CFGExplainer: Explaining Graph Neural Network-Based Malware Classification from Control Flow Graphs

Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, Guanhua Yan
State University of New York at Binghamton, Binghamton, NY, USA

***Abstract*—With the ever increasing threat of malware, extensive research effort has been put on applying Deep Learning for malware classification tasks. Graph Neural Networks (GNNs) that process malware as Control Flow Graphs (CFGs) have shown great promise for malware classification. However, these models are viewed as black-boxes, which makes it hard to validate and identify malicious patterns. To that end, we propose CFGExplainer, a deep learning based model for interpreting GNN-oriented malware classification results. CFGExplainer identifies a subgraph of the malware CFG that contributes most towards classification and provides insight into importance of the nodes (i.e., basic blocks) within it. To the best of our knowledge, CFGExplainer is the first work that explains GNN-based malware classification. We compared CFGExplainer against three explainers, namely GNNExplainer, SubgraphX and PGExplainer, and showed that CFGExplainer is able to identify top equisized subgraphs with higher classification accuracy than the other three models.**

I. Introduction

Malicious software (or malware) is any software that performs harmful actions on computer systems. Malware can be in the form of viruses, worms, trojan horses, and so on. According to the report by AV-TEST [1], there are over 1228 million malware in the world as of 2021 [1], which is roughly 12 times more than in the year of 2012. To cope with this rapidly evolving and increasing malware threat, extensive research efforts have been put on automating malware detection and classification tasks. Among these research efforts, Machine Learning (ML), in particular Deep Learning (DL), is a promising approach for identifying and classifying malware [2], [3], [4], [5], [6]. There are a variety of DL-based solutions, including models inspired by success in computer vision [7], [8], sequence models [9], [10], and more recently graph based models [11], [12], [13], [14] that process malware as Control Flow Graphs (CFGs).

Graph Neural Network (GNN) based solutions [11], [12], [15] have shown great promise in malware classification due to their ability to not only process block level features but also handle topological relationships across nodes (i.e., basic blocks) in graphs. However, like many Deep Neural Network models, GNN-based CFG classifiers are treated as black boxes as they do not provide insight into reasons behind malware classification, which makes it hard for malware analysts to verify classification results and identify malicious patterns. This leads to the growing need for solutions that explain the classification results made by the GNN models.

Recently, there have been efforts in explaining GNN-based classification results [16], [17], [18] by identifying graph structures that contribute most towards classification. These models provide explanations in forms of a subgraph pruned out of the original graph that leads to high classification accuracy. However, these approaches do not provide insight into the contribution of each node in the subgraph (i.e., basic blocks in CFG) towards classification and hence malware analysts may need to analyze all nodes in it to identify malicious behaviors. When the size of the subgraph is large, this could be an exhaustive task. While there has been a recent effort in developing an interpretability solution for malware classification [19], it is not applicable in explaining GNN-based malware classification based on graph structure. To the best of our knowledge, there is no previous work specifically focused on explaining GNN-based malware classification.

Against this backdrop, we develop CFGExplainer, a deep learning based graph explainer for interpreting malware classification results. CFGExplainer identifies a subgraph of a malware CFG that contributes most towards malware classification and provides insight into the importance of the nodes (i.e., basic blocks) within the subgraph. CFGExplainer helps malware analysts zoom in on the most important blocks of code and view any topological relationships in the graph at the same time. It would also alleviate the complexity of identifying malicious patterns when used in tandem with tools such as IDA-Pro [20] and Ghidra [21].

To circumvent the issue of searching through a large number of subgraph combinations, CFGExplainer initially finds node importance with respect to its perceived contribution towards malware classification and then uses it to prune and identify an important subgraph. Taking inspirations from the attention mechanisms, our proposed method defines a supervised learning task that uses two inter-connected feed-forwarding deep neural networks. The first one learns scores for node embeddings produced from the target GNN, whose internal structure is treated as a blackbox. The second component weights the original node embeddings with these scores and uses them to train a surrogate malware classification model. As these two inter-connected neural networks are jointly trained with a negative log-likelihood loss function, the scores learned from the first model can boost the contribution of important node embeddings to malware classification in the second one. Unlike other attention mechanisms where interpretability of

attention weights may be ambiguous [22], the node scores generated in our method directly capture the importance of node embeddings to the classification task.

We have compared CFGExplainer against three state-of-the-art GNN-oriented explainers, namely GNNExplainer [16], SubgraphX [18] and PGExplainer [17], using eleven malware families (Bagle, Bifrose, Hupigon, Ldpinch, Lmir, Rbot, Sdbot, Swizzor, Vundo, Zbot and Zlob) and one benign family. Our experimental results show that CFGExplainer is able to identify top subgraphs of the same size that lead to higher classification accuracy than GNNExplainer, SubgraphX and PGExplainer. For example, the subgraph constructed from the top 20% nodes identified by CFGExplainer leads to around 70% classification accuracy on average, which is 4.2 times higher than GNNExplainer, 3.6 times higher than SubgraphX, and 2 times higher than PGExplainer.

In a nutshell, our contributions are summarized as follows:

- We propose CFGExplainer, a deep learning based model-agnostic explainer, for explaining CFG-based malware classification. CFGExplainer produces a subgraph that contributes most towards the classification task and provides the insight into nodes in the subgraph that are deemed useful for malware classification. To the best of our knowledge, CFGExplainer is the first work that explains GNN-based malware classification.
- We have compared the classification accuracy of subgraphs produced by CFGExplainer against three state of the art graph explainers GNNExplainer, SubgraphX and PGExplainer. Our experimental results show that CFGExplainer is able to identify top equisized subgraphs with higher classification accuracy than the other three models.
- We have analyzed subgraphs containing top 20% of nodes of malware samples produced by CFGExplainer and offered insights into malware patterns that would be useful for further human examination.

The rest of the paper is organized as follows. Section II provides a brief overview of Attributed Control Flow Graphs, Graph Neural Networks, and the interpretability models used in our evaluation. Section III provides the problem formulation. Section IV presents the architecture and algorithmic details of CFGExplainer. Our experimental results are given in Section V. Section VI gives the related work and Section VII draws the concluding remarks.

II. Background

In this section, we provide a brief overview of Attributed Control Flow Graphs (ACFGs) used for malware classification, Graph Neural Networks (GNNs), and three graph based interpretability models used in our evaluations, namely GNNExplainer [16], SubgraphX [18] and PGExplainer [17].

A. Attributed Control Flow Graphs

An Attributed Control Flow Graph (ACFG) is a Control Flow Graph (CFG) [23] associated with node attributes. In an ACFG [11], a vertex or node represents a basic block that contains a sequence of assembly instructions. A directed edge between two nodes (u, v) in the ACFG represents that either the last instruction in node u falls through to the very first instruction in node v, or there is a jump or call instruction in u that leads to instructions in node v.

We follow the method in [11] to obtain node attributes for ACFGs due to its prior success in achieving high classification results. As shown in Table I, we consider 12 node features generated from the code sequence in the block and the node structure. These features are generated based on the number of assembly instructions categorized into different types (e.g., #numeric constants, #mov instructions) as well as the degree and the number of total instructions in a vertex. Let $G = (V, E)$ be an ACFG where V is the set of nodes and $E = V \times V$ the set of edges. The graph can be described as a weighted adjacency matrix $A \in \{0, 1, 2\}^{N \times N}$, where N is the maximum number of nodes in the graph. If the code naturally flows from block i to j or if it is a jump instruction, then $A_{ij} = 1$. If it is a call instruction $A_{ij} = 2$, otherwise $A_{ij} = 0$. Nodes in V are associated with d-dimensional features X ($d = 12$ in our work), denoted by $X \in \mathbb{R}^{N \times d}$ where $\mathbb{R}$ represents a set of real numbers.

