge data. Therefore, there might be multiple connections to one single API call node. The MHDG is directional, the principles to determine the direction of each connection is similar to the work done by Hung et al. [11]. In the work, all API calls that operate the task of opening, creating, writing, or any modifying actions towards a file or registry would be the source nodes, and the destination nodes would be 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 the destination node). Fig. 1 shows an example of the behavioral analysis report returned by Cuckoo from analyzing a malware sample.

```
{
        "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":{}
}
```

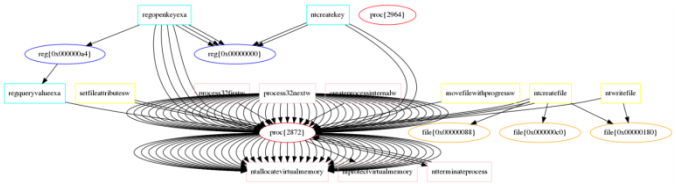

Fig. 1: An example of a binary behavior report generated by Cuckoo

The example given in Fig. 1 is a small behavioral sample constructed using 2 nodes: a process entity (ID 768) and a ProcessAPI entity (WriteProcessMemory) (because this API belongs to category process). One edge exists from the ProcessAPI entity to the process entity. The node features are generated by encoding either the API name (if it is an API entity) or the name of its type (e.g. whether it is a process/file/registry entity). For each connection, the feature data is obtained from the flag fields of the API that the connection links to (or from). These flags fields are generated by Cuckoo giving an insight into important information about that call.

In our paper, we define meta-path differently from the original work of Wang et al. [31]. We do not define connections between two nodes of the same type (for example movie-movie (connection between a "movie" node with a "movie" node)) through a middle node of different types (such as movie-actor-movie (2 "movie" nodes connected via an "actor" node) and movie-director-movie (2 "movie" nodes connected via a "director" node). We only define a type for the connection between two nodes directly. For example, Process-ProcessAPI (connection between a "Process" node and a "ProcessAPI" node) through a "Process-ProcessAPI" edge (an edge that has the type of "Process-ProcessAPI"), or it could be written as Process-(Process-ProcessAPI)-ProcessAPI. After all, the importance of a heterogeneous graph is the heterogeneity of the nodes and edges the graph can support.



(a) Graph representation of a benign sample



(b) Graph representation of a malware sample

Fig. 2: Graph representation of a malware and benign sample.

Embedding Entities and Edges Argument: 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 will test on both skip-gram and TF-IDF encoding for node names (API names) and edge arguments (flags fields of an API). Fig. 1 shows a representation of a malicious and benign sample.

3.2 Malware Detection

Detecting malware from a constructed graph can be considered as a graph classification task. There are two main approaches for this task: graph embedding (trying to find representations of graph nodes and edges) and graph feature extraction (e.g. using graph convolutional neural networks). A recent survey [32] grouped graph deep learning models into 5 categories: graph convolution, graph attention, graph generative, graph spatial-temporal networks and graph auto-encoders. Zhou et.al. [33] 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. Hung et al. [11] used GCN to extract features from the graph but this spectral approach requires training and detection on a specific graph structure, since the learned filters depend on the Laplacian eigenbasis. 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 a 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. Wang et al. [31] 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.

When analyzing the behavior of malware, many calls between two entities and related information may be very important and should be accounted for when detecting malware. Inspired by the idea of Wang [31], we propose a new approach, whereby the main contribution is the edge-weighing layers that can learn the importance of each connection among a set of connections between two nodes, since our graphs is multi-edge. The specifics of this process are outlined below.

Table 1: Notations and Explanations

Notation	Explanation
$\oslash$	Meta-path
h	Node features
e_{ijp}	Importance of node i to node j through path p
$\alpha_{ijp}^{\oslash}$	Weight of node pair i, j through path p
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
l_p	Initial edge features
$l_p^{'}$	Weighted edge features
Y_p^{ij}	Weight of path p
τ_i	Weight of final node i
$N^{\oslash}$	Meta-path based neighbors
q	Semantic-level attention vector
$w_{\oslash}$	Importance of meta-path $\oslash$
$\beta_{\oslash}$	Weight of meta-path $\oslash$
$Z^{\oslash}$	Semantic-specific node embedding
Z	Graph Embedding

Edge-Weighting: In [31], embedding $z_i^{\oslash}$ of node i is computed by weighted-aggregation of the embedding of its meta-path based neighbors:

$$z_i^{\oslash} = \sigma(\sum_{j \in N+i^{\oslash}} \alpha_{ij}^{\oslash} \cdot h_j)) \tag{1}$$

$$a_{ij}^{\oslash} = \frac{exp(\sigma(a_{\oslash}^T \cdot [h_i \vee h_j]))}{\sum_{k \in N_i^{\oslash}} exp(\sigma(a_{\oslash}^T \cdot [h_i \vee h_k]} \tag{2}$$

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 also 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). Equation 2 would then become:

$$a_{ij}^{\oslash} = \frac{exp(\sigma(a_{\oslash}^{T} \cdot [h_i \vee l_{ij} \vee h_j]))}{\sum_{k \in N_i^{\oslash}} exp(\sigma(a_{\oslash}^{T} \cdot [h_i \vee l_{ik} \vee h_k]} \tag{3}$$

This is the case when there is only one connection between node j and node i, l_p is therefore l_{ij}. However, graphs in our problem are multi-edge, which means there could be multiple connections between two nodes. For example, Fig. 3 exemplifies multiple calls to RegQueryValueExW but with different arguments, therefore each call should have different importance values.

