Malware identification on Portable Executable files using Opcodes Sequence

Alexandre R de Mello
Centro de Convergência Digital e Mecatrônica
Fundação CERTI
Florianópolis, Brazil
aem@certi.org.br

Vitor Gama Lemos
Intelligence Lab
PSafe Cyberlabs
Rio de Janeiro, Brazil
0000-0003-1290-5192

Flávio G O Barbosa
Computer Vision Department
SENAI Institute of Innovation in Embedded Systems
Florianópolis, Brazil
flavio.barbosa@sc.senai.br

Emílio Simoni
AHT Security
AHT Security
Rio de Janeiro, Brazil
emilio@ahtsecurity.com

***Abstract*—Malicious software (malware) is a relevant cybersecurity threat, as it can damage target systems, hijack data or credentials, and allow remote code execution. In recent years, researchers and companies have focused on uncovering distinct methods for malware detection to avoid system infection. This paper assesses a method that employs opcode sequence analysis, Graph Theory, and Machine Learning to identify malware on Portable Execution files without the need for execution. An approach used by many researchers is to find patterns through the opcode sequences of a file and use some Artificial Intelligence based strategy to classify the file as malware or benign. In this work, we introduce the OSG (Opcode Sequence Graph), a concept for malware detection based on Opcode Sequence, Graph Theory, and Artificial Intelligence with two new methods: the OSGT (Opcode Sequence Graph Theory) detector and the OSGNN (Opcode Sequence Graph Neural Network). The OSGT extracts the opcode sequence linearly, creates a graph for each file section, calculates features from a combination of Pagerank and node degree of each section, and uses ensemble learning to classify the files. The OSGNN logically extracts the opcode sequence to construct a control flow graph, uses the longest available path to create a graph, and applies a graph neural network to classify the files. We also propose a novel dataset composed of 28,000 files that contain 14,000 updated malware and 14,000 trusted portable executable Windows files. The experimental results show that both methods outperform the baseline methods and provide up to 99% malware detection. The outcomes of this study shows that the OSGT is suitable for real-world application considering the processing time and malware detection capacity, and the OSGNN achieves state-of-art detection capacity for malware with an extra cost of computational cost.**

***Index Terms*—Malware detection, Opcode sequence, Graph theory, Opcode graph, Feature Extraction**

I. Introduction

Recently, many people use the internet in the most diverse ways possible, such as accessing social networks, performing instant banking transactions, and buying online items. For this reason, criminals also act in the virtual world, using technology to commit virtual crimes and spread malware to access confidential data of people and companies. Malware is a software that intentionally executes malicious payloads on victim machines with different goals, such as damaging the targeted system, allowing remote code execution, data hijacking, stealing confidential data, etc [1].

To avoid unwanted malware execution, and of course be a victim of cybernetic attacks, this work proposes a framework to identify malware on Portable Execution (PE) files. The relevance of the proposed method relies on identifying the malware before its executing, i.e., the PE file does not need to be executed to evaluate if the file is a malware or not. We present two different methods that rely on extracting the opcode sequence from files, create a graph that represents the opcode sequence of a file, create features from the graph, and train a binary classifier to identify if a file is trusted or malware. To evaluate the performance on real world scenario, we compare multiple evaluation metrics of the proposed methods against two variations of Long Short-term Memory (LSTM) Recurrent Neural Networks depicted along this paper. The main contribution of this paper are:

- The method to convert the opcode sequence of a PE file into a single or multiple graphs;
- The featurization process based on an opcode sequence and its graph representation;
- Using a graph neural network or an ensemble tree-based method for classification for malware detection;
- Evaluation of processing time on different disassembler methods for real-world usage;
- A real-world dataset;

The proposed work is relevant to both academia and real-world usage as it introduces two different methods for malware detection without the need for file execution, and it explores and compares the usage of a graph representation of opcode sequences for PE files using different disassembling, featurization, and classification methods.

In the malware files investigation, there are two techniques

Identify applicable funding agency here. If none, delete this.

to determine if the software has malicious behavior, such as static and dynamic analysis [2]. While the static method examines a program without executing it, the dynamic analysis must run the malware using virtual machines or a sandbox. In this paper, we explore the static method as a way to parse the portable executable file (PEFile) using the disassembly process.

Disassembling software reveals opcode sequences that are useful for a binary classification between benign and malware. Opcodes are machine instruction codes that specify the CPU to operate internally [3]. Traditional approaches to static detection are signature-based [4], which consists of comparing file bytes against a database of malicious files. Although it is a simple method, it becomes ineffective for malware with new variants and mutations. In these cases, opcode-based approaches are more efficient, as they are concerned with parsing the machine instructions in the file's source code.

Many approaches use opcodes as a feature to create advanced machine learning models for static malware detection [5]–[16]. For opcodes feature extraction the file sample needs to be dissasembled, and it is not trivial considering that an opcode sequence might have an extended length. Thereupon, the opcodes features can be extracted in two ways, using linear order or logical order. In the first case, the opcodes sequence is stored in a one-dimensional array, in which the operators are in the same order as they appear in the source code. In the second case, the opcodes sequence is a control flow graph (CFG), which follows the logical flow of operator instructions execution.

The use of opcode sequences to identify or classify malware is a common technique among many works [5] [6] [7] [8] [9] [10] [11]. More closely related to this work, there are multiple studies that use opcode sequences as a text to apply Natural Language Processing (NLP) on malware detection or classification.