TABLE I
BLOCK LEVEL FEATURE DESCRIPTION.

Feature Type	Feature Description
Generated from code sequence	# Numeric constants # String constants # Transfer instructions # Call instructions # Arithmetic instructions # Compare instructions # Mov instructions # Termination instructions # Data declaration instructions # Total instructions
Generated from node structure	# Offspring (The degree) # Instructions in the vertex

B. Graph Neural Networks

A graph neural network (GNN) model typically operates in two steps. First, the GNN model processes the graph and generates node embeddings, which are low-dimensional vector representations of nodes generated by considering both the graph structure and node/edge features. Next, the GNN model processes embeddings for the node/graph classification task. A GNN model Φ can be divided into two sub-models $\{\Phi_e, \Phi_c\}$. Φ_e produces f-dimensional node embeddings (denoted by $Z \in \mathbb{R}^{N \times f}_{\geq 0}$) based on the adjacency matrix A and node features $\bar{X}$. Φ_e can be constructed using Recurrent Neural Networks (RNNs) (e.g., [24]) or by combining Graph Convolutional Network (GCN) layers with additional pooling layers (e.g., [11], [12], [13]) Φ_c is typically a feed forwarding neural network that acts as a classifier which processes node embeddings Z.

C. Interpretability Models for Graph Neural Networks

In this paper, we compare the performance of CFGExplainer against that of GNNExplainer [16], SubgraphX [18] and

PGExplainer [17], three state-of-the-art interpretability models that operate on graph neural networks. These explainers are post-hoc models that aim to provide explanations on the classification result produced by a pre-trained graph classification model. The explanation takes the form of a subgraph of the original graph. These models provide explanations for each graph individually.

GNNExplainer provides interpretations in the form of a subgraph that is pruned out of the original graph. To do so, GNNExplainer learns a mask which takes the form of a $[N, N]$ matrix for a graph with N nodes that can be element-wise multiplied with the adjacency matrix of the original graph. The resulting matrix can be used as scores to order the edges in the graph. For every graph to be interpreted, edges with smallest scores are removed. GNNExplainer learns to generate these masks as an optimization task. The masks are optimized by maximizing the mutual information between the classification of the original graph and the classification of the newly obtained subgraph (i.e., the subgraph generated after applying the mask). The pre-trained graph classifier is probed iteratively to identify the masks. Unlike CFGExplainer, GNNExplainer needs to optimize the masks for each input graph individually and hence does not leverage any global information that may be present across all the graphs.

SubgraphX uses Monte Carlo Tree Search (MCTS) [25] to explore different subgraph combinations and identify a subgraph that contributes most towards classification. The nodes in the tree correspond to subgraphs pruned out of the original graph. A child in the tree is obtained by performing node-pruning on the subgraph associated with the parent in the search tree. The reward for MCTS is generated using Shapley values [26], [27], which is a game theoretic approach for fairly assigning gains to different game players. For this explanation task, the pre-trained GNN predictions are used as the game gain and different graph structures are considered as players. A subgraph structure with a higher Shapley value score indicates higher importance towards classification. Similar to GNNExplainer, SubgraphX is a local explainability model where it needs to employ MCTS for each ACFG individually.

PGExplainer provides interpretations in the form of a subgraph. PGExplainer learns an approximated discrete mask for edges that explain the graph classification. It trains a mask predictor, which is a generative deep neural network [28], to generate edge masks. Unlike GNNExplainer and SubgraphX, PGExplainer leverages a global view of the graphs. During its training phase, it obtains the embeddings for each edge by concatenating the embeddings of the two nodes that form the edge. The mask predictor then uses the edge embeddings to predict the probability of each edge being useful for the classification task. Afterwards, new graphs are sampled based on the result of the mask predictor and are trained to maximize the mutual information between original classifications and classifications made for sampled graphs. Again, the classifications are made with respect to a pre-trained graph classifier. During the subgraph identification, the output from the deep learning model is used as a score to prune graphs.

III. Problem Formulation

The main objective of our work is to provide useful interpretations as to why an ACFG is classified as a specific malware family by the GNN model. Our interpretation separates an ACFG G into two subgraphs G_s and ΔG (i.e., $G = G_s + \Delta G$) similar to works in [16], [17], where G_s is the subgraph that makes important contribution towards malware classification and ΔG the subgraph containing the rest of the nodes. Given this setting, we break down the interpretability problem into two sub-problems.

Sub-problem 1: Identifying the subgraph G_s that makes important contribution towards the malware classification. Identifying G_s would enable malware analysts to understand the topological relationship between nodes (i.e., blocks of code). We quantify the usefulness of these interpretations by evaluating the classification accuracy of the identified subgraph using a pre-trained GNN classification model. If the identified subgraph makes important contribution towards malware classification, then the difference between the classification accuracy of the identified subgraph and the original graph should be small.

Sub-problem 2: Identifying nodes that contain malware behaviors by ordering the nodes based on their importance on the classification task. This would enable malware analysts to study the most important nodes to identify malware behaviors.

IV. Design of CFGExplainer

Figure 1 gives the operational pipeline of CFGExplainer. CFGExplainer is designed as a post-hoc model for interpreting malware classification results. First, a GNN-based malware classifier is used to obtain the classification result and node embeddings based on the malware ACFGs. CFGExplainer then processes the embeddings and class labels to interpret the results.

For each ACFG to be interpreted, CFGExplainer produces two outputs: (1) a set of nodes in the ACFG ordered based on their importance with respect to classification, and (2) a set of subgraphs constructed based on the node orderings and a user defined step size. The step size enables an analyst to control the size of the subgraph he/she wishes to analyze for malware behaviour. For example, if the step size is 10%, then the subgraphs contain 10%, 20%, ..., 100% nodes, and the smallest subgraph is constructed from the 10% nodes that contribute most towards malware classification. The step size needs to be carefully chosen, as using a large step size would result in large subgraphs while using a small one would increase the time in finding subgraphs. By analyzing the output of CFGExplainer, malware analysts would be able to identify the most important code blocks (i.e., the nodes) and topological connections, and their growth as the graph increases in size. Like other explainers considered in this work [16], [17], [18], CFGExplainer explains the prediction by a particular classifier on a particular sample. Different GNN classifiers may assign importance scores to different nodes for the same prediction result, and hence the subgraphs generated may be different. In addition, the subgraphs generated may be

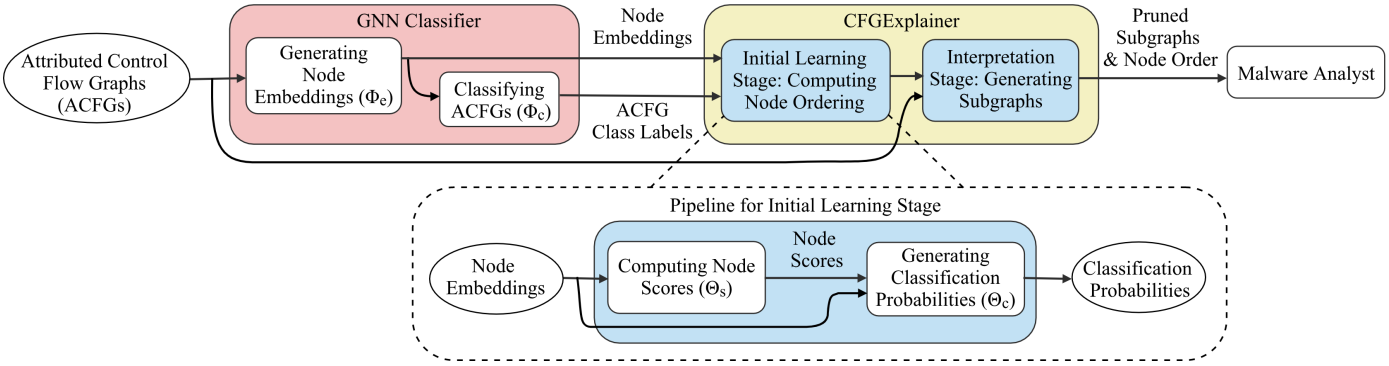

Fig. 1. The operational pipeline of CFGExplainer.

disjoint depending on what edges are considered important by the corresponding explainer.