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

$$a_{ijp}^{\oslash} = \frac{exp(\sigma(a_{\oslash}^{T} \cdot [h_i \vee l_p \vee h_j]))}{\sum_{k \in N_i^{\oslash}} \sum_{m \in P_{ik}} exp(\sigma(a_{\oslash}^{T} \cdot [h_i \vee l_m \vee h_k]} \tag{4}$$

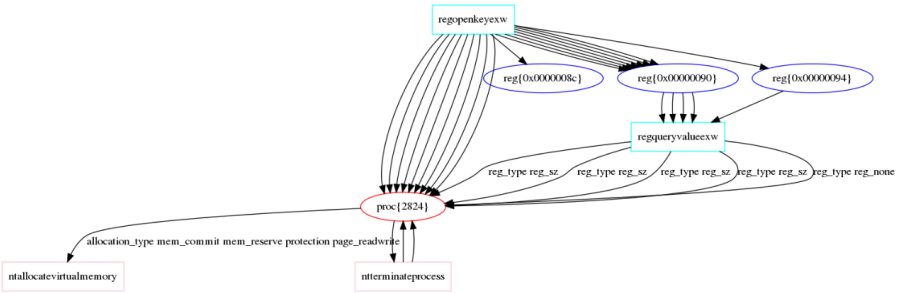


Fig. 3: The same API (RegQueryValueExW) is called from process id 2824 with different arguments.

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 $h_j \vee l_p;\ p \in m$) instead of m connections to one node (having features h_j). However, the purpose of building a multi-edge graph is to expect that the model could learn the importance of each edge in the set of connections between two nodes. In other words, we want to focus more on learning the importance of the edge arguments.

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

$$u_p^{ij} = att_p(l_p) = \sigma(U_{ij}^{T} \cdot l_p + b) \tag{5}$$

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

$$Y_p^{ij} = softmax(u_p^{ij}) = \frac{exp(\sigma(U_{ij}^{T} \cdot l_p + b))}{\sum_{m \in P_{ij}} exp(\sigma(U_{ij}^{T} \cdot l_m + b)} \tag{6}$$

And the weighted embedding of path p:

$$l_p^{'} = Y_p^{ij} \cdot l_p \tag{7}$$

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 through path p:

$$\alpha_{ijp}^{\oslash} = softmax(\sigma(a_{\oslash}^T \cdot [h_i \vee l_p^{'} \vee h_j])) \tag{8}$$

And the meta-path based embedding of node j:

$$z_i^{\oslash} = \sigma(Q^T \cdot \sum_{k \in N_i^{\oslash}} \sum_{m \in P_{ik}} \alpha_{ijp}^{\oslash} \cdot [h_i \vee l_p^{'}]) \tag{9}$$

Fig. 4 illustrates a node-level embedding being calculated by aggregating the 3 meta-path based embeddings.

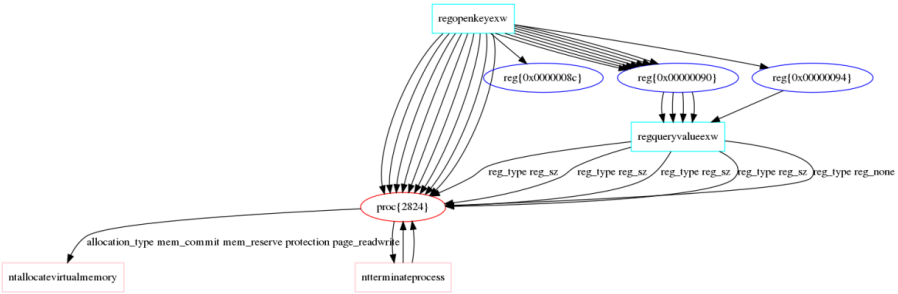


Fig. 4: Aggregation of meta-path based neighbors

Sematic-level Embedding: Once we have the node-level embedding, an attention network is used for learning semantic meaning:

$$w_{\oslash_i} = \frac{1}{\mathrm{V}} \sum_{i \in V} q^T \cdot \sigma(W \cdot z_i^{\oslash} + b) \tag{10}$$

$$\beta_{\oslash_i} = softmax(w_{\oslash_i})$$

And the final embedding of node i:

$$Z_i = \sum_{k=1}^{P} \beta_{oslash_k} \cdot z_i^{\oslash_k} \tag{11}$$

Graph Embedding: here are a variety of ways to obtain the graph embedding after computing the embedding for the nodes. In this work, the final graph embedding is obtained by accumulating the weighted final node embedding as in 12.