On [17], the authors classify malicious code using continuous bag-of-words model (CBOW) in the word2vec method to vectorize opcodes, a block selection method to reduce analysis overhead, and a text convolutional neural network (text-CNN) to classify the files. The highest accuracy achieved on the Malicous Code Classification Challenge by Microsoft on Kaggle [18] is 98,62%, and for the SOREL-20M dataset [19] the accuracy among all classes varies from 100% to 86,26%.

[20] proposes a semi-supervised approach with deep learning, feature engineering, image transformation, and processing techniques for malware detection. The work extracts features from ASM files (opcode, segment, pixel count, number of lines and characters), applies PageRank methodology on the opcode sequence to select opcodes given scores threshold. The pixel count is used to create grayscale images that represent each file. The authors use an ensemble of three XGboost models trained on different combination of opcode, segment and secondary features, and a CNN model to classify images. The ensemble model achieved an accuracy of 99,12% in the Malicous Code Classification Challenge [18].

The work of [21] presents malware detection with a convolutional recurrent neural network using opcode sequences. The proposed method randomly extracts several opcode sequences, applies a convolutional autoencoder to compress the opcode sequences, and uses a dynamic recurrent neural network to classify the target code. The evaluation was performed in a dataset with 1000 benign and 1000 malware files, collected from Windows 10 Pro System 32 folder and VirusShare.com [22] respectively, and the method achieved an accuracy of 96.2%.

The method proposed by [23] use a word2vec based long short-term memory (LSTM) network to analyze opcodes and API function names. To disassemble the files the authors uses the Interactive Disassembler Pro (www.hex-rays.com/products/ida/) (which is a logical disassembler), but does not provide the average, minimum or maximum time to disassemble each file. The Microsoft Malware Classification Challenge was used to validate the method, achieving 97.59% of accuracy on malware classification, however, there is no analysis on identifying malware against trusted files.

There are also several approaches to detecting malware using graph-based techniques and opcode sequences. [12] proposes the Sequential Opcode Embedding-based Malware Detection (SOEMD). The method is composed of the following phases: i) obtain the opcode sequence instructions using diStorm3 [24] and create an opcode graph; ii) edge selection to reduce the size of the graphs and extraction of short random paths to indicated substructures in the opcode graph; iii) opcode sequence extraction by performing random walk; iv) node selection to select a node subset from each opcode-subgraph; and v) sequence embedding to model the sequence opcode patterns using a skip-gram model. The results show that SOEDM achieved 100% true positive rate in identifying malware or benign classes in the proposed VXHeaven-based dataset.

The work of [14] proposes a method that disassemble portable executable files (using the Distorm3 [24] disassembler) into a sequence of opcode instructions to create a graph that connects each opcode instruction with its neighbors. From the created graph, many sub-graphs are created by removing the edges that connect different opcodes, and a node degree histogram is formed by the combination of the many subgraphs, resulting in the feature vectors that are used to train and evaluate the machine learning models. The authors state that the proposed method achieves a detection accuracy of 98% in the used dataset.

[13] introduces a ransomware detection method based on opcodes and k-nearest neighbors. The work consists in extracting the Control Flow Graph (CFG) from a file to obtain the opcode instructions sequence. To create the feature vector of a file, an N-gram algorithm is used on the opcode instructions with N from 1 to 4, and a K-Nearest Neighbor classifier detects if a file is malware or benign. The best accuracy achieved is 98,86% with N=1.

One study by [15] classifies Android malware using Opcode-level Function Call Graph (FCG) and deep learning.

The method gets the Dalvik code from the Opcode sequences, creates the FCG, and encodes the Opcode sequence from the FCG into a feature vector. The study trains a Long short-term memory (LSTM) based neural network to classify the android malware achieving an accuracy of 97%.

II. The Scientific Method

This work follows the positivist epistemology with a quantitative research using experiments on numerical data as method. We introduce the data collection process in Section VI-A, and the objective is to explore the usage o graph-theory and machine learning on malware detection. Section VII presents an statistical analysis on the experiments defined in Section VI.

III. Proposed method

We introduce the OSG (Opcode Sequence Graph), a methodology to detect malware using Opcode Sequence as input, a Graph theory to create features, and machine learning classifiers. The OSG was designed considering the detection time, feature size, and capacity to detect malware. The detection time is a crucial component to the system because it can not compromise the user experience, i.e., consume a significant amount of memory or spend too much time in processing a single file. The feature size is a relevant factor when making the inference in a cloud infrastructure, as smaller features use fewer data from users and cost less to be uploaded to the cloud. To implement the methods using the OSG methodology, the following steps are necessary:

- Extract the Opcode Sequence
- Create the Opcode Graphs
- Create features from the Opcode Graphs
- Train a machine learning classifier

To further evaluate the proposed method, we implemented two variations of text classifiers based on Long short-term Memory (LSTM) neural networks adopted to the Opcode Sequence context, an C-LSTM [25] and a bi-LSTM with attention using the opcode sequence as text for comparison purposes.

IV. The OSG

The OSG brings the concept of using graph theory to represent a file, hence, from this concept we present two different approaches: i) the OSGT, which create features from multiple graphs created from each file section and use tree-based models to classify, and the ii) OSGNN which uses a graph neural network to classify the control flow graph (CFG).

