

On the Consistency of GNN Explanations for Malware Detection

Hossein Shokouhinejad, Griffin Higgins, Roozbeh Razavi-Far, Hesamodin Mohammadian, Ali A. Ghorbani

Canadian Institute for Cybersecurity (CIC), University of New Brunswick, Fredericton, NB, Canada

Email: {hossein.shokouhinejad, griffin.higgins, roozbeh.razavi-far, h.mohammadian, ghorbani}@unb.ca

Abstract—Control Flow Graphs (CFGs) are critical for analyzing program execution and characterizing malware behavior. With the growing adoption of Graph Neural Networks (GNNs), CFG-based representations have proven highly effective for malware detection. This study proposes a novel framework that dynamically constructs CFGs and embeds node features using a hybrid approach combining rule-based encoding and autoencoder-based embedding. A GNN-based classifier is then constructed to detect malicious behavior from the resulting graph representations. To improve model interpretability, we apply state-of-the-art explainability techniques, including GNNExplainer, PGExplainer, and CaptumExplainer, the latter is utilized three attribution methods: Integrated Gradients, Guided Backpropagation, and Saliency. In addition, we introduce a novel aggregation method, called RankFusion, that integrates the outputs of the top-performing explainers to enhance the explanation quality. We also evaluate explanations using two subgraph extraction strategies, including the proposed Greedy Edge-wise Composition (GEC) method for improved structural coherence. A comprehensive evaluation using accuracy, fidelity, and consistency metrics demonstrates the effectiveness of the proposed framework in terms of accurate identification of malware samples and generating reliable and interpretable explanations.

Index Terms—Graph Neural Network, Explainability, Machine Learning, Malware Detection, Dynamic Analysis, Control Flow Graph.

I. INTRODUCTION

In the past few years, malware attacks have escalated dramatically, highlighting the limitations of conventional malware detection methods, such as signature-based techniques. While these methods are quick and widely utilized, they fall short in identifying zero-day and sophisticated malware threats [1]. Consequently, there has been a significant shift towards integrating machine learning (ML) strategies, noted for their enhanced detection capabilities, adaptability to emerging threats, and reduced false positives. Numerous studies have thus shifted focus towards leveraging ML [2]–[5] for malware detection and classification.

As malware becomes more complex and developers employ advanced tactics to evade detection, the need for robust representation of malware samples has become evident. Recent research highlights the effectiveness of graph-based models in depicting malware behavior, which facilitates improved detection performance. These models provide a detailed view of a program’s execution, helping analysts understand the program’s logic, pinpoint vulnerabilities, and expose malicious activities, including hidden or scrambled code. Various graph

structures, including Control Flow Graph (CFG) [6]–[9], Function Call Graph (FCG) [10]–[12], and Application Programming Interface (API) Call Graph (ACG) [13]–[15], have been increasingly utilized to feed data into ML models designed for malware detection. Although initial efforts involved using these graph structures with CNN or RNN architectures, the advent of Graph Neural Networks (GNNs) with variants like Graph Convolutional Networks (GCNs), Graph Isomorphism Networks (GINs), GraphSage, and Graph Attention Networks (GATs) have demonstrated superior outcomes when integrated with CFGs and FCGs.

The complexity of graph-based models often results in a lack of clarity in their decision-making processes, which is particularly crucial in the field of malware detection. To address this, various explainability methods for GNNs have been developed to identify the key subgraphs that influence the model’s decisions [16]. Explainability techniques provide insights into how GNN models reach their decisions by highlighting the most important nodes, edges, and subgraphs that contribute to the prediction. These methods improve the interpretability of GNN models, making them more transparent and trustworthy, which is especially valuable in security-related applications such as malware detection.

A critical factor influencing the performance of explainers is the method used to generate the important subgraph based on edge importance weights. Most existing research employs a conventional approach that selects the top-weighted edges to construct the important subgraph. However, the quality of the generated subgraph directly affects the reliability of the explanation and the model’s overall interpretability. Therefore, evaluating the effectiveness and correctness of explainers is essential to ensure consistent and meaningful insights.