$$Z = \sum_{i \in V} \tau_i \cdot Z_i \tag{12}$$

Table 2: Feature types in MFMAD dataset

Feature Name	Description	Extraction Method
Byte sequence	Sequence of byte code	Binary code
CFG	Control Flow Graph	Constructed from assembly code
ASM code	Assembly code	PE files disassembled using radare2
Image	Image representation of PE files	Binary code converted to greyscale images as described in [22] Convert binary code into a fixed-sized color image as described in [37].
Behavior report	.json description of PE files' behavior	Use Cuckoo to generate by executing binary files in Windows 7 environment
MDHG representation of API calls (our proposed graph)		Constructed from behavior report (Section 3).

4 Evaluation

4.1 Datasets

To evaluate our proposed solution, we will be using real-world malware samples. To accomplish this, we propose the Multiple Features for Malware Analysis and Detection (MFMAD) dataset, which contains various features (summarized in Table 1) extracted from PE files. The 8,109 PE files used to generate this dataset were collated from numerous online sources such as Virustotal and filtered by their Cuckoo score to ensure correct identification as malware. The dataset is comprised of 1,685 benign and 6,424 malicious samples, meaning it is imbalanced. However, it is important to note that unlike normal circumstances, the imbalance is reversed (i.e. the number of malware samples is significantly higher than that of benign samples).

This dataset is actually an extended version of our previously published work [11]. The dataset is divided into 4 sub-datasets, and in turn, each of these is divided into subsets, as outlined in Table 2. It is important to note that, within each sub-dataset, none of the samples are duplicated in any of the subsets. The training-test ratio in each of the sub-datasets is 7:3.

Original Dataset (OD): We use the train/test subset for training and testing. As shown in Table 3, this subset includes 1,088 samples in total, which is composed of 655 malware and 433 benign samples, the same as in [11]. Similarly, the unknown subset includes 637 malware samples that ClamAV was unable to detect until 2/6/2019. The benign_555 and pack1 subsets are also included, which consist of 555 benign and 4,620 malware samples respectively. These two subsets do not contain any samples from the train/test subset.
Enhanced Dataset (ED): The enhanced train/test subset consists of 2,379 items, 988 of which are benign, the rest are malware. This enhanced subset is made up by combining the original train/test subset, benign_555 subset, and additional 741 malware samples, which are not duplicated in any other set (i.e. original train/test, unknown or pack1 subset).

Table 3: Feature types in MFMAD dataset

Sub-dataset	Description	Malware (M) Benign (B)	Total	Subsets
Original Dataset (OD)	The same dataset as in [15], adding 555 benign samples.	5912 M	6900	train/test (655M, 433B)
				unknown (637M)
		988 B		pack1 (4620M)
				benign_555 (555B)
Enhanced Dataset (ED)	Enhancement of OD, benign files are 433 Windows system files and 555 samples.	6648 M	7636	enhanced train/test (1391M, 988B)
		988 B		unknown (637M)
				pack1 (4620M)
Nosys Dataset (NSD)	None of the 1091 benign files are Windows system files.	1119 M	2210	None
		1091 B		
Full Dataset (FD)	Contains all types of files.	1804 M	3489	None
		1685 B		

Table 4: Original Dataset Composition

Subset	Total No. Samples	No. Malicious Samples	No. Benign Samples	Purpose
train/test	1088	655	433	-
train	761	463	298	Training
test	327	192	135	Testing
unknown	637	637	0	Testing
benign_555	555	0	555	Testing
pack1	4620	4620	0	Testing

Table 5: Enhanced Dataset Composition

Subset	Total No. Samples	No. Malicious Samples	No. Benign Samples	Purpose
enhanced train/test	2379	1391	988	-
train	1665	954	711	Training
test	714	437	277	Testing
unknown	637	637	0	Testing
pack1	4620	4620	0	Testing

Nosys Dataset (NSD): This dataset does not contain any Windows system files. The dataset composition is summarised in Table 5.

Full Dataset (FD). Note that "full" here does not mean "all samples in the full dataset", but implies all types of files (e.g. system files, normal, known/unknown malware), which are randomly selected from full data. The exact composition is detailed in Table 6.