One main challenge in obtaining a useful subgraph is the large number of subgraph combinations. For a graph with $|E|$ edges, the total number of subgraph combinations is exponentially large (i.e., $2^{|E|}$). Searching through all these combinations can be a daunting task. CFGExplainer tackles this issue by breaking down the process into two main stages: an initial learning stage and an interpretation stage.

In the initial learning stage, CFGExplainer trains a deep learning model that learns to generate node scores based on node embeddings. A large score on a node entails that the node is important for malware classification. As shown in Figure 1, the input to the initial learning stage is the node embeddings generated from the GNN. For a GNN with reasonable classification capability, the node embeddings can be considered as a latent representation of the nodes that are useful for classification. By directly using node embeddings, CFGExplainer need not re-learn any information in the graphs from scratch. This enables the use of simple model architectures such as feed-forwarding neural networks to construct CFGExplainer as opposed to more complex model variants. Using node embedding also ensures that CFGExplainer is model agnostic, and hence does not need any internal parameters of the classification model for its learning task. Additionally, using node embeddings reduces the space complexity of computation. For a graph with N nodes and for embedding size f, then the input would be a matrix of size $[N, f]$ at all times. This is in contrast to PGExplainer [17] that uses edge embeddings, where the deep learning model could require a constructed input of size $[N^2, 2f]$. In addition, the deep learning model is trained using node embeddings of a set of different malware families, which enables the model to better discriminate between different malware families while at the same time learning similarities within the same malware family. This provides CFGExplainer the ability to better generalize when producing interpretations later on. This is in contrast to a purely local search method like GNNExplainer [16] or SubgraphX [18], where interpretations for each graph must be generated without leveraging any global knowledge.

Once CFGExplainer sufficiently learns to generate node scores, in the interpretation stage, the deep learning model is probed iteratively based on a user defined step size to obtain the node ordering and the subgraphs. The trained model is used as a surrogate model to directly determine what nodes to prune at each step, thereby circumventing the exploration problem stemming from the large number of possible subgraph combinations. Below, we provide detailed algorithms for the learning and interpretation stages.

A. The Initial Learning Stage

In the initial training stage, a deep learning model (denoted by Θ) is trained to rank nodes in ACFGs based on their importance towards the classification task. The inputs to Θ are node embeddings and ACFG class labels (i.e., Bagel, Bifrose, etc.) generated by a GNN classifier $\Phi = \{\Phi_e, \Phi_c\}$, where Φ_e represents the node embedding generation component and Φ_c represents the classification component. As shown in Figure 1, Θ consists of two feed forwarding neural networks: Θ_s and Θ_c. Here, Θ_s is the node scoring component that obtains node embeddings Z produced by Φ_e and generates a score for each node $\Psi \in [0,1]^{1\times N}$. A large score on a node (i.e., $\Psi_i \approx 1$) indicates that the node is important for the classification task and a low score indicates otherwise. After generating Ψ, the original node embeddings are multiplied by the individual scores in Ψ to obtain weighted node embeddings $Z_{weighted}$. These weighted node embeddings are subsequently processed by the second component Θ_c to generate a vector of classification probabilities Y across all ACFG families, where $Y[C]$ is the classification probability associated with the class label C as identified by the GNN model Φ. Although having the score as a separate input to Θ_c may provide freedom to find a better classification model, it does not help learn the weights of node embeddings. In our model, weights are tied to embeddings, which helps to identify important nodes.

The objective of the initial learning stage is to identify important nodes with respect to malware classification. However, it is challenging to generate a labeled dataset for node importance as it requires manual analysis of all nodes. A connected model $\Theta = \{\Theta_s, \Theta_c\}$ with negative log-likelihood loss (i.e., $loss = -\frac{1}{m}\sum_{\forall i=1:m} log(Y[C_i])$ for m ACFGs) circumvents the need to obtain labeled data with node impor-

tance for all the ACFGs. Due to the architectural connectivity between Θ_c and Θ_s, minimizing this loss through an optimizer such as Adam [29] leads to a joint training procedure, where back propagation updates weights in both neural networks. Note that $log(Y[C_i])$ would induce an undefined result if $Y[C_i] == 0$. To avoid this, we use a small positive bias when computing the log value (i.e., $log(Y[C_i] + 10^{-20})$) in our implementation. When Θ_c learns to better predict the ACFG class based on $Z_{weighted}$, it also results in Θ_s learning to predict higher scores for node embeddings that have higher contribution towards classification. Here, we train a new classifier Θ_c instead of Φ_c, because otherwise, training of CFGExplainer would change the internal weights of the GNN, subsequently changing the embeddings Z. For a given ACFG, the embeddings Z generated by the GNN needs to be fixed, so that any score learnt by Θ_s does in fact reflect node importance. Although theoretically the node scores Ψ could be the same when node embeddings are the same, it is highly unlikely to happen in practice due to the complex graph structures of ACFGs. Algorithm 1 gives the detailed steps of the initial learning stage.

TABLE II
SYMBOL TABLE.

Symbol	Explanation
G	The ACFG $G = (V, E)$ where V is a set of nodes and $E = V \times V$ a set of edges
N	The maximum number of nodes in the ACFG
A	The weighted adjacency matrix $A \in \{0, 1, 2\}^{N \times N}$
X	The feature matrix $X \in \mathbb{R}^{N \times d}$
d	The number of features per node
C	The class label of the ACFG
Φ	The pre-trained GNN classification model $\Phi = \{\Phi_e, \Phi_c\}$
Φ_e	The node embedding generation component of Φ
Φ_c	The classification component of Φ
Z	Node embeddings generated by Φ_e ($Z \in \mathbb{R}^{N \times f}_{\geq 0}$)
f	The size of the embeddings per node
Θ	The DL model of CFGExplainer ($\Theta = \{\Theta_s, \Theta_c\}$)
Θ_s	The node scoring component of CFGExplainer
Θ_c	The classification component of CFGExplainer
Ψ	The node scores generated by Θ_s
Y	The classification probabilities predicted by Θ_c
M	The #training samples used by CFGExplainer
m	The mini-batch size used in training CFGExplainer

In Algorithm 1, CFGExplainer takes as input M ACFG samples, denoted by $\mathbf{D} = \{\{A_1, X_1\}, \{A_2, X_2\}, ..., \{A_M, X_M\}\}$. Each sample $\{A_i, X_i\}$ is constructed from the adjacency matrix A_i associated with graph G_i and the feature matrix X_i. In Line 1, the algorithm randomly initializes the weights of Θ_s and Θ_c. For each training $epoch \leq num\_epoch$, the neural network components Θ_s and Θ_c are jointly trained as follows. First, a mini-batch D' of $m < M$ training samples are randomly selected from $\mathbf{D}$ and the initial training $loss$ is set as 0 (lines 3-4). In line 6, node embeddings of all the nodes (denoted by Z_i) are obtained from the node embedding generation component Φ_e of the GNN model, using the adjacency matrix A_i and the feature matrix X_i. The classification component Φ_c then processes the node embeddings Z_i to predict the class label C_i (line 7).

Next, the node scoring component Θ_s computes scores Ψ for each node (line 8). Each of the original embeddings are weighted according to Ψ to obtain $Z_{weighted}$ (lines 9-11). $Z_{weighted}$ is then used by the classification component Θ_c to generate class probabilities Y for the malware (line 12). The negative log-likelihood loss value is computed in lines 13 and 14 for malware class C_i predicted by the GNN model. The weights of both Θ_s and Θ_c are then updated using the Adam optimizer (line 15). Once the training is complete, Θ_s is used as a surrogate to identify subgraphs that contribute most towards the malware classification.