To assess the quality of explainers, several quantitative metrics are introduced to complement qualitative analyses. Among these, fidelity is a widely recognized metric that measures how faithfully the explainer represents the model’s behavior. Specifically, fidelity assesses whether the model’s performance on the original graph is consistent with its performance on the identified important subgraph while showing a significant deviation when evaluated on the unimportant subgraph. Another useful metric is the explainer accuracy, which measures how well the important subgraph retains the crucial information needed for the model’s decision. Ideally, the explainer accuracy should align closely with the target GNN test accuracy, confirming that the identified subgraph

captures the essential decision-making components of the model. However, evaluating the explainer with these metrics without considering the robustness of these metrics can lead to misleading conclusions. For example, high fidelity might reflect overfitting to the original graph rather than true model understanding. Similarly, explainer accuracy may be influenced by noise in the data or the structure of the original graph, potentially compromising the validity of the evaluation. Therefore, ensuring the consistency and stability of these metrics under different graph configurations is crucial for a reliable assessment of the explainer’s performance.

In this paper, we propose a graph-based malware detection framework that leverages dynamically generated CFGs and hybrid node embeddings derived from assembly instructions. A GNN model is trained to classify the resulting graphs as benign or malicious. To enhance interpretability, we incorporate multiple explanation techniques and introduce a novel aggregation-based approach to improve explanation quality. We further propose an effective subgraph extraction strategy and assess the framework’s robustness using fidelity and consistency-based evaluation metrics.

The key contributions of this article are as follows:

- A dynamic graph-based malware detection framework utilizing CFGs and a hybrid rule-based and autoencoder-based node embedding strategy.
- A novel explanation aggregation method, RankFusion explainer, that enhances the informativeness and reliability of subgraph-based interpretations.
- A new subgraph extraction technique, GEC, designed to generate well-connected, high-importance subgraphs.
- A systematic evaluation using accuracy, fidelity, and a consistency metric that measures the sensitivity of explanations to input perturbations.

The structure of the paper is organized as follows: Section II provides a review of the relevant background. Section III details the proposed framework, while Section IV focuses on the explainability metrics in more detail. Section V, presents the experimental results and analysis. Finally, Section VI concludes the paper and outlines potential directions for future research.

II. BACKGROUND

The growing complexity and sophistication of malware have driven the need for more advanced detection techniques capable of thoroughly analyzing and identifying malicious programs. In recent years, GNNs have emerged as a powerful tool for malware detection due to their ability to model complex relationships and dependencies within the structural data of malicious code [16]. Unlike traditional machine learning models that rely on flat feature representations, GNNs leverage the inherent graph structure of malware, such as CFGs, to capture both local and global patterns. Recent advances in GNN-based malware detection have focused on improving scalability, interpretability, and adaptability to evolving threat landscapes. Moreover, explainability frameworks integrated with GNNs are becoming increasingly important, allowing

security analysts to understand the rationale behind classification decisions, identify critical graph components, and uncover hidden attack patterns. In this section, we provide an overview of the state-of-the-art techniques in GNN-based malware detection and discuss their approaches.

Recent advancements in GNN-based malware detection have introduced innovative techniques to improve detection accuracy, scalability, and robustness. One notable work is the Spectral-based Directed Graph Network (SDGNet), which addresses the challenge of detecting malware using directed graphs. Traditional spectral-based GNNs struggle with directed graphs due to the asymmetry of adjacency matrices. SDGNet overcomes this by employing three weighted graph matrix normalization methods (normal, aggregation-based, and propagation-based) to transform directed graphs into symmetrical matrices. It then applies a Multi-aspect Directed GCN (MDGCN) to learn comprehensive graph representations from these matrices [17].

A dynamic malware analysis approach is presented in the DMalNet framework [15], which constructs an API call graph from API call sequences and applies a hybrid feature encoder to extract semantic features from both API names and arguments using techniques like Word2Vec, feature hashing, and similarity encoding. A GNN combining a modified GIN and GAT is then used to learn both content and structural features from the graph, capturing complex relationships between API calls for effective malware detection and classification.

An alternative static detection model for malicious JavaScript is introduced in JStrong [18]. It generates an abstract syntax tree from JavaScript source code and integrates data flow and control flow information into a program dependency graph. A GNN is then employed to analyze the graph and classify malicious code based on its structural patterns.