Table 6: Nosys Dataset Composition

Subset	Total No. Samples	No. Malicious Samples	No. Benign Samples	Purpose
Nosys Dataset	2210	1119	1091	-
train	1565	783	782	Training
test	645	336	309	Testing

Table 7: Full Dataset Composition

Subset	Total No. Samples	No. Malicious Samples	No. Benign Samples	Purpose
Full Dataset	3489	1804	1685	-
train	2616	1353	1263	Training
test	873	451	422	Testing

4.2 Test Scenarios

The evaluations undertaken in this paper are summarized below:

1. Train on Original Dataset: We evaluate on the test set of train/test subset and unknown subset, to test the ability of the model in detecting zero-day malware, and on benign_555 subset to examine the relations/similarities between Windows system files and normal applications.
2. Train on Enhanced Dataset: We evaluate on the test set (of enhanced train/test subset) to see if the FPR improves when enhancing benign samples in the training set (instead of using only Windows system files), and the unknown subset to compare performance changes against Scenario 1.
3. Train on Nosys Dataset: We evaluate the model by training on the Nosys Dataset and testing on a dataset that contains none of the Windows file systems. We also test the model on 433 Windows system files from the Original Datasets, combined with the result from Scenario 1 and 2 to conclude whether the significant difference between Windows system files and other normal applications might confuse the model, thence leading to a poor FPR.
4. Train on Full Dataset: In this scenario, we train and test the model using the full dataset.

4.3 Implementation of Other Models

In order to compare our results against cutting-edge techniques proposed in existing related research, we have also implemented several models based on these devised solutions that are shown in Table 8.

4.4 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

Table 8: Implementations of other models

Method	Features used	Description
Asm_cnn	OpCode. Created from assembly code, which is retrieved by using radare2 to disassemble PE files.	Apply a CNN 1D on generated OpCode. [45]
Asm_lstm		Apply a LSTM network on generated OpCode. [45]
Img	Image. Created from binary code.	Apply a CNN on generated image. [45]
Ngram	Use frequency of extracted n-grams as feature vectors.	Apply an SVM on the n-grams frequency. [64]

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 cases covered, and a null entry for potentially unseen words. When using TF-IDF encoding, we construct a 4-dimensional feature vector of each edge. For skip-gram encoding, input is the whole argument string and the output is a 10-dimensional feature vector.

4.4.1 Training on the train set from Original Dataset

Benign files in the Original Dataset are Windows system files only. In this scenario, we will train on train set (of train/test subset), then test on: test set (of train/test subset), unknown, pack1, and benign_555 subset. Table 8 shows the evaluation of different models using various methods of encoding node and edge data on the Original Dataset. The results show that using edge-weighting gives the best performance on train/test subset, and that using edge-weighing layers outperforms the original GAT model for heterogeneous graphs as proposed by Wang et al. [31].

We have implemented a simple classifier on embedding sequences using these skip-gram and TD-IDF encoders to investigate the performance of each encoding method, the results from this are shown in Table 9. It is noteworthy that the 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 the sequences of nodes names only does not convey much meaning, in a sense that there is not much difference between the sequences of API being called by benign and malware samples. As mentioned in Section 3, differences usually lie within the arguments of each call. Also, 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

Table 9: Comparing different encoding techniques on the Original Dataset

	Train/test (1088)						Unknown (637 M)
	Train (298 B, 463M)			Test (135B, 192M)			
	ACC	**TPR**	**FPR**	**ACC**	**TPR**	**FPR**	**TPR**
Skip-gram + TF-IDF	96.19%	96.98%	5.03%	92.66%	92.19%	6.67%	89.64%
Skip-gram	93.82%	95.90%	9.40%	88.69%	89.06%	11.85%	96.55%
TF-IDF	90.41%	92.44%	12.75%	91.74%	92.19%	8.89%	96.23%
Skip-gram (no edge-weighing)	88.04%	86.39%	10.07%	85.63%	83.33%	11.11%	85.22%
TF-IDF (no edge-weighing)	80.81%	79.05%	16.44%	84.40%	80.21%	9.63%	84.46%

Table 10: Classifying based on encoding edge arguments and nodes names

	Train/test (1088)				Unknown (637 M)		Pack1 (4620 M)		Benign_555 (555 B)	
	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%

Table 11: Comparison of our model with other methods on Test set from Original Dataset

Model	ACC	TPR	FPR
Our model	92.66%	92.19%	6.67%
Cnn bytes	80.12%	75.00%	12.59%
MalGCN [11]	86.22%	88.02%	9.66%
Lstm bytes	78.59%	72.40%	12.59%
QDFG-GCN [11]	74.31%	87.05%	44.04%
Cnn asm	90.52%	96.35%	17.78%
QDFG-KNN [11]	62.37%	49.59%	15.49%
Lstm asm	88.69%	94.27%	19.26%
Cnn img	81.19%	87.24%	19.27%
DGCNN [34]	86.24%	90.15%	19.77%
Ngram	79.88%	96.65%	44.80%

Table 12: Comparison of our model and others on unknown subset

Engine	Acc	Engine	Acc
Lstm asm	98.58%	ESET-NOD32	77.75%
Ngram	97.80%	K7GW	74.21%
Cnn asm	97.01%	Endgame	74.08%
Cnn img	96.69%	K7AntiVirus	73.95%
Our model	**89.64%**	Invincea	73.43%
MalGCN	84.03%	CrowdStrike	72.38%
McAfee-GW631	82.59%	Sophos	70.29%
Fortinet	82.59%	AVG	69.63%
Cnn bytes	82.22%	GData	69.24%
Lstm bytes	79.17%	Rising	68.06%
Microsoft	78.93%	Avira	67.54%
McAfee	77.75%	VBA32	67.28%

is the same in every circumstance, therefore can detect from an early stage which calls seem to be abnormal.