Procedure `Training`

1 Randomly initialize the weights of DNN models Θ_s and Θ_c
2 **for** $epoch = 1$; $epoch \leq num\_epoch$, $epoch$++ **do**
3 Randomly pick a subset of training samples $D' \subset \mathbf{D}$
4 $loss = 0$
5 **for** *each training sample* $\{A_i, X_i\} \in D'$ **do**
6 $Z_i = \Phi_e(A_i, X_i)$
7 $C_i = \Phi_c(Z_i)$
8 $\Psi = \Theta_s(Z_i)$
9 $Z_{weighted} = zeros[N, f]$
10 **for** $j = 1$; $j \leq N$, j++ **do**
11 $Z_{weighted}[j, :] = \Psi_j \times Z_i[j, :]$
end
12 $Y = \Theta_c(Z_{weighted})$
13 $loss \mathrel{+}= log(Y[C_i])$
end
14 $loss = \frac{-loss}{m}$
15 Adjust weights in Θ based on the loss computed
end
16 **return** $\Theta = \{\Theta_s, \Theta_c\}$

Algorithm 1: The initial learning stage.

B. The Interpretation Stage

As shown in Algorithm 2, CFGExplainer needs the trained instances of the node scoring component Θ_s and the GNN based node embedding generation component Φ_e to generate interpretability results. The algorithm also takes the following parameters: the number of real nodes in the ACFG $N_{real} \leq N$ disregarding the padded nodes, the $step\_size$ ($step\_size \leq 100$ and $100\% step\_size == 0$) specifying the percentage of the graph to prune at each iteration, and the set of all nodes V associated with the ACFG to be interpreted. The output of this algorithm is $V_{ordered}$, which is a set of nodes ordered based on their importance on the classification task. Additionally, Algorithm 2 produces the adjacency matrices of subgraphs, each of which contains top $N_{real} * step\_size\%$ most important nodes.

First, the algorithm initializes $V_{ordered}$ and $subgraphs$ as empty sets (line 1). Then it assigns a set of all node indices of the graph to variable $all\_node\_indices$ (line 2), which stores the indices of nodes that have not yet been pruned. Next, the algorithm computes N_{step}, which stores the number of nodes to be pruned off from the graph at each step based on the user defined $step\_size$ (line 3). The algorithm then loops through $graph\_size$ with $step\_size$ starting from the original graph (i.e., $graph\_size = 100$) down to the smallest graph (i.e., $graph\_size = step\_size$) and prunes the graph in lines 4-18.

During each pruning iteration, the adjacency matrix of the current subgraph A is appended to the final result $subgraphs$ (line 5). Then the node embeddings are generated by calling $\Phi_e(A, X)$ (line 6) followed by obtaining the scores Ψ (line 7). In lines 8-18, the algorithm prunes off N_{step} least important nodes that have the lowest scores in Ψ. For each node to be removed within the loop at lines 8-18, the algorithm first identifies the node with the lowest score (i.e., min_score) and the corresponding node index (i.e., min_index) among nodes in $all\_node\_indices$ (lines 9-14). Afterwards, the set $all\_node\_indices$ is updated by removing min_index and appending the corresponding node to $V_{ordered}$ (lines 15-16). Next, the graph adjacency matrix is masked by zeroing out all the outgoing edges from the node associated with min_index (i.e., $A[min\_index, :] = [0, 0, ..., 0]$) (line 17). Afterwards, all incoming edges to the node with min_index (i.e., $A[:, min\_index]$) are zeroed out in a similar fashion (line 18). This process leads to an adjacency matrix where node v_{min_index} is removed. In line 19, the set $V_{ordered}$ is reversed such that the first node is the node that is most important for the classification task while the last node is the least important node. Finally, the adjacency matrices are reversed in order such that the first adjacency matrix corresponds to the smallest subgraph with top $step\_size\%$ most important nodes, whereas the last adjacency matrix corresponds to the original graph (line 20).

Procedure `Interpret`

1 $V_{ordered} = \emptyset, subgraphs = \emptyset$
2 $all\_node\_indices = \{1, 2, \ldots, N_{real}\}$
3 $N_{step} = \frac{step\_size}{100} \times N_{real}$
4 **for** $graph\_size = 100$; $graph\_size \geq step\_size$, $graph\_size \mathrel{-}= step\_size$ **do**
5 $subgraphs.append(A)$
6 $Z = \Phi_e(A, X)$
7 $\Psi = \Theta_s(Z)$
8 **for** $i = 1$; $i \leq N_{step}$, i++ **do**
9 $min\_index = 1$
10 $min\_score = +\infty$
11 **for** $j \in all\_node\_indices$ **do**
12 **if** $min\_score > \Psi_j$ **then**
13 $min\_score = \Psi_j$
14 $min\_index = j$
 end
15 $all\_node\_indices.remove(min\_index)$
16 $V_{ordered}.append(v_{min\_index})$
17 $A[min\_index, :] = [0, 0, ..., 0]$
18 $A[:, min\_index] = [0, 0, ..., 0]$
 end
 end
19 $V_{ordered}.reverse()$
20 $subgraphs.reverse()$
21 **return** $V_{ordered}$, $subgraphs$

Algorithm 2: The interpretation stage of CFGExplainer.

V. Evaluation

This section presents the experimental results of CFGExplainer. We first present the dataset and the model architecture, and then compare the performance of CFGExplainer against three graph-based interpretability models, namely GNNExplainer, SubgraphX and PGExplainer. All deep learning models presented in this section were trained and tuned on a 2.3-3.7 GHz Intel Xeon Gold 6140 machine with NVIDIA Tesla P100 12GB GPU. The source code of CFGExplainer and the top 10% and 20% of subgraphs generated from malware samples are available in Github[1].

A. Dataset and Model Architecture

Below, we provide details about the dataset and the model architecture of CFGExplainer and the GNN classifier used in our evaluations.

[1]https://github.com/dherath/CFGExplainer

YANCFG dataset: We consider a graph dataset in [11], which contains the CFG and the binary executable of 11 distinct malware families: Bagle, Bifrose, Hupigon, Ldpinch, Lmir, Rbot, Sdbot, Swizzor, Vundo, Zbot, Zlob, and one Benign class. Unlike a malware classification task, our objective is to interpret and understand the classification result, which requires us to analyze the binary executable of malware samples. This was the only dataset we were able to obtain that had both the CFGs and the malware executables. According to [11], the truth labels for the dataset were generated by a majority voting scheme, based on detection results of five antivirus scanners returned by VirusTotal online malware analysis service. The CFGs are converted into their attributed forms presented in section II. We note that CFGExplainer is able to handle any other format of node features as long as the inputs to CFGExplainer are the node embeddings generated by the GNN. We consider 1056 graphs equally distributed across all the families mentioned above. The dataset contains graphs with at most 7352 nodes and at most 13576 edges.