A. Basic Concepts

We use the following definitions and notations throughout the paper. Consider S a collection of sample files and op_s the opcode sequence of a sample $s \in S$. For each sample s, and opcode graph can be created as $G_s\{V_s, E_s\}$, where V_s is a set of nodes from a finite set of possibilities $P = \{1, \cdot, t\}$ (which t is the number of opcode instructions), E_s is a set of edges that denotes the relation between the nodes, and M_{adj} is the adjacency matrix representation of the graph G_s. To each opcode graph G_s or opcode sequence op_s a feature vector is created, and we define the problem in the feature space in a $n-$dimensional space $\mathcal{R}^n$. An input instance is defined as $\mathbf{x_i} \in \mathcal{R}^n$, and the training set as $D = \{(\mathbf{x_i}, y_i) \| i = 1, 2, \dots, l\}$, where l is the number of instances in the training set. The malware detection is a binary classification problem, hence, we assign a positive label (for benign files) or a negative label (for malware files) $y_i \in \{+1, -1\}$. The training set D is a composition of the aggregation per binary problem $X = [X_+^\top X_-^\top]$, and it denotes all input instances from both classes, where each class is contains $X_\pm = l_\pm \times n$.

B. OSGT

The OSGT is a method that combines characteristics from graph theory, feature engineering, and ensemble classifiers. The proposed creates a graph for each section of the file that contains instructions, creates features that summarize characteristics of the graphs, and uses tree-based models to classify the files as malware or benign. Figure 1 illustrates the general workflow process of the OSGT.

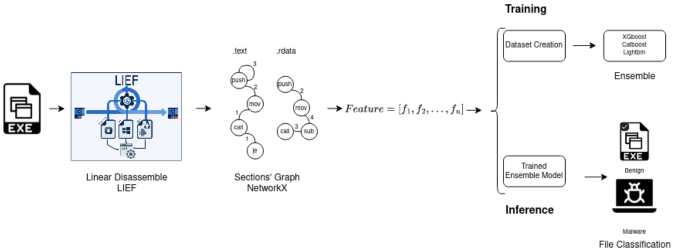


Fig. 1: OSGT workflow

To create the features that define an input instance x_i, we first linearly disassemble the PE file (using the *Library to Instrument Executable Formats* (LIEF) [26]) by disassembling all sections from a file that contains instructions and extracts an Opcode sequences op_s to each section. Second, we create a graph from the op_s generated by assuming that each instruction is a node and the edges are the connection between consecutive instructions, i.e., we start in the position 0 of op_s and iterate in a pairwise sliding window connecting the nodes until the end of the list. Each connection adds 1 to the edge weight, and to avoid creating graphs with different sizes, it only considers opcode instructions from a predefined Opcode list (that contains 931 instructions), resulting in a weighted undirected graph (in the NetworkX library [27] format) that may contain isolated nodes. Figure 2 illustrates the graph creation process.

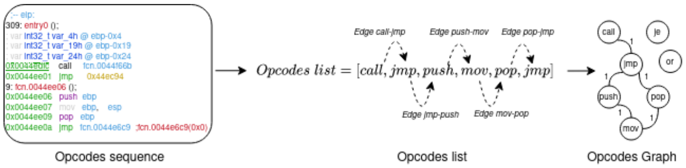

```
;-- eip:
309: entry0 ();
; var int32_t var_4h @ ebp-0x4
; var int32_t var_19h @ ebp-0x19
; var int32_t var_24h @ ebp-0x24
0x0044edfc    call    fcn.0044f66b
0x0044ee01    jmp     0x44ec94
9: fcn.0044ee06 ();
0x0044ee06    push  ebp
0x0044ee07    mov   ebp,   esp
0x0044ee09    pop   ebp
0x0044ee0a    jmp   fcn.0044e6c9  ;fcn.0044e6c9(0x0)
```

Fig. 2: Graph creation process

Each graph is represented as an adjacency matrix M_{adj} to calculate the node degree and Pagerank [28]. The node degree is the number of edges connected to the node and is defined by Equation 1.

$$V_{ND}i = \sum_{j} e_{i,j} \tag{1}$$

where $e_{i,j}$ is an edge between nodes i and j, and $j \in E_s$. The output of the node degree calculation is a vector.

We calculate the Pagerank using the Power iterative approximation method [29], as defined by Equation 2.

$$PR(v) = c \sum_{u \in e_v} \frac{PR(u)}{Num_u} + cE(u) \tag{2}$$

where $PR(v)$ is the Pagerank of the node v, c is the normalized factor, Num_u is the number of links from u, and $E(u)$ is a vector over the graph that corresponds to a source of rank. The Equation 3 presents the Pagerank as a matrix multiplication, so we can describe a Pagerank row vector in terms of the H row-normalized adjacency matrix.

$$\pi^{(k+1)^T} = \pi^{(k)^T} H \tag{3}$$

where π^T is the $1 \times u$ Pagerank row vector, and at the $k-th$ iteration, the successive iterations $\pi^{(k)^T}$ converge to the Pagerank vector π^T, and the the matrix H is defined by Equation 4:

$$H_u = \frac{A_u}{\sum_{k=1}^{n} A_{uk}} \tag{4}$$

where A_u is the adjacency matrix at row u. To guarantee stochasticity and irreducibility of the matrix H, we first define the stochastic S as Equation 5.

$$S = H + \frac{aeA^T}{u} \tag{5}$$

where a is a column vector that $a_u = 1$ if $\sum_{k=1}^{n} H_{uk} = 0$ and 0 otherwise, and e is a column vector of ones. To guarantee that S has a unique stationary distribution vector, S must be irreducible, thus, the graph must be strongly connected. So, we define the irreducible row-stochastic matrix G by Equation 6, also known as the Google matrix.