Table 10 and Table 12 present comparisons of our best model (using skip gram encoding for node names and TF-IDF encoding for edge arguments) and other published methods on two subsets: test set from train/test subset and the unknown subset. The results of other methods are derived from published literature [11],[34] and from our implementation demonstrated in Section 4.2 (due to a lack of source code for other methods). Note that from now on, unless specified, "our model" refers to the 1st model in Table 9 (using skip gram encoding for node names and TF-IDF encoding for edge arguments). Table 10 shows that our best model outperformed the existing methods in all of the assessed metrics.

Table 12 is the comparison on unknown subset to test the ability of detecting new (zero-day) malware. Note that our implementations of the other four methods have higher accuracies (True Positive Rate (TPR)), however, running these models on the benign_555 subset to measure the False Positive Rate (FPR) gives very poor performance (Table 12). Although our model has the lowest value of FPR, it is still quite high.

Table 13: Comparison of our model and others on benign_555 subset

Engine	FPR
Lstm asm	97.78%
Our model	**21.29%**
Cnn bytes	92.04%
Cnn asm	97.78%
Ngram	100%
Lstm bytes	92.48%
Cnn img	100%

Table 14: Results of our models on pack1 and benign_555 subset

	Pack1 (4620 M)			Benign_555 (555 B)		
	ACC	TPR	FPR	ACC	TPR	FPR
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%

When we conducted experiments to evaluate our models (trained on train/test subset from Original Dataset) on pack1 and benign_555 subsets, all models still yield a high TPR on the pack1 subset. However, they achieve a poor FPR on the benign_555 subset. The results are shown in Table 13.

It can also be inferred from this table that combining skip-gram and TF-IDF encoder (using skip-gram to encode edge arguments and TF-IDF to encode nodes names) does not achieve the best results in both subsets, but it offers superior stability, hence it is considered more promising. Test scenarios demonstrated in subsequent sections will attempt to investigate further into the high FPR issue.

4.4.2 Training on the train set from Enhanced Dataset

We could see from Table 13 that all models result in considerably higher FPRs on benign_555, when compared with the train/test subset. We hypothesize that this is caused by differing DLL usage behaviors between the benign_555 and train/test subsets. More specifically, benign_555 samples all require external DLLs to be loaded in order to execute. Additionally, the benign samples in the train/test subset are Windows system files, with many calls to system DLLs, performing actions very similar to malware (e.g. changing important registry values, retrieving OS information). Therefore, we have trained and evaluated our model on the Enhanced Dataset, the results of which are shown in Table 14. It can be seen Table 14 that our model (when trained on the Enhanced train/test subset) causes a slight decline in TPR on the unknown subset, but achieves improved results with the other subsets.

Table 15: Evaluation result of our model training with Enhanced Dataset

Dataset/testset		ACC	TPR	FPR
Enhanced train/test	Train	96.22%	96.02%	3.52%
	Test	93.00%	92.68%	6.45%
Unknown		–	88.23%	–
Pack1		–	90.77%	–

Table 16: Results from train and evaluating on the Nosys Dataset

	Train			Test			Sys files
	ACC	**TPR**	**FPR**	**ACC**	**TPR**	**FPR**	**FPR**
Our model	**94.20%**	**93.56%**	**5.15%**	**89.71%**	**85.37%**	**6.10%**	**13.21%**
Asm_cnn	97.86%	98.93%	3.20%	97.33%	98.43%	3.77%	76.91%
Asm_lstm	93.80%	99.87%	12.25%	94.97%	99.06%	9.12%	65.82%
Bytes_cnn	99.36%	100%	1.37%	98.01%	100%	4.21%	77.60%
Bytes_lstm	85.85%	86.10%	14.44%	88.56%	77.99%	16.84%	39.96%
Img	77.92%	82.89%	27.03%	80.82%	85.85%	24.21%	15.94%

Table 17: Train and evaluate on Full Dataset

	Train			Test		
	ACC	**TPR**	**FPR**	**ACC**	**TPR**	**FPR**
Our model	**92.23%**	**90.21%**	**3.13%**	**90.02%**	**87.50%**	**4.30%**
Asm_cnn	96.69%	99.22%	5.99%	93.77%	96.52%	7.90%
Asm_lstm	92.90%	98.70%	13.27%	91.13%	97.01%	12.46%
Bytes_cnn	98.88%	99.82%	2.20%	95.80%	96.52%	4.68%
Bytes_lstm	92.67%	97.84%	13.29%	88.4%	96.02%	16.72%
Img	86.38%	92.99%	20.65%	81.89%	93.03%	24.92%

4.4.3 Training on the train set from Nosys

In this experiment, we train on train subset of Nosys Dataset, and evaluate on both subsets, and 433 Windows system files (from train/test subset of the Original Datasets). The results obtained are presented in Table 15.