Model Architecture: Below, we provide the model architecture of the ACFG classifier and the deep learning models used in CFGExplainer. We classify the ACFGs using a Graph Neural Network (GNN) $\Phi = \{\Phi_e, \Phi_c\}$ constructed as follows. Φ_e is constructed by three inter-connected Graph Convolution Network (GCN) layers with different sizes (1024, 512, 128), where the size of the final node embedding is 128. Each of the

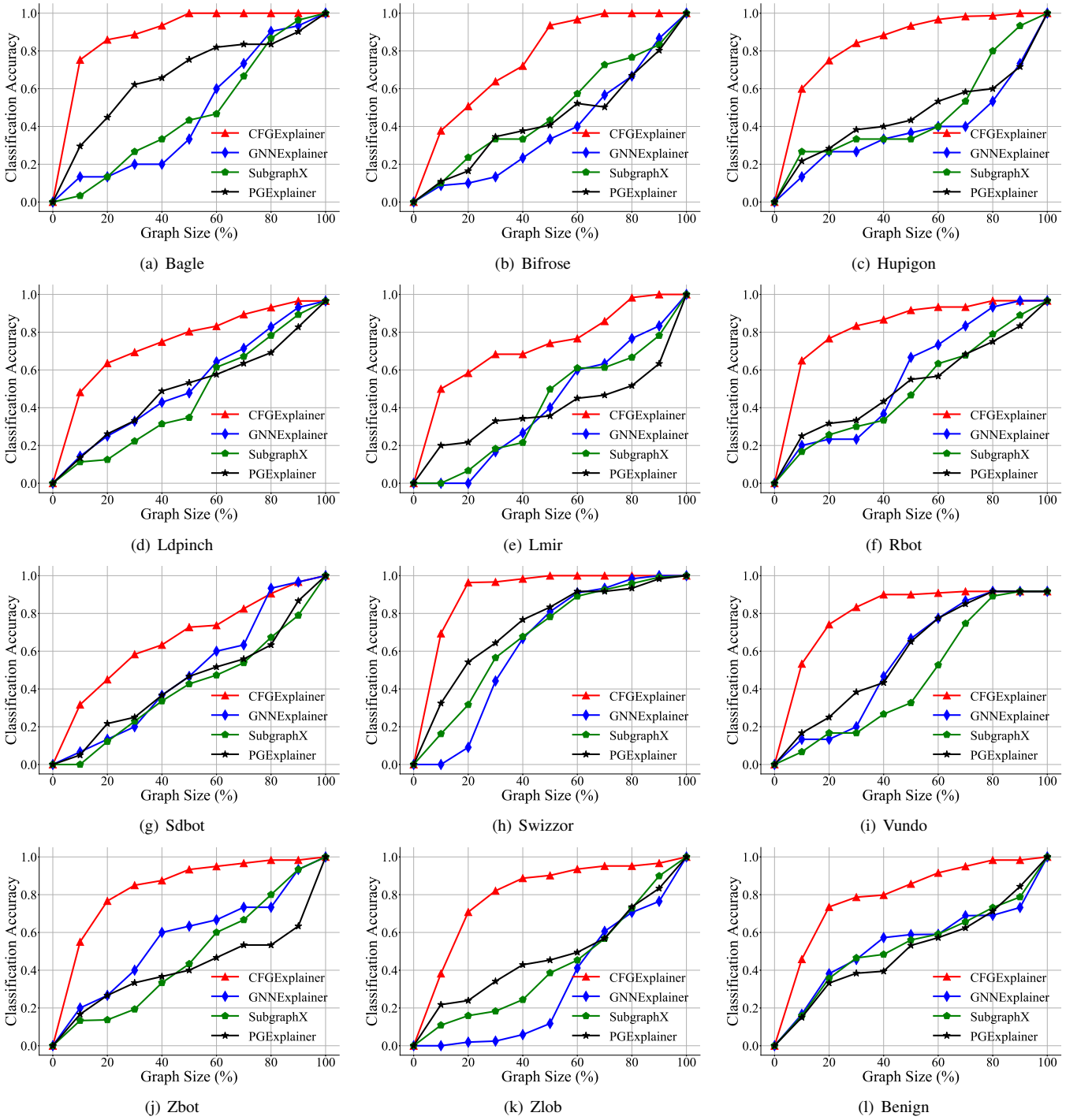

Fig. 2. The classification accuracy of subgraphs constructed.

GCN layers is activated by Relu [30] activation function. The classifier Φ_c is a densely connected linear layer that produces class labels across 12 ACFG families. All node embeddings generated by Φ_e are considered by Φ_c for classification. The intermediate representations in Φ_e include not only node features but also topological relationships among nodes. Having a large dimension for embeddings allows us to project ACFGs so that we can classify with high accuracy (i.e., 98% across all ACFG types). As the GCN-based classifier requires the input graph to contain a fixed number (i.e., 7352) of nodes, all ACFGs with fewer than 7352 nodes are padded up to that value using temporary nodes with zeroed out features with no edges.

In CFGExplainer, the node scoring component Θ_s is a feed-forwarding neural network constructed with three connected dense layers with sizes 64, 32, and 1, where the sigmoid

TABLE III
SUMMARY OF THE CLASSIFICATION RESULTS FOR TOP 10%, 20% SUBGRAPHS AND THE AUC RESULTS IN FIGURE 2.

ACFG Family	CFGExplainer			GNNExplainer			SubgraphX			PGExplainer		
	Classification Accuracy		AUC	Classification Accuracy		AUC	Classification Accuracy		AUC	Classification Accuracy		AUC
	10% Graph	20% Graph		10% Graph	20% Graph		10% Graph	20% Graph		10% Graph	20% Graph	
Bagle	0.7531	0.8594	0.8933	0.1333	0.1333	0.4670	0.0334	0.1333	0.4663	0.2958	0.4480	0.6669
Bifrose	0.3781	0.5073	0.7646	0.0876	0.1000	0.3887	0.1000	0.2358	0.4835	0.1083	0.1645	0.4401
Hupigon	0.6000	0.7500	0.8448	0.1333	0.2666	0.3933	0.2666	0.2666	0.4700	0.2166	0.2833	0.4650
Ldpinch	0.4821	0.6357	0.7469	0.1428	0.2500	0.5226	0.1128	0.1250	0.4567	0.1345	0.2613	0.4960
Lmir	0.5000	0.5833	0.7300	0.0000	0.0000	0.4166	0.0000	0.0667	0.4134	0.2000	0.2166	0.4013
Rbot	0.6500	0.7666	0.8316	0.2000	0.2333	0.5650	0.1673	0.2562	0.4997	0.2500	0.3166	0.5200
Sdbot	0.3170	0.4500	0.6645	0.0666	0.1333	0.4866	0.0000	0.1213	0.4087	0.0500	0.2166	0.4425
Swizzor	0.6939	0.9635	0.9107	0.0000	0.0909	0.6334	0.1621	0.3166	0.6769	0.3242	0.5424	0.7360
Vundo	0.5333	0.7416	0.8025	0.1333	0.1333	0.5532	0.6667	0.1666	0.4533	0.1666	0.2500	0.5799
Zbot	0.5500	0.7666	0.8358	0.2000	0.2666	0.5666	0.1333	0.1369	0.4730	0.1666	0.2666	0.4200
Zlob	0.3823	0.7078	0.8005	0.0000	0.0192	0.3208	0.1088	0.1592	0.4228	0.2176	0.2392	0.4812
Benign	0.4583	0.7348	0.7967	0.1667	0.3816	0.5367	0.1583	0.3567	0.5290	0.1500	0.3318	0.5043
Average	0.5239	0.7055	0.8018	0.1053	0.1673	0.4875	0.1591	0.1950	0.4794	0.1900	0.3391	0.5127

activation function is used by the final layer to obtain scores $\Psi \in [0,1]^{1\times N}$. The classifier of CFGExplainer is a feed-forwarding neural network constructed with three connected dense layers with sizes 64, 32, and 16, which is in turn connected to a final dense layer that produces classification probabilities with the softmax activation function. When the surrogate model in CFGExplainer learns to classify malware using weighted node embeddings, CFGExplainer also learns to assign higher scores to more important nodes.

B. Quantitative Evaluation

In this section, we evaluate the classification accuracy of equisized subgraphs produced by four GNN-based interpretability models – CFGExplainer, GNNExplainer [16], SubgraphX [18] and PGExplainer [17].