$$G = \alpha S + (1 - \alpha)E \tag{6}$$

where $0 \neq \alpha \neq 1$ and $E = \frac{ee^T}{Num_u}$. Finally, we define the power method to calculate the Pagerank as Equation 7.

$$\pi^{(k+1)^T} = \pi^{(k)^T} G \tag{7}$$

After calculating both node degree and Pagerank to each node, we apply the Hadamard product (i.e., multiplying element-wise) in both vectors and sum all vectors (nodes), creating a feature vector that represents a section, as presented by Equation 8.

$$Sec_i = \sum_{k=i}^{u} V_{ND}i \circ \pi_i^T \tag{8}$$

After iterating throughout all the sections of a file, we have p features vectors where each vector represents a section of the file that contains executable code. Hence, we can create a feature vector x_i that represents the file by summing all feature vectors Sec_i (Equation 9).

$$x_i = \sum_{k=i}^{p} Sec_i \tag{9}$$

where p is the number of sections of a file that contains executable code, and x_i has a fixed length of u (number of Opcode instructions in the predefined list).

1) The ensemble classifier: We evaluate two ensemble strategies with three different tree-based classifiers to classify the files: an XGBoost [30], a Catboost [31], and a Light Gradient Boosting Machine (LGBM) [32]. We use a major voting scheme, i.e., at least two classifiers must predict the instance as malware; and a consensus scheme, where all classifiers must predict the instance as malware. To select the hyperparameters of each model we use a Bayesian optimization from the Scikit Optimize project with negative mean absolute error from a 5-fold cross validation as objective function and the ROC curve (AUC) as score. We split the training set in an 80-20% ratio (following the Paretto principle [33]) for the training and validation sets (in a stratified manner) respectively, and perform 50 runs to each classification algorithm. Table I presents the best hyperparameters found in each model are:

C. OSGNN

The OSGNN creates a weighted undirected graph G_s to each file by extracting the opcodes instructions following the CFG order and uses the General Graph Neural Networks (GeneralGNN) [34] to classify the files. Figure 3 depicts the overall workflow process for the OSGNN.

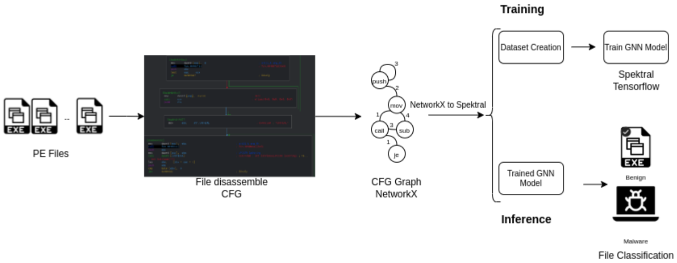


Fig. 3: OSGNN workflow

To create a graph G_s, we extract the Opcode instructions using a logical disassembler that follows the CFG using the R2pipe, which is a python interface to Radare2 [35] (The Libre Unix-Like Reverse Engineering Framework) for disassembler and CFG construction. From the extracted CFG, we select

[0] *https://scikit-optimize.github.io/stable/*

XGboost	Hyperparameters	Catboost	Hyperparameters	Lgbm	Hyperparameters
colsample_bylevel	0.8638616319679682	bagging_temperature	4.57	bagging_fraction	657
colsample_bytree	0.6254544089121741	depth	8	feature_fraction	76
eta	0.24323177277623972	iterations	87	learning_rate	0.541
gamma	1.7603487694545775e-07	l2_leaf_reg	0.117	logloss	1.216
max_delta_step	19	learning_rate	0.936	max_bin	65
max_depth	36			max_depth	0.523
min_child_weight	5			min_child_samples	40
n_estimators	141			n_estimators	0.878
reg_alpha	0.02445570898036228			num_leaves	0.195
subsample	0.9986177687592744			subsample	44

TABLE I: Table with best hyperparameters found to each classifier

the longest available path that starts from the entry section to create a list of Opcodes sequence op_s. To build the graph from the Opcodes sequence, we use the same method as described in Section IV-B and illustrated in Figure 2. For this case, each input instance x_i is a G_s graph.

We select the GeneralGNN model for graph classification, using the Spektral library [36] implementation, with the following characteristics: 256 hidden channels, four message passing layers, two pre-processing layers, two post-processing layers, skip connections with concatenation, batch normalization, no dropout, PReLU activations, sum aggregation in the message-passing layers, and global sum pooling. Table II presents the model summary, and to train the model we define the following parameters:

- Batch size: 32
- Epochs: 25
- Early Stopping: 10
- Learning rate: $1e-3$
- Activation: softmax
- Loss function: Categorical Cross-entropy
- Optimizer: Adam

Model: ”generalGNN		
Layer (type)	Output Shape	#Param
concatenate_1 (Concatenate)	multiple	0
global_sum_pool_1 (GlobalSumPool)	multiple	0
mlp_2 (MLP)	multiple	69632
general_conv_4 (GeneralConv)	multiple	67072
general_conv_5 (GeneralConv)	multiple	132608
general_conv_6 (GeneralConv)	multiple	198144
general_conv_7 (GeneralConv)	multiple	263680
mlp_3 (MLP)	multiple	329738
Total params: 1,060,874		
Trainable params: 1,057,286		
Non-trainable params: 3,588		

TABLE II: Graph classification model summary

V. Other methods

To enhance comparison, we introduce two baseline methods: C-LSTM and a bidirectional LSTM (bi-LSTM) with attention module. For both baseline methods, we create the input features x_i by extracting linearly up to 3000 opcodes instructions from the entry section, iterating throughout the sections.