We can infer that the significant difference between Windows system files and other normal applications might confuse the model, thus leading to the poor FPRs (model trained with Windows system files fails to recognize benign samples as normal, and vice versa, model trained with non-system files does not perform well on detecting system files as benign). One more detail that we can deduce is that our model is more stable in all scenarios (although in some cases it does not achieve the best result), irrespective of the size or the composition of dataset.

4.4.4 Training on the train set from the Full Dataset

Final test scenario is to train and evaluate on the Full Dataset, the results obtained are shown in Table 16.

```
{
  "markcount": 2,
  "families": [],
  "description": "Allocates execute permission to another process
                                        indicative of possible code
                                            injection",
  "severity": 3,
  "marks": [
    {
      "call": {
        "category": "process",
        "status": 1,
        "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. 5: A signature for malicious activity according to YARA rules

From multiple test cases, we can infer that our model, in all cases, outperformed in terms of stability and in some cases, surpasses the TPR and FPR of other approaches.

5 Discussion

5.1 Edge-weighing

For a more intuitive evaluation and deeper understanding, we have visualized the weights of each edge produced by our model.

Fig. 5 shows an example of a signature for malicious activity of a malware sample. The signature is generated along with with Cuckoo report, by applying YARA rules which are contributed by the 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 Fig. 6.

As can be seen from Fig. 6, 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,

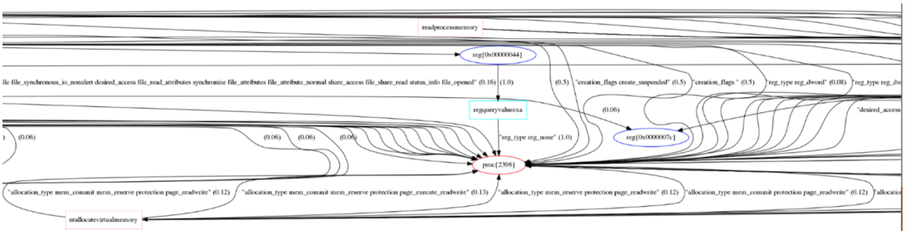


Fig. 6: Visualization of edge after weighting

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

Fig. 7: A query for the computer name

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

Fig. 8: Check for the Locally Unique Identifier on the system for a suspicious privilege

could be learned automatically using this approach. However, we expect the model could learn not only behaviors that humans can explicitly see but also those that are more abstract that prove difficult or impossible for humans to manually analyze.

5.2 Information Used for Embedding

For now, only three types of API are represented in our graph, therefore, some important information might be overgeneralized. For example, the two behaviors shown in Fig. 7 and Fig. 8 are considered malicious activities:

The above two calls belong to category misc and system. Our model is unable to take these specific calls into consideration. To evaluate the effects

Table 18: The most distinctive API for detecting malicious behaviors

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

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 18. The number of those APIs grouped by category is presented in Table 19. 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 of grouping APIs.

It is not just the overgeneralization of the API that can cause issues for the model. The flags field used conveys a limited amount of information. For example, the action demonstrated in Fig. 9 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:

Currently, our graph only encodes the flag field, however, the importance it not specifically within flag field, but the accompanying regkey in the arguments section, which specifies the registry path this API is trying

Table 19: Number of interesting APIs by category

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