Our intuition is that a better interpretability solution should be able to identify a smaller subgraph that leads to similar classification accuracy as when the original graph is used. Similar to CFGExplainer, GNNExplainer, SubgraphX and PGExplainer learn to identify useful subgraphs by perturbing the original graphs. The perturbations for the models GNNExplainer and PGExplainer are carried out by first identifying a mask for edges and then combining it with the original graph. GNNExplainer directly learns this mask as an optimization task for each graph separately. PGExplainer uses a generative deep neural network for this task. SubgraphX searches through different permutations of the original graph using a Monte-Carlo tree search [25] method assisted by Shapley value calculations [26], [27]. The Shapley values are used as a score to identify important subgraphs. To make a fair comparison, CFGExplainer uses the same trained GNN classifier used by the other three models.

Figure 2 compares the classification accuracy of the subgraphs produced by three models for eleven malware families (Figure 2 (a) – (k)) and one benign class (Figure 2(l)). The subgraphs contain $10\% - 100\%$ nodes with a step size 10%. Table III summarizes the classification accuracy for subgraphs containing top 10% and 20% important nodes, and the Area Under the Curve (AUC) for Figures 2(a)-2(l). For the AUC computation, the graph size ($0\% - 100\%$) is normalized between $0-1$ so that AUC $\in [0,1]$. A larger AUC score entails that the model is able to identify smaller subgraphs that lead to high classification accuracy. We use this metric because it encompasses the classification accuracy for all the subgraphs into one numeric value.

When considering subgraphs containing top 20% important nodes, CFGExplainer achieves high classification accuracy (over 70%) for eight malware families: Bagle, Hupigon, Rbot, Swizzor, Vundo, Zbot, Zlob and Benign. In contrast, the average classification accuracy of subgraphs containing top 20% nodes produced by GNNExplainer, SubgraphX and PGExplainer for the same families are around 19%, 29% and 33%, respectively. For example, for the Bagle family, the subgraphs constructed using the top 20% nodes produced by CFGExplainer lead to 85% classification accuracy (shown in Figure 2(a)), which is 6.4 times higher than both GNNExplainer and SubgraphX, and also 1.9 times higher than PGExplainer. Although the subgraphs containing top 20% nodes produced by CFGExplainer for families Bifrose, Ldpinch, Lmir, Sdbot have lower classification accuracy (54% in average), it is still much higher than GNNExplainer (i.e., 12% in average), SubgraphX (i.e., 14% in average) and PGExplainer (i.e., 21% in average).

We see similar variation for subgraphs containing top 10% important nodes. As shown in Table III, on average the subgraphs containing top 10% important nodes produced by CFGExplainer has 52% classification accuracy, which is 4.9, 3.3 and 2.7 times higher than GNNExplainer, SubgraphX and PGExplainer, respectively. As expected, when the subgraph size increases, the classification accuracy increases for all three models. In addition, for all families, the subgraphs produced by CFGExplainer (except those containing 100% of nodes) lead to similar or higher classification accuracy than GNNExplainer, SubgraphX and PGExplainer. When considering the AUC variation, the models GNNExplainer, SubgraphX and PGExplainer have similar scores on average (0.4875, 0.4794

and 0.5127, respectively). The AUC value of CFGExplainer (0.8018) is 1.6 times higher than both GNNxplainer and SubgraphX, and also 1.5 times higher than PGExplainer, indicating that on average CFGExplainer can identify smaller subgraphs that make important contributions to classification with respect to the GNN classifier.

CFGExplainer is able to identify equisized subgraphs with high classification accuracy because of the following reasons. First, in the initial learning stage, CFGExplainer learns important patterns by considering a global view of all ACFGs. The node scoring component of CFGExplainer is trained using many graphs from different ACFG families so that CFGExplainer can generalize and learn similar patterns across the same ACFG family (e.g., common patterns in Bagle malware ACFGs) while at the same time learning discriminative patterns across different ACFG families. Secondly, CFGExplainer leverages the inherent importance of nodes in an ACFG for the explanation task. In contrast, neither GNNExplainer nor SubgraphX considers the global knowledge present across multiple ACFGs, since they employee a local search heuristic for providing explanations. While PGExplainer considers global knowledge when it trains a generative deep neural network, it gives more prominence in learning patterns from the edge distribution in graphs. In ACFGs, nodes are more important to an analyst since they represent code blocks. Additionally, PGExplainer is trained to generate new graphs that are similar to an ACFG classified by the GNN model. In contrast, CFGExplainer directly learns the importance of nodes in the graphs to be interpreted, which we believe is an easier learning task.

We followed the metrics (i.e., accuracy) used in GNNExplainer and PGExplainer to conduct experiments. SubgraphX uses *sparsity* and *fidelity* to evaluate the effectiveness of explainers. The fixed step size in our experiments corresponds to the fixed level of sparsity [31]. The accuracy results are comparable with $fidelity{-}^{acc}$ in [31], which shows the prediction change by keeping the important structure and removing the unimportant structure in the graphs. $fidelity{-}^{acc}$ computes the difference between the accuracy of subgraphs and the original graph. In the future, we plan to obtain more thorough results with the sparsity and fidelity metrics by searching combinations of subgraphs with different step sizes to decide the best step size for an explanation.

C. Complexity Analysis

Below, we provide a complexity analysis on the explainers presented in this section. Table IV gives the average time taken to produce a single explanation for an ACFG. The table shows that, on average, GNNExplainer and SubgraphX take the most time to produce a single explanation (i.e., around 42.8 minutes and 127.8 minutes respectively). This is because GNNExplainer and SubgraphX employ a local search heuristic to identify explanations. In contrast, CFGExplainer and PGExplainer generate explanations in around 3.9 and 6.4 minutes, respectively. CFGExplainer is around 11 times faster than GNNExplainer and 33 times faster than SubgraphX. This extra speed in explaining graphs comes at the cost of an offline training procedure in both CFGExplainer and PGExplainer. While both CFGExplainer and PGExplainer require an offline training procedure, PGExplainer requires an input constructed from edge embeddings, as opposed to node embeddings that are used by CFGExplainer. Therefore, the maximum size of the input to PGExplainer could be $[N^2, 2f]$ where N is the number of nodes in the graph and f the embedding size ($N = 7352$ and $f = 128$ in our work), whereas the input to CFGExplainer is always of the size $[N, f]$.

TABLE IV
EXPLANATION TIME

Explainer	Offline Training Time	Average Time for Single Explanation
CFGExplainer	2 hours 11 minutes	3.9 $\pm$ 0.5 minutes
GNNExplainer	-	42.8 $\pm$ 3.1 minutes
SubgraphX	-	127.8 $\pm$ 10.2 minutes
PGExplainer	2 hours 46 minutes	6.4 $\pm$ 0.3 minutes

D. Qualitative Evaluation

This section provides a qualitative evaluation, where we analyze the top 20% subgraphs produced by CFGExplainer for interesting behavioural patterns in the malware.

For each malware family, we chose $11-15$ samples for deep static analysis. We perform two types of malware code analysis based on the top subgraphs returned by CFGExplainer:

- **Micro-level analysis**: We try to understand the unique malware patterns (e.g., obfuscation tricks) used by each malware sample from the code blocks included in its top subgraph.
- **Macro-level analysis**: We try to hypothesize the behavior of each malware sample based on the code blocks included within its top subgraph, particularly those Windows API calls used.

Micro-level analysis: Sample micro-level analysis results are provided in Table V. We have identified the following malware patterns.

Code manipulation: The general-purpose registers store the result of different operations and are one of the key factors to recognize unique behaviors. For example, the register EAX holds the return value for function calls. If the EAX register is used by the instruction immediately following the function call, then it could mean that a malware tries to manipulate the return value of a function [32]. For example, we have identified this type of code manipulation in the Bifrose malware sample. The instruction, [‘call’, ‘ds:Sleep’] followed by [‘mov’, ‘eax, [ebp+var_EC.hProcess]’] is a function call immediately followed by the instruction that modifies the return value of the function call.