A robust ensemble model for malware detection is described in REMSF [19], which leverages semantic feature fusion. The model extracts static and semantic features from PE files, including byte histograms, entropy, and string information. It constructs a heterogeneous graph to model the relationships between PE files, imported DLLs, and APIs. By combining different classifiers through ensemble learning, REMSF improves detection accuracy and captures complex semantic relationships more effectively.

Another noteworthy approach is MalwareExpert [20], an expert system that detects malicious binaries using a GNN-based model. It identifies essential functions in the analyzed sample and highlights the most critical subgraphs involved in malicious behavior, providing an explainable output to improve transparency and understanding of the detection process.

A few-shot malware classification model using a graph transformer with a triplet-loss function is introduced in [21]. This method extracts CFGs from assembly-level code and applies a path-sampling algorithm to capture functional patterns. The graph transformer, equipped with an attention mechanism, selectively embeds attack pathways from the CFGs. The triplet-loss function enables the model to learn a disentangled feature space, improving classification even with

limited samples.

Documentation-augmented malware detection is explored in DawnGNN [22]. It constructs API graphs from Windows API call sequences and enhances them with semantic information extracted from official Windows API documentation using a pre-trained BERT model. The enhanced API graphs are then processed using GAT, which learns contextual information and improves malware detection by capturing both structural and semantic relationships among API calls.

Temporal and structural feature learning for malware detection is introduced in TS-Mal [23]. This model extracts fine-grained temporal patterns from API call sequences using a TextRCNN-based temporal vector learning method. It then models the relationships between API categories using a heterogeneous graph and generates dense structural representations through GAT. By combining both temporal and structural features, TS-Mal enhances the model's ability to detect complex malware patterns.

MalGNE [7] presents a novel malware detection framework based on CFG node embedding in a low-dimensional space. It addresses limitations in node feature extraction by applying a unique instruction encoding rule to handle out-of-vocabulary (OOV) issues and reduce redundancy. The model processes node vectors through an aggregation layer and a sequence layer to extract execution sequence and aggregation features. These vectors are mapped into a low-dimensional continuous space, improving both detection accuracy and efficiency through GNN-based learning.

Recent advancements in malware detection have not only focused on improving accuracy but also on enhancing model interpretability using GNN explainers. GNN explainers aim to provide insights into how a model reaches its decisions by identifying the most influential graph components. A widely adopted method in GNN explainability is GNNExplainer [24], which provides model-agnostic explanations for predictions made by any GNN model on tasks such as node classification, link prediction, and graph classification. GNNExplainer identifies a compact subgraph structure and a small subset of node features that are most influential for a GNN's prediction. It formulates the explanation process as an optimization problem that maximizes the mutual information between a GNN's prediction and the distribution of possible subgraph structures. By learning both a structural and feature mask, GNNExplainer generates clear and consistent explanations, enhancing the interpretability of complex GNN models.

Building on this foundation, SubgraphX [25] introduces a GNN explanation method that explains GNN predictions through subgraph exploration. Unlike methods that focus on individual nodes or edges, SubgraphX identifies important subgraphs using a Monte Carlo tree search (MCTS) algorithm. It evaluates the importance of subgraphs by computing Shapley values, which measure the marginal contribution of each subgraph to the model's prediction. This approach allows SubgraphX to highlight key subgraph structures responsible for the classification decision, providing more intuitive and interpretable explanations.

In the context of malware classification, CFGExplainer [26] is specifically designed to interpret GNN-based malware classification results from CFGs. CFGExplainer identifies the most influential subgraphs contributing to classification and provides insight into the importance of individual nodes (basic blocks) within these subgraphs. The framework employs a two-stage process: an initial learning stage where a deep learning model assigns importance scores to node embeddings, and an interpretation stage where these scores are used to prune the graph and identify critical subgraphs. CFGExplainer enhances interpretability by revealing both the structural and functional aspects of malware behavior.

To improve scalability and generalization, PGExplainer [27] introduces a parameterized explanation method for GNNs that provides consistent and generalized explanations for multiple instances. Unlike GNNExplainer, which generates explanations independently for each instance, PGExplainer employs a deep neural network to parameterize the generation of explanations, enabling it to generalize across different instances. PGExplainer models the explanatory subgraph as an edge distribution and generates explanations by optimizing the mutual information between the subgraph structure and the GNN's prediction. This approach improves both scalability and efficiency, making it suitable for inductive settings where new instances can be explained without retraining the explainer.