```
  "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"
  }
},
```


Fig. 9: An activity of a malware trying to install itself for auto-run at Windows startup

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 path-based information, we cannot simply use n-gram or similar encoding methods, since the paths 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 follows:

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 blacklist. 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 blacklist, 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.

5.3 Graph Embedding

As mentioned in Section 3, there are multiple methods for generating the graph embedding. Our model currently only uses the weighted-sum of all the nodes to represent the graph embedding. However, this approach would omit temporal information about API execution, in other words, the sequence 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 complex to determine the exact execution sequence if multiple APIs have the same time field value, as demonstrated in Fig. 10. 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 be to ignore the relationship between the caller and the node being called, and consider only the time the nodes are called. The latter groups all nodes being called by the same process, and then orders each group of nodes by the time they are called.

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 [35][35]. He et al. proposed a modified random walk on heterogeneous graph in [36]. Yet, all these models are not either designed for, or evaluated on malware detection tasks, and the information of the nodes in these papers does not contain temporal data. Nevertheless, these approaches do produce promising results.

6 Conclusions & Future Work

As with other deep learning approaches, our proposed solution is designed to complement existing well-established methods (e.g. signature-based detection). One of the main reasons for this is the fact that a sandbox environment is required, which means that real-time analysis and protection for every file is impossible. Moreover, not every executable can be activated in a virtualized environment due to anti-virtualization techniques, or the fact that some executable files require human interaction, especially those that are benign.

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

With the continuous increase in the level of threat posed by malware, it is obvious that new and more effective solutions are required. This paper has outlined several key challenges in the field of automated malware detection that need to be addressed.

In order to address these challenges, this paper has outlined two novel contributions that will allow for the identification of malware based largely on their API calls. Firstly, a new method to represent executable file as a multi-edge directional heterogeneous graph was proposed. This allows for the fusion of important data that provides additional context to the behavioral representation, and weightings to quantify the significance of specific behaviors. Secondly, a neural network-based deep learning model was been developed to identify malicious files using the graphs constructed. Our proposed solution was evaluated using a dataset crafted from Cuckoo analysis reports of real-world malware samples. To demonstrate the benefits of our solution, it was evaluated against implementations of other cutting-edge techniques. The results obtained show improved results in both accuracy and the false positive rate. The accuracy of the proposed solution was then compared to existing malware detection engines when identifying 0-day samples. The results showed that although our solution was not the most accurate, it was able to offer

performance that is comparable with leading methods. Despite the promising results, there are limitations with our devised strategy of representing behaviors, which we have discussed in Section 4. For our future work, our primary objective is to overcome these limitations in the hope of further improving upon the results obtainable by our approach.

Declarations

This research is funded by the project "A smart network surveillance system based on artificial intelligence" under Vinh Phuc Province Research Programs (Grant no. 20/ĐTKHVP/2021-2022).

References

[1] Security, S.: Vulnerability and Threat Trends Report 2021 - Cybersecurity Comes of Age

[2] Cybersecurity Statistics During the Spiraling Panic Around COVID-19. https://www.greycampus.com/blog/cybersecurity/covid-cybersecurity-statistics/. Accessed: 2021-06-23

[3] A Not-So-Common Cold: Malware Statistics in 2021. https://dataprot.net/statistics/malware-statistics/. Accessed: 2021-06-24

[4] He, K., Kim, D.-S.: Malware detection with malware images using deep learning techniques. In: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pp. 95–102 (2019). https://doi.org/10.1109/TrustCom/BigDataSE.2019.00022

[5] Bendiab, G., Shiaeles, S., Alruban, A., Kolokotronis, N.: Iot malware network traffic classification using visual representation and deep learning. In: 2020 6th IEEE Conference on Network Softwarization (NetSoft), pp. 444–449 (2020). https://doi.org/10.1109/NetSoft48620.2020.9165381

[6] Kishore, P., Barisal, S.K., Mohapatra, D.P.: An incremental malware detection model for meta-feature api and system call sequence. In: 2020 15th Conference on Computer Science and Information Systems (FedCSIS), pp. 629–638 (2020). https://doi.org/10.15439/2020F73

[7] Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20 (2015). https://doi.org/10.1109/MALWARE.2015.7413680

[8] Wan, T.-L., Ban, T., Cheng, S.-M., Lee, Y.-T., Sun, B., Isawa, R., Takahashi, T., Inoue, D.: Efficient detection and classification of internet-of-things malware based on byte sequences from executable files. IEEE Open Journal of the Computer Society **1**, 262–275 (2020). https://doi.org/10.1109/OJCS.2020.3033974

[9] Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T.: Malware detection with deep neural network using process behavior. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 577–582 (2016). https://doi.org/10.1109/COMPSAC.2016.151