XOR obfuscation: XOR operation is often used in the assembly code to set a register’s value zero by computing the XOR of the same registers. Malware authors also use the XOR instruction to obfuscate malicious activities in files or programs. For example, by computing the XOR of a 4-byte key with every byte of the data, malware authors can

TABLE V
SAMPLE ANALYSIS RESULTS OF TOP 20% BLOCK OF NODES IDENTIFIED BY CFGEXPLAINER.

No.	Malware Family	Type of unique patterns	Examples
1	Bagel	Code Manipulation	call sub_414120; pop eax; add esi,eax;
		Semantic-NOP obfuscation	nop; nop; nop; nop; nop; nop;
2	Bifrose	Code Manipulation	Call ds:Sleep; mov eax, [ebp+var_EC.hProcess];
		XOR obfuscation	xor [ecx],al; xchg al,ah; xchg ah,al; xor eax,ecx;
3	Hupigon	XOR obfuscation	xor al,55h;
5	Ldpinch	Code Manipulation	call sub_4010A6; pop eax;
		Windows API calls and DLL's	push offsetsub_40467A;lpStartAddress; call CreateThread; call ReadFile;
5	Lmir	Code manipulation	call GetModuleFileNameA; mov eax,ebx;
		XOR obfuscation	xor bl,ds:byte_40B28C[eax];
6	Rbot	Code manipulation	call sub_619E4; mov eax,[ebp+var_18];
7	Sdbot	Code manipulation	call QueryPerformanceCounter; mov eax,[ebp+var_9C];
8	Swizzor	Code manipulation	call _SEH_prolog; mov eax,dword_4347E8;
		XOR obfuscation	xor eax,0FFFFFFFFh;
9	Vundo	XOR obfuscation	xor edi,68A25749h;
		Semantic-NOP obfuscation	xchg esp,esp; push esi; mov esi,esi; push edi; mov eax,eax;
10	Zbot	Code manipulation	call j_SleepEx; movzx eax,wordptr[ecx];
		XOR obfuscation	xor edx,87BDC1D7h;
11	Zlob	Code manipulation	call ds:wsprintfA; mov eax,[ebp+hModule];

obfuscate the malicious data [33]. Presence of XOR operation of two different registers or XOR operation of a register and a constant (e.g. 'xor' '[ecx], 87BDC1D7h'), may represent an encryption or decryption of malware code to hide the malware data/code. We found this pattern in Bifrose and many other malware samples.

Semantic-NOP obfuscation: NOP is a one-byte instruction used in the assembly program that does nothing. NOP instructions may appear in benign or malicious applications. Malicious software sometimes use NOP instructions or other one-byte instructions that are semantically equivalent to NOP (e.g., mov edx, edx) to delay the execution, obfuscate the code (called Semantic-NOP obfuscation), or perform buffer overflow attacks. While analyzing malware samples Bagel and Vundo, we observed that many code blocks contain NOP instructions that do not modify any memory contents. One-byte instructions such "mov edx, edx", "mov esi, esi", or "xchg dl,dl" are also used as aliases for NOP to waste memory cycles and space [34]. Additionally, many instructions in Bagel and Vundo are only looping themselves using unconditional jumps.

Macro-level analysis: We can also analyze the malware behavior by analyzing the Windows API calls and Dynamic Link Libraries (DLLs) made in the important nodes reported by CFGExplainer. Windows API is the application program interface through which malware communicates with the system. For instance, malware can use Windows APIs to perform file operations, Windows registry accesses, or network communications. Using DLLs, malware can use other system libraries [32].

As an example, we observed the following function calls in Ldpinch malware family. Ldpinch tries to steal user credentials and then pass them to attackers through network. We have observed this behavior when analyzing Windows APIs called by Ldpinch in the top 20% nodes. Ldpinch uses a backdoor to gain access to user computers. It creates threads using multiple approaches to perform malicious activities on user computers. One approach used by Ldpinch is to load a new malicious library into a process by passing the library name to the *CreateThread* function call and specifying the start address for the library. In another approach, Ldpinch first uses the *CreateProcess* function call to create a new process. This newly created process then creates two threads using the *CreateThread* function call. One thread reads the data using the *ReadFile* function call from one end of a pipe and sends out the data through network using the *send* function call. Another thread is used to receive the data using the *recv* function call and then writes the data to another end of the pipe using the *WriteFile* function call. The pipe is created using the *CreatePipe* system call to perform read and write operations [32]. This behavior shows that Ldpinch tries to get control of the data coming to the socket.

VI. RELATED WORK

This section presents the related work for deep learning techniques used for malware classification and interpretability techniques.

A. Deep Learning for Malware Classification

There have been numerous attempts at identifying and classifying malware using Machine Learning (ML) techniques (e.g., [2], [3], [35], [36]). With the success of Deep Learning (DL) in other domains such as computer vision and natural language processing, there has been an increase in the usage of DL methods for malware classification [4], [5], [6], [37]. The works in [38], [39], [40] show that DL methods outperform classical ML techniques such as decision trees and random forests on the same training datasets.

With the success in computer vision, Convolutional Neural Network (CNN) architectures have been successfully used for malware classification [7], [8]. In these work, malware byte sequences are transformed into binary or colored images

for processing (e.g., [41], [4]). Other researchers have used sequence based deep learning models such as LSTM, GRU, and attention mechanisms for malware classification (e.g., [9], [10]). These models generally view program execution as a sequence of system calls or API calls. Attributed by the recent success in using Graph Neural Networks (GNNs) in other graph-based domains [42], [43], [44], recent research efforts have been targeted at using GNNs to classify malware samples based on their control flow graphs [11], [12], [13], [14]. These models have shown great promise in malware classification due to their capability to handle block level information as well as the topological relationships across nodes (i.e., blocks) in the graphs. MAGIC [11] is one such model that is built atop of DGCNN [45], where the DGCNN component is combined with pooling layers to handle CFGs of variable size. SDGNet [12] is a Graph Convolutional Network (GCN) similar to MAGIC, except that it uses spectral based laplacian convolution. GNNs have also been used to detect program similarity and characterize code based on CFGs. sciGCN [46] combines graph convolution with capsule networks to identify similarity scores between CFGs of two programs. funcGNN [47] builds atop of GraphSage [24] and uses attention mechanisms to identify program similarity. None of them explain classification results.

B. Interpretability Techniques

LEMNA [19] is one of the first explanation models that aim to give reasons for malware classifications carried out by DNNs, mainly Recurrent Neural Networks (RNN) and Multi-Layer Perceptron (MLP) models. However, LEMNA does not provide explanations for GNN based models that process malware CFGs. Explanation techniques used in image and text domains [48], [49], [50], [51] cannot be directly applied for graphs [31] as they do not consider topological and structural relationships in graphs. This has led to an increasing research effort in developing explanation methods specifically targeting graph data [31], [52], [53], [54], [18].

Gradient based methods such as Guided BP [55] and Grad CAM [56] provide instance-level explanations using gradient scores as a measure of importance with respect to predictions. However, these models may suffer from the saturation problem [57] due to masking attempts where the gradients hardly reflect change in value for GNN outputs that change minimally for some given inputs. Perturbation based methods such as GNNExplainer [16] and PGExplainer [17] learn to identify subgraphs by perturbing the original graphs by means of identifying a mask for edges or features and then combining it with the original graph. A successful mask would enable the GNN model to predict similar results as when the original graph was used. However, GNNExplainer [16] operates on a local perturbation heuristic, where masks are learnt from the beginning for each individual graph.