III. PROPOSED METHOD

Our proposed framework for malware detection begins with dynamically generating a graph-based representation of the input sample through dynamic analysis. Specifically, a CFG is constructed, where each node represents a basic block of assembly instructions and edges represent control flow relationships between them. To enhance the accuracy and robustness of the detection model, a hybrid node feature embedding technique is used, which combines rule-based encoding and autoencoder-based embedding to convert the assembly instructions in each node into numerical vectors. A GNN-based model is then employed to classify the malware sample based on the graph structure and node features. To improve the interpretability of the model, the framework integrates GNNExplainer, PGExplainer, and CaptumExplainer to identify key subgraphs. Additionally, an aggregation scheme combines the edge rankings from the two top explainers to enhance the consistency of the extracted subgraphs. A new subgraph extraction method, GEC, is also introduced to generate more connected and meaningful subgraphs by prioritizing edges with the highest importance weights. The overall structure of the proposed framework is illustrated in Figure 1.

A. Node Feature Embedding

The dynamically generated CFGs contain several types of features in each node, but not all are suitable or useful for the decision-making module. The assembly instructions of each node in the CFG are selected as the key features, and a two-step embedding technique is used to map them into

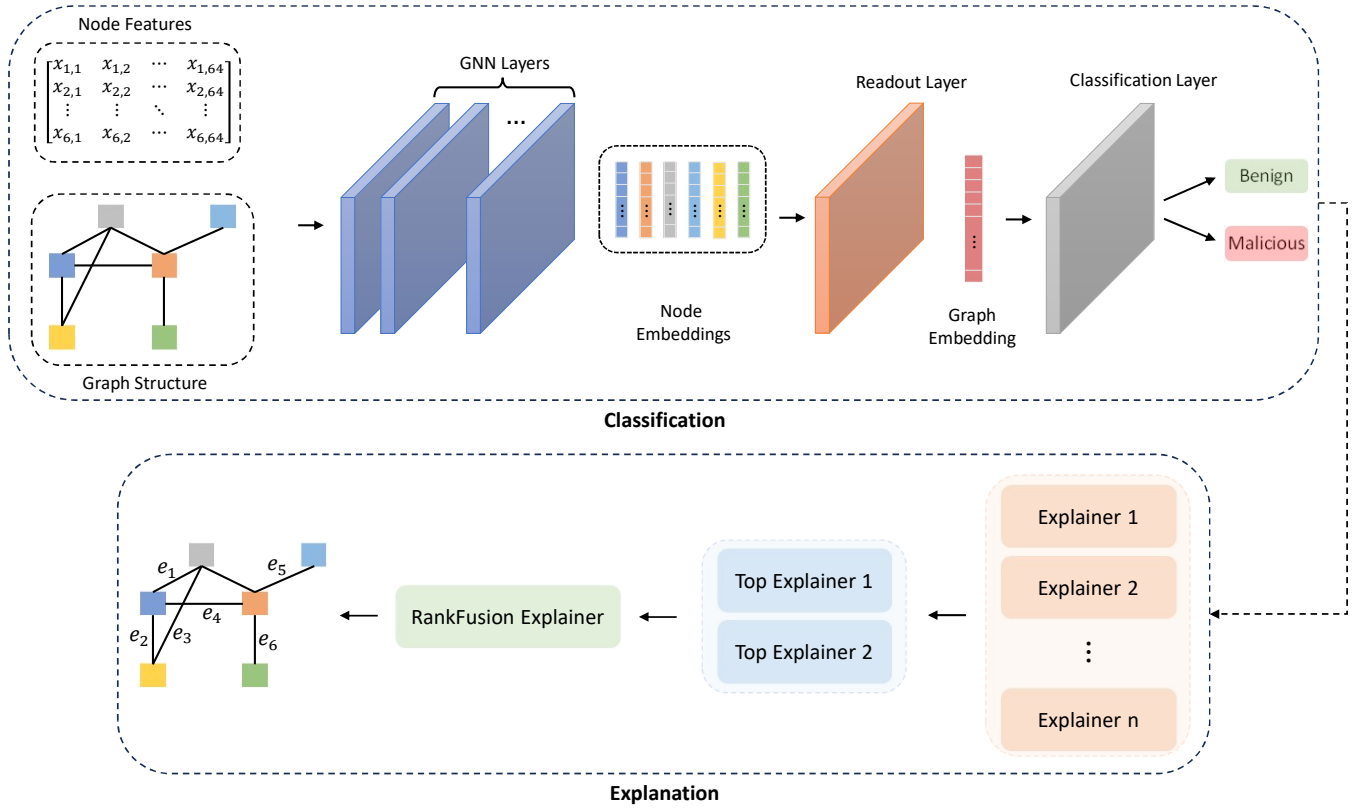


Fig. 1: Proposed framework for interpretable malware detection.

a 64-dimensional real-valued vector. This process includes a rule-based instruction encoding strategy followed by machine learning-based dimensionality reduction using an autoencoder, as shown in Figure 2.

1) *Rule-Based Instruction Encoding*: To encode the assembly instructions of CFG nodes, we adapted the encoding process outlined in [7] with slight modifications to better suit the structure of dynamically generated CFGs. Each x86-64 assembly instruction consists of up to seven components: option, prefix, opcode, ModRM, SIB, displacement, and immediate. The encoding process for each component is defined as follows:

- **Prefix**: The prefix includes four fields: the extra segment (ES) register, operand-size override, address-size override, and lock prefix. The ES segment register has seven possible values, while the other three fields are binary (0 or 1). Thus, the prefix is encoded as a nine-dimensional one-hot vector.
- **Opcode**: The opcode defines the core operation performed by the instruction, with 256 possible values. It is encoded as a 256-dimensional one-hot vector.