[10] Wüchner, T., Ochoa, M., Pretschner, A.: Malware detection with quantitative data flow graphs. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. ASIA CCS '14, pp. 271–282. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2590296.2590319. https://doi.org/10.1145/2590296.2590319

[11] Hung, N.V., Ngoc Dung, P., Ngoc, T.N., Dinh Phai, V., Shi, Q.: Malware detection based on directed multi-edge dataflow graph representation and convolutional neural network. In: 2019 11th International Conference on Knowledge and Systems Engineering (KSE), pp. 1–5 (2019). https://doi.org/10.1109/KSE.2019.8919284

[12] Yu, B., Fang, Y., Yang, Q., Tang, Y., Liu, L.: A survey of malware behavior description and analysis. Frontiers of Information Technology & Electronic Engineering **19**, 583–603 (2018)

[13] Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: A comprehensive study. IEEE Access **7**, 65889–65912 (2019). https://doi.org/10.1109/ACCESS.2019.2917668

[14] Ding, Y., Dai, W., Yan, S., Zhang, Y.: Control flow-based opcode behavior analysis for malware detection. Computers & Security **44**, 65–74 (2014). https://doi.org/10.1016/j.cose.2014.04.003

[15] Ki, Y., Kim, E., Kim, H.K.: A novel approach to detect malware based on api call sequence analysis. International Journal of Distributed Sensor Networks **11**(6), 659101 (2015) https://arxiv.org/abs/https://doi.org/10.1155/2015/659101. https://doi.org/10.1155/2015/659101

[16] Tran, T.K., Sato, H.: Nlp-based approaches for malware classification from api sequences. In: 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), pp. 101–105 (2017). https://doi.org/10.1109/IESYS.2017.8233569

[17] Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A.: Malware classification with recurrent networks. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1916–1920 (2015). https://doi.org/10.1109/ICASSP.2015.7178304

[18] Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., Eckert, C.: Empowering convolutional networks for malware classification and analysis. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3838–3845 (2017). https://doi.org/10.1109/IJCNN.2017.7966340

[19] Wang, X., Yiu, S.: A multi-task learning model for malware classification with useful file access pattern from API call sequence. CoRR **abs/1610.05945** (2016) https://arxiv.org/abs/1610.05945

[20] Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K.: Android malware detection based on system call sequences and lstm **78**(4), 3979–3999 (2019). https://doi.org/10.1007/s11042-017-5104-0

[21] Homayoun, S., Dehghantanha, A., Ahmadzadeh, M., Hashemi, S., Khayami, R., Choo, K.-K.R., Newton, D.E.: Drthis: Deep ransomware threat hunting and intelligence system at the fog layer. Future Generation Computer Systems **90**, 94–104 (2019). https://doi.org/10.1016/j.future.2018.07.045

[22] Qin, B., Wang, Y., Ma, C.: Api call based ransomware dynamic detection approach using textcnn. In: 2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE), pp. 162–166 (2020). https://doi.org/10.1109/ICBAIE49996.2020.00041

[23] Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. Journal in Computer Virology **7**, 247–258 (2011). https://doi.org/10.1007/s11416-011-0152-x

[24] Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M.: Employing program semantics for malware detection. IEEE Transactions on Information Forensics and Security **10**(12), 2591–2604 (2015). https://doi.org/10.1109/TIFS.2015.2469253

[25] Nikolopoulos, S.D., Polenakis, I.: A graph-based model for malware detection and classification using system-call groups. Journal of Computer Virology and Hacking Techniques **13**, 29–46 (2016)

[26] Ding, Y., Xia, X., Chen, S., Li, Y.: A malware detection method based on family behavior graph. Computers & Security **73**, 73–86 (2018). https://doi.org/10.1016/j.cose.2017.10.007

[27] Zhang, S., Zhou, Z., Li, D., Zhong, Y., Liu, Q., Yang, W., Li, S.:

Attributed heterogeneous graph neural network for malicious domain detection. In: 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 397–403 (2021). https://doi.org/10.1109/CSCWD49262.2021.9437852

[28] Jiang, J., Liu, Z., Yu, M., Li, G., Li, S., Liu, C., Huang, W.: Hetersupervise: Package-level android malware analysis based on heterogeneous graph. In: 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 328–335 (2020). https://doi.org/10.1109/HPCC-SmartCity-DSS50907.2020.00040

[29] Ebad, S.A., Darem, A., Abawajy, J.H.: Measuring software obfuscation quality—a systematic literature review. IEEE Access (2021)

[30] Tu, N.M., Hung, N.V., Anh, P.V., Van Loi, C., Shone, N.: Detecting malware based on dynamic analysis techniques using deep graph learning. In: Dang, T.K., Küng, J., Takizawa, M., Chung, T.M. (eds.) Future Data and Security Engineering, pp. 357–378. Springer, Cham (2020)

[31] Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., Yu, P.S.: Heterogeneous graph attention network. In: The World Wide Web Conference. WWW '19, pp. 2022–2032. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3308558.3313562. https://doi.org/10.1145/3308558.3313562

[32] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems **32**(1), 4–24 (2021). https://doi.org/10.1109/TNNLS.2020.2978386

[33] Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Sun, M.: Graph neural networks: A review of methods and applications. CoRR **abs/1812.08434** (2018) https://arxiv.org/abs/1812.08434

[34] Mathew, J., Ajay Kumara, M.A. : API Call Based Malware Detection Approach Using Recurrent Neural Network—LSTM. Intelligent Systems Design and Applications. 2020, Springer International Publishing, ISBN=978-3-030-16657-1.

[35] 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, pp. 701–710. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2623330.2623732. https://doi.org/10.1145/2623330.2623732

[36] He, Y., Song, Y., Li, J., Ji, C., Peng, J., Peng, H.: Hetespaceywalk: A heterogeneous spacey random walk for heterogeneous information network embedding. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management. CIKM '19, pp. 639–648. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3357384.3358061. https://doi.org/10.1145/3357384.3358061

[37] K. D. T. , Nguyen., T. M. , Tuan, H. , Le, et., al., Comparison of Three Deep Learning-based Approaches for IoT Malware Detection. 10th International Conference on Knowledge and Systems Engneering (KSE) (2018). https://doi.org/10.1109/KSE.2018.8573374