SubgraphX [18] is another local interpretability model that generates important subgraphs with respect to GNN classifications. Unlike GNNExplainer, SubgraphX utilizes Monte Carlo Tree Search (MCTS) [25] for this task. It populates a search tree where nodes represent the subgraphs and edges show parent-child relationships among them. A child subgraph is obtained by pruning nodes from a parent subgraph. SubgraphX utilizes shapley values [26], [27] as a score to identify important graph structure that contribute to classification. Similar to GNNExplainer, SubgraphX needs to locally search for explanations for each graph individually. Therefore, it also do not leverage any global information that may be present across similar graph types. This is in contrast to our solution CFGExplainer and PGExplainer [17] which leverage global information to provide instance-level explanations.

PGExplainer leverages a generative deep neural network that learns the edge distributions of the graphs in an offline manner so that it can generate new graphs for a given family. Once trained, this model is used to generate the masks to prune out unimportant edges. CFGExplainer, in contrast, not only prunes out unimportant edges, but also generates a score for each node indicating whether the node is important for the classification task. This enables malware analysts to identify blocks of code containing malicious activities more quickly. GraphLime [58], is a GNN explainer that is based on a local interpretability model called LIME [59]. However, GraphLime is designed for explaining node classification, instead of graph classification and hence cannot be used for malware ACFG classification. To the best of our knowledge, CFGExplainer is the first solution designed to provide interpretations for malware classification based on ACFGs.

VII. Conclusion

In this paper, we propose CFGExplainer, a deep learning based model for interpreting malware classification results. CFGExplainer identifies subgraphs of malware CFGs that contribute most towards malware classification and provides insight into the importance of the nodes within these subgraphs. Our experimental results show that CFGExplainer is able to identify top equisized subgraphs with higher classification accuracy than three state-of-the-art graph explanation solutions, namely GNNExplainer, SubgraphX and PGExplainer. In the future, we plan to extend our experiments with more datasets (e.g., MSKCFG [35]) and evaluate the effectiveness of CFGExplainer using additional metrics such as sparsity and fidelity in [18].

Acknowledgement: This work is supported in part by the National Science Foundation under grant OAC-1738929. We also thank the anonymous reviewers for their constructive comments.

References

[1] AV-TEST. (2021) Malware statistics & trends report by av-test. https://www.av-test.org/en/statistics/malware, Accessed: 2021-06-04.

[2] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede, "Performance analysis of machine learning and pattern recognition algorithms for malware classification," in *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*. IEEE, 2016, pp. 338–342.

[3] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 2014, 2014.

[4] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

[5] B. Cakir and E. Dogdu, "Malware classification using deep learning methods," in *Proceedings of the ACMSE 2018 Conference*, 2018, pp. 1–5.

[6] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 301–308.

[7] M. Kalash, M. Rochan, N. Mohammed, N. D. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*. IEEE, 2018, pp. 1–5.

[8] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, "Empowering convolutional networks for malware classification and analysis," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 3838–3845.

[9] B. Athiwaratkun and J. W. Stokes, "Malware classification with lstm and gru language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 2482–2486.

[10] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[11] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 52–63.

[12] Z. Zhang, Y. Li, H. Dong, H. Gao, Y. Jin, and W. Wang, "Spectral-based directed graph network for malware detection," *IEEE Transactions on Network Science and Engineering*, 2020.

[13] A. Abusnaina, M. Abuhamad, H. Alasmary, A. Anwar, R. Jang, S. Salem, D. Nyang, and D. Mohaisen, "Dl-fhmc: Deep learning-based fine-grained hierarchical learning approach for robust malware classification," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[14] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen, "Soteria: Detecting adversarial examples in control flow graph-based malware classifiers," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 888–898.

[15] J. Busch, A. Kocheturov, V. Tresp, and T. Seidl, "Nf-gnn: Network flow graph neural networks for malware detection and classification," in *33rd International Conference on Scientific and Statistical Database Management*, 2021, pp. 121–132.

[16] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, p. 9240, 2019.

[17] D. Luo, W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang, "Parameterized explainer for graph neural network," *Advances in neural information processing systems*, vol. 33, pp. 19 620–19 631, 2020.

[18] H. Yuan, H. Yu, J. Wang, K. Li, and S. Ji, "On explainability of graph neural networks via subgraph explorations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 241–12 252.

[19] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 364–379.

[20] IDA Pro. https://hex-rays.com/ida-pro/, Accessed: 2021-06-04.

[21] Ghidra. https://ghidra-sre.org/, Accessed: 2021-06-04.

[22] B. Bai, J. Liang, G. Zhang, H. Li, K. Bai, and F. Wang, "Why attentions may not be interpretable?" in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 25–34.

[23] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[24] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[25] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai." *AIIDE*, vol. 8, pp. 216–217, 2008.

[26] E. Kalai and D. Samet, "On weighted shapley values," *International journal of game theory*, vol. 16, no. 3, pp. 205–222, 1987.

[27] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st international conference on neural information processing systems*, 2017, pp. 4768–4777.

[28] I. V. Serban, R. Lowe, L. Charlin, and J. Pineau, "Generative deep neural networks for dialogue: A short review," *arXiv preprint arXiv:1611.06216*, 2016.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[30] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[31] H. Yuan, H. Yu, S. Gui, and S. Ji, "Explainability in graph neural networks: A taxonomic survey," *arXiv preprint arXiv:2012.15445*, 2020.

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

[33] M. K. A, *Learning malware analysis : explore the concepts, tools, and techniques to analyze and investigate Windows malware*, 1st ed. Birmingham ;: Packt, 2018.

[34] C. Jämthagen, P. Lantz, and M. Hell, "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries," in *2013 Workshop on Anti-malware Testing Research*, 2013, pp. 1–9.

[35] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *arXiv preprint arXiv:1802.10135*, 2018.

[36] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[37] Q. Le, O. Boydell, B. Mac Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digital Investigation*, vol. 26, pp. S118–S126, 2018.

[38] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *computers & security*, vol. 77, pp. 578–594, 2018.

[39] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data," in *2017 IEEE symposium on computers and communications (ISCC)*. IEEE, 2017, pp. 438–443.

[40] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 3422–3426.

[41] T. Hsien-De Huang and H.-Y. Kao, "R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 2633–2642.

[42] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2018.

[43] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, 2020.

[44] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[45] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[46] P. Haridas, G. Chennupati, N. Santhi, P. Romero, and S. Eidenbenz, "Code characterization with graph convolutions and capsule networks," *IEEE Access*, vol. 8, pp. 136 307–136 315, 2020.

[47] A. Nair, A. Roy, and K. Meinke, "funcgnn: A graph neural network approach to program similarity," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[48] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," in *In Workshop at International Conference on Learning Representations*. Citeseer, 2014.

[49] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[50] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The building blocks of interpretability," *Distill*, vol. 3, no. 3, p. e10, 2018.
[51] H. Yuan, Y. Chen, X. Hu, and S. Ji, "Interpreting deep models for text analysis via optimization and regularization methods," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 5717–5724.
[52] M. S. Schlichtkrull, N. De Cao, and I. Titov, "Interpreting graph neural networks for nlp with differentiable edge masking," in *International Conference on Learning Representations*, 2020.
[53] Y. Zhang, D. Defazio, and A. Ramesh, "Relex: A model-agnostic relational model explainer," in *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, 2021, pp. 1042–1049.
[54] H. Yuan, J. Tang, X. Hu, and S. Ji, "Xgnn: Towards model-level explanations of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 430–438.
[55] F. Baldassarre and H. Azizpour, "Explainability techniques for graph convolutional networks," in *International Conference on Machine Learning (ICML) Workshops, 2019 Workshop on Learning and Reasoning with Graph-Structured Representations*, 2019.
[56] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann, "Explainability methods for graph convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 772–10 781.
[57] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3145–3153.
[58] Q. Huang, M. Yamada, Y. Tian, D. Singh, D. Yin, and Y. Chang, "Graphlime: Local interpretable model explanations for graph neural networks," *arXiv preprint arXiv:2001.06216*, 2020.
[59] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.