The Table III describes the C-LSTM model, where the LSTM layers are the Tensorflow Keras implementation.

Model: C-LSTM		
Layer (type)	Output Shape	#Param
embedding_1 (Embedding)	(None, 3000, 500)	1500000
conv1d_3 (Conv1D)	(None, 3000, 128)	192128
max_pooling1d_3 (MaxPooling1D)	(None, 1500, 128)	0
conv1d_4 (Conv1D)	(None, 1500, 64)	24640
max_pooling1d_4 (MaxPooling1D)	(None, 750, 64)	0
conv1d_5 (Conv1D)	(None, 750, 32)	6176
max_pooling1d_5 (MaxPooling1D)	(None, 375, 32)	0
lstm_1 (LSTM)	(None, 200)	186400
dropout_1 (Dropout)	(None, 200)	0
dense_1 (Dense)	(None, 2)	402
Total params: 1,909,746		
Trainable params: 1,909,746		
Non-trainable params: 0		

TABLE III: C-LSTM model summary

Table IV depicts the bi-LSTM with attention module.

Model: bi-LSTM with attention		
Layer (type)	Output Shape	#Param
embedding_1 (Embedding)	(None, 3000, 500)	1500000
conv1d_3 (Conv1D)	(None, 3000, 128)	192128
max_pooling1d_3 (MaxPooling1D)	(None, 1500, 128)	0
conv1d_4 (Conv1D)	(None, 1500, 64)	24640
max_pooling1d_4 (MaxPooling1D)	(None, 750, 64)	0
conv1d_5 (Conv1D)	(None, 750, 32)	6176
max_pooling1d_5 (MaxPooling1D)	(None, 375, 32)	0
bidirectional_1 (Bidirectional)	(None, 375, 400)	372800
dropout_1 (Dropout)	(None, 375, 400)	0
attention_1 (attention)	(None, 400)	775
dense_1 (Dense)	(None, 2)	802
Total params: 2,097,321		
Trainable params: 2,097,321		
Non-trainable params: 0		

TABLE IV: bi-LSTM with attention model summary

VI. Experimental protocol

To the experimental protocol section, we introduce the OSG academic dataset, compare the performance of the previously described methods, and evaluate the results by considering relevant metrics for real-world usage.

A. OSG academic dataset

The OSG academic dataset has 28,000 portable executable files (53,4 GB size) in the *.exe* or *.dll* format, where 14,000

[0] Further information about the LSTM based models can be found at *https://github.com/areeberg/OSG*

	OSGT (s)	OSGNN (s)
Maximum	1.2738	2852.2582
Minimum	0.0007	0.0881
Median	0.0189	0.4733
Mean	0.0246	1.7783

TABLE V: Feature extraction processing time metrics

files are malware and 14,000 are benign files collected from January to March 2022. The malware was downloaded from the Malware Bazaar website *https://bazaar.abuse.ch/*, and the benign files were randomly selected, and individually evaluated, from several computers (with Windows OS) from non-administrative employees To avoid data leakage during tests, i.e., the model does not have access to future samples, the first 10,000 files acquired from each class (malware and benign) are considered the training set, and the last (newest) 4,000 files are the test set.

To encourage reproducibility, we share the hash function (sha256) of each file in the project's repository *https://github.com/areeberg/OSG*.

VII. Results

The feature creation for each file is a critical step for the real use case of the proposed methods. Table V presents the feature creation processing time (in seconds) comparison between OSGT and OSGNN methods, where 100% of the codebase were covered, excluding only the data sections.

Table V shows that the OSGT processing time is substantially faster (especially on large files) compared to OSGNN. Considering the mean value of the feature extraction processing time and 100,000 files, the OSGT would take approximately 41 minutes, and the OSGNN roughly 49 hours.

To evaluate the performance of all methods, we evaluate the proposed methods regarding the accuracy VII, true positive rate (TPR) VII, false positive rate (FPR) VII [1], and malware detection rate (MDR) VII. Table VI shows the results of the detection performance of all methods described in this work.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

$$TPR = \frac{TP}{TP + FN} \tag{11}$$

$$FPR = \frac{FP}{FP + TN} \tag{12}$$

$$MDR = \frac{TN}{FN + TN} \tag{13}$$

	Accuracy(%)	TPR	FPR	MDR
OSGT XGboost	97.32	0.9846	0.0214	0.9846
OSGT LGBM	96.48	0.9803	0.0511	0.9808
OSGT Catboost	96.91	0.9854	0.0471	0.9854
OSGT Consensus Voting	**98.29**	0.9740	**0.0082**	0.9746
OSGT Major Voting	98.24	**0.9910**	0.0259	**0.9910**
OSGNN	97.92	0.9732	0.0350	0.9725
C-LSTM	89.69	0.9423	0.0765	0.8978
bi-LSTM w/ att	92.69	0.8954	0.1330	0.9451

TABLE VI: Detection performances on OSG academic dataset.

where *TP* is true positive, *TN* is true negative, *FP* is the number of false positives (wrongly classified benign file), and *FN* is the number of false negatives (wrongly classified malware).

We analyze the results and summarize them as follows:

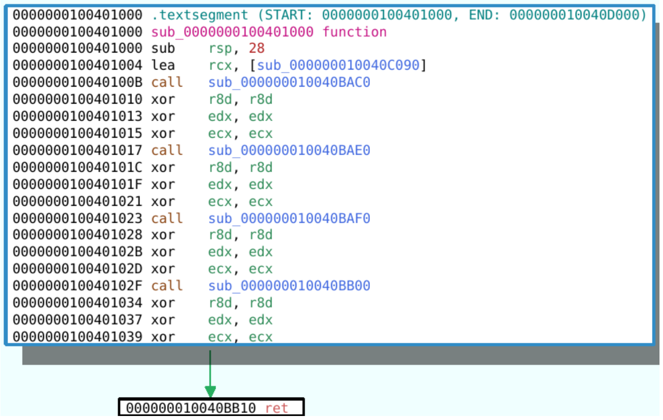
- The Consensus Voting scheme composed of XGboost, LGBM, and Catboost presents the best accuracy with lower FPR, thus less wrongly classified malware which is a critical metric.
- The files that are misclassified contain, commonly, a short and homogeneous opcode sequence, hence, the graph representation only has a few nodes. Fig 4 presents a PE file (sha256 - *a25429265e315df2e3fe02b577a7c9a415c206de228b224483abbf3417fbe0cf*) with low opcode variance (three nodes) and length, whereas Figures 5 and 6 depicts a ”regular” file (sha256 - *bce9a61970f6ac977552a785a4e233fbe76bd492f4428d2507afd4ea393b46fb*) with long and heterogeneous opcodes sequence.
- The usage of ensemble learning appears to be effective in reducing the false positive rate.
- Both LSTM architectures achieve lower accuracy and higher FP and FN compared to the other methods. The reason is that LSTM-based methods use truncation, which is pruned to lose or ignore information when submitted to an extended length of opcode instructions.
- Despite the OSGNN achieving promising results, the feature extraction process takes considerable time, which makes the method impractical for real-world usage. The main contributor to the extended processing time is the CFG construction (especially on large files), as disassembling a file into a CFG is a complex task and requires a logical disassembly.

VIII. Conclusion and Future Work

This paper presents the OSGT and the OSGNN, two methods for static-based malware detection. Both methods combine graph theory with artificial intelligence using the opcode sequences inputs for detecting malware portable executable. The OSGT uses a linear disassembly method to extract the opcode sequence from a file, create a graph from the sequence and calculate the features using the Pagerank and node degree calculations. The classification method is an ensemble method composed of an XGBoost, a Light Gradient Boosting Machine, and a Catboost. The OSGNN uses a logical disassembly

[0] The benign files were evaluated by different methods to ensure that are not malware.

[1] The false-positive rate must be as low as possible to avoid a bad user experience.



```
0000000100401000 .textsegment (START: 0000000100401000, END: 000000010040D000)
0000000100401000 sub_0000000100401000 function
0000000100401000 sub    rsp, 28
0000000100401004 lea    rcx, [sub_000000010040C090]
000000010040100B call   sub_000000010040BAC0
0000000100401010 xor    r8d, r8d
0000000100401013 xor    edx, edx
0000000100401015 xor    ecx, ecx
0000000100401017 call   sub_000000010040BAE0
000000010040101C xor    r8d, r8d
000000010040101F xor    edx, edx
0000000100401021 xor    ecx, ecx
0000000100401023 call   sub_000000010040BAF0
0000000100401028 xor    r8d, r8d
000000010040102B xor    edx, edx
000000010040102D xor    ecx, ecx
000000010040102F call   sub_000000010040BB00
0000000100401034 xor    r8d, r8d
0000000100401037 xor    edx, edx
0000000100401039 xor    ecx, ecx

000000010040BB10 ret
```

Fig. 4: Complete CFG and opcodes sequence.

method to construct the control flow graph from a file and, using the longest available path from the entry section, It extracts the opcode sequence to create a weighted undirected graph. To perform the classification, it uses a graph neural network (using the Spektral library, hence Tensorflow) as the framework. Both proposed methods handle the challenge of processing long opcode sequences without losing information in the learning process. Another strong aspect of the proposed method is that it is scalable for extended opcode sequences, as the set of opcodes is predefined, so there is a maximum number of nodes to each graph.

We propose a dataset of real-world files, i.e., some files are auxiliary functions (especially *.dlls*) that do not have all fields from a regular portable executable structure, and recent malware. The results show that OSGNN provides 99% on malware detection, and the OSGT has the best accuracy overall with a fast feature extraction process.

For future work, we will evaluate including new opcode instructions for the graph creation, expanding the dataset with new malware and benign files, exploring new logical disassembly strategies, and new graph neural network architectures for classification.

Acknowledgment

I would like to express gratitude to *PSafe Cyberlabs* for their support towards making this research possible, and to *Fundação CERTI*, *SENAI Institute of Innovation in Embedded Systems*, and *AHT Security* for their support on continuing the research.

References

[1] O. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[2] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. USA: No Starch Press, 2012.

[3] F. R. A. Hopgood, "Assemblers and loaders. by d. w. barron. pp. vi, 61. 25s. 1969. (macdonald.)," *The Mathematical Gazette*, vol. 54, no. 389, p. 320–320, 1970.

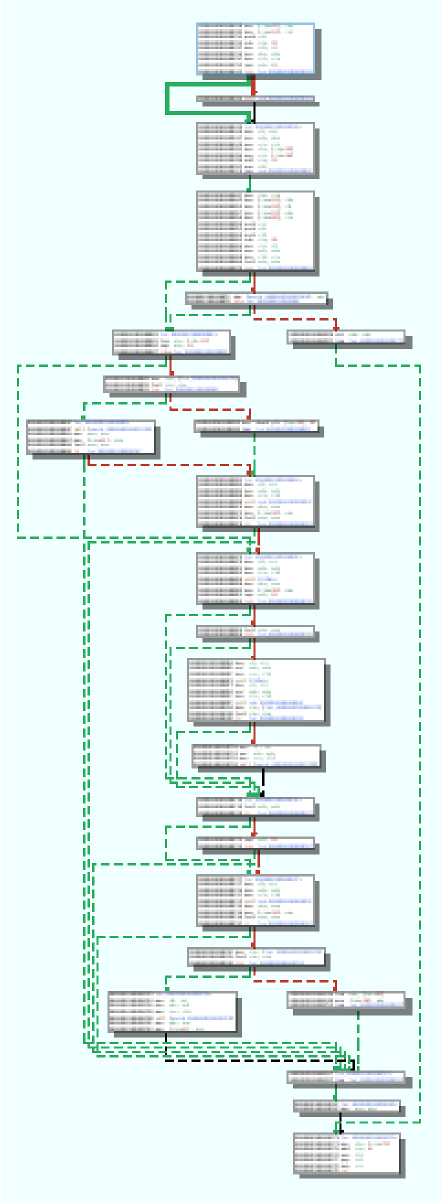


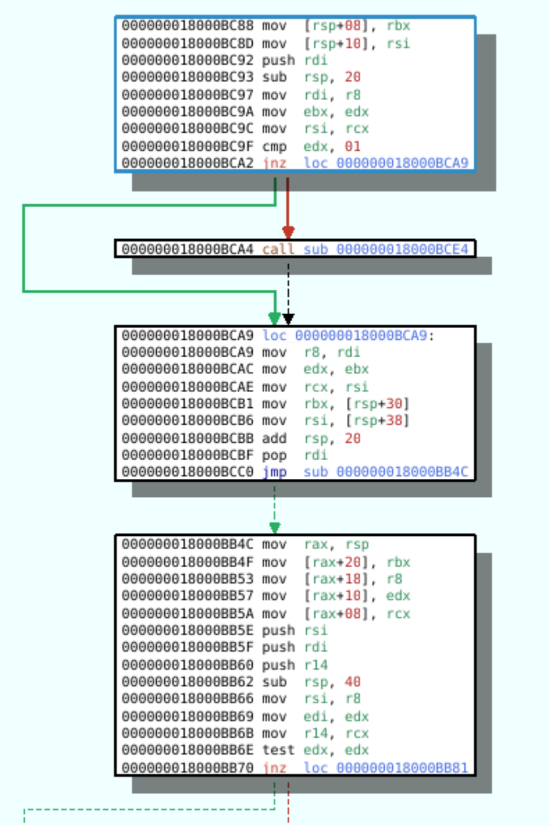
Fig. 5: The overview of a regular Portable Executable file - Complete CFG.

[4] M. Siddiqui, M. C. Wang, and J. Lee, "A survey of data mining techniques for malware detection using file features," in *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ser. ACM-SE 46. New York, NY, USA: Association for Computing Machinery, 2008, p. 509–510. [Online]. Available: https://doi.org/10.1145/1593105.1593239

[5] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5965 LNCS. Springer, Berlin, Heidelberg, 2010, pp. 35–43.

[6] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, may 2013.

[7] J. Gaviria de la Puerta and B. Sanz, "Using Dalvik opcodes for malware detection on android," *Logic Journal of the IGPL*, vol. 25, no. 6, pp. 938–948, dec 2017. [Online]. Available: http://academic.oup.com/jigpal/article/25/6/938/4389322



```
000000018000BC88 mov  [rsp+08], rbx
000000018000BC8D mov  [rsp+10], rsi
000000018000BC92 push rdi
000000018000BC93 sub  rsp, 20
000000018000BC97 mov  rdi, r8
000000018000BC9A mov  ebx, edx
000000018000BC9C mov  rsi, rcx
000000018000BC9F cmp  edx, 01
000000018000BCA2 jnz  loc_000000018000BCA9
```

```
000000018000BCA4 call sub_000000018000BCE4
```

```
000000018000BCA9 loc_000000018000BCA9:
000000018000BCA9 mov  r8, rdi
000000018000BCAC mov  edx, ebx
000000018000BCAE mov  rcx, rsi
000000018000BCB1 mov  rbx, [rsp+30]
000000018000BCB6 mov  rsi, [rsp+38]
000000018000BCBB add  rsp, 20
000000018000BCBF pop  rdi
000000018000BCC0 jmp  sub_000000018000BB4C
```

```
000000018000BB4C mov  rax, rsp
000000018000BB4F mov  [rax+20], rbx
000000018000BB53 mov  [rax+18], r8
000000018000BB57 mov  [rax+10], edx
000000018000BB5A mov  [rax+08], rcx
000000018000BB5E push rsi
000000018000BB5F push rdi
000000018000BB60 push r14
000000018000BB62 sub  rsp, 40
000000018000BB66 mov  rsi, r8
000000018000BB69 mov  edi, edx
000000018000BB6B mov  r14, rcx
000000018000BB6E test edx, edx
000000018000BB70 jnz  loc_000000018000BB81
```

Fig. 6: A regular Portable Executable file - The first four blocks of opcode sequences.

[8] A. Yewale and M. Singh, "Malware detection based on opcode frequency," in *Proceedings of 2016 International Conference on Advanced Communication Control and Computing Technologies, ICACCCT 2016*. Institute of Electrical and Electronics Engineers Inc., jan 2017, pp. 646–649.

[9] Y. M. Kwon, J. J. An, M. J. Lim, S. Cho, and W. M. Gal, "Malware classification using simhash encoding and PCA (MCSP)," *Symmetry*, vol. 12, no. 5, p. 830, may 2020.

[10] H. Darabian, A. Dehghantanha, S. Hashemi, S. Homayoun, and K. R. Choo, "An opcode-based technique for polymorphic Internet of Things malware detection," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 6, p. e5173, mar 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.5173

[11] H. Bai, N. Xie, X. Di, and Q. Ye, "FAMD: A fast multifeature android malware detection framework, design, and implementation," *IEEE Access*, vol. 8, pp. 194 729–194 740, 2020.

[12] A. G. Kakisim, S. Gulmez, and I. Sogukpinar, "Sequential opcode embedding-based malware detection method," *Computers and Electrical Engineering*, vol. 98, p. 107703, mar 2022.

[13] D. Stiawan, S. M. Daely, A. Heryanto, N. Afifah, M. Y. Idris, and R. Budiarto, "Ransomware detection based on opcode behaviour using k-nearest neighbours algorithm," *Information Technology and Control*, vol. 50, no. 3, pp. 495–506, sep 2021.

[14] S. Gulmez and I. Sogukpinar, "Graph-based malware detection using opcode sequences," in *9th International Symposium on Digital Forensics and Security, ISDFS 2021*. Institute of Electrical and Electronics Engineers Inc., jun 2021, pp. 1–5.

[15] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, "OpCode-Level Function Call Graph Based Android Malware Classification Using Deep Learning," *Sensors*, vol. 20, no. 13, p. 3645, jun 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/13/3645

[16] Z. Sun, Z. Rao, J. Chen, R. Xu, D. He, H. Yang, and J. Liu, "An opcode sequences analysis method for unknown malware detection," in *Proceedings of the 2019 2nd International Conference on Geoinformatics and Data Analysis*, ser. ICGDA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–19. [Online]. Available: https://doi.org/10.1145/3318236.3318255

[17] Q. Wang and Q. Qian, "Malicious code classification based on opcode sequences and textCNN network," *Journal of Information Security and Applications*, vol. 67, p. 103151, jun 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2214212622000412

[18] "Microsoft malware classification challenge (big2015)," 2015. [Online]. Available: https://www.kaggle.com/c/malware-classification/data

[19] R. Harang and E. M. Rudd, "SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection," *arXiv*, dec 2020. [Online]. Available: http://arxiv.org/abs/2012.07634

[20] A. Darem, J. Abawajy, A. Makkar, A. Alhashmi, and S. Alanazi, "Visualization and deep-learning-based malware variant detection using OpCode-level features," *Future Generation Computer Systems*, vol. 125, pp. 314–323, dec 2021.

[21] S. Jeon and J. Moon, "Malware-Detection Method with a Convolutional Recurrent Neural Network Using Opcode Sequences," *Information Sciences*, vol. 535, pp. 1–15, oct 2020.

[22] VirusShare.com. (2022) Virusshare.com-because sharing is caring. [Online]. Available: https://virusshare.com/

[23] J. Kang, S. Jang, S. Li, Y. S. Jeong, and Y. Sung, "Long short-term memory-based malware classification method for information security," *Computers and Electrical Engineering*, vol. 77, pp. 366–375, 7 2019.

[24] G. Dabah, "distorm3," Dec. 2020. [Online]. Available: https://github.com/gdabah/distorm/

[25] C. Zhou, C. Sun, Z. Liu, and F. C. M. Lau, "A c-lstm neural network for text classification," 2015. [Online]. Available: https://arxiv.org/abs/1511.08630

[26] R. Thomas, "Lief - library to instrument executable formats," https://lief.quarkslab.com/, apr 2017.

[27] D. A. Schult, "Exploring network structure, dynamics, and function using networkx," in *In Proceedings of the 7th Python in Science Conference (SciPy)*, 2008.

[28] T. Haveliwala, "Efficient computation of pagerank," 1999.

[29] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, and G. Golub, "Computing pagerank using power extrapolation," Stanford InfoLab, Technical Report 2003-45, 2003. [Online]. Available: http://ilpubs.stanford.edu:8090/605/

[30] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD 16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[31] A. V. Dorogush, V. Ershov, and A. Gulin, "Catboost: gradient boosting with categorical features support." *CoRR*, vol. abs/1810.11363, 2018.

[32] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[33] V. R. Joseph, "Optimal ratio for data splitting," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 15, pp. 531–538, 8 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/sam.11583

[34] J. You, R. Ying, and J. Leskovec, "Design space for graph neural networks," pp. 17 009–17 021, 2020. [Online]. Available: https://github.com/snap-stanford/graphgym

[35] radareorg, "radare2," dec 2015. [Online]. Available: https://rada.re/n/

[36] D. Grattarola and C. Alippi, "Graph neural networks in tensorflow and keras with spektral [application notes]," *IEEE Computational Intelligence Magazine*, vol. 16, pp. 99–106, 2 2021.