- **ModRM**: ModRM is a one-byte field divided into three segments: two bits for the mode field, three bits for the register field, and three bits for the memory address field. This corresponds to four, eight, and eight possible values, respectively. Therefore, ModRM is encoded as a 20-dimensional one-hot vector.

- **SIB**: SIB (Scale-Index-Base) is another one-byte field divided into three segments: two bits for the scale factor, three bits for the index, and three bits for the base register. This results in four, eight, and eight possible values, respectively, leading to a 20-dimensional one-hot vector for SIB encoding.
- **Displacement**: In x86-64 assembly, displacement is an offset value used in memory addressing to calculate the effective memory address. It is part of the instruction's addressing mode and is added to a base or index register to determine the actual memory address being accessed. It is represented using a 64-dimensional binary vector.
- **Immediate**: An immediate value is a constant value that is directly encoded as part of the instruction itself. Unlike a displacement, which is used for memory addressing, an immediate is used as an operand directly in the instruction. It is represented as a binary vector with 64 dimensions.
- **Option**: Since prefix, ModRM, SIB, displacement, and immediate fields are optional, a 5-dimensional binary vector is used to indicate their presence or absence.

The final encoded vector for each instruction is formed by concatenating all these components, resulting in a 438-dimensional vector.

Since a single CFG node may contain multiple instructions, the instruction vectors within a node are aggregated using a general aggregation function, which can be mean pooling, max

pooling, or a learnable attention-based mechanism:

$$\mathbf{E}_{\text{node}} = \mathcal{A}(\mathbf{E}_{\text{instr}}^{(1)}, \mathbf{E}_{\text{instr}}^{(2)}, \dots, \mathbf{E}_{\text{instr}}^{(n)}) \quad (1)$$

where n is the number of instructions within the node, \mathcal{A} represents the aggregation function (which can be mean, max, or attention-based), and $\mathbf{E}_{\text{node}} \in \mathbb{R}^{438}$ is the final aggregated vector for the node.

Dimensionality Reduction: The high-dimensional 438-dimensional vectors are reduced to a 64-dimensional latent representation using an autoencoder. The autoencoder consists of an encoder-decoder architecture, where the encoder reduces the dimensionality, and the decoder reconstructs the original vector. The reduced representation retains the most relevant information for malware detection.

The autoencoder is optimized using a mean squared error (MSE) loss function:

$$L_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^N \|\mathbf{E}_{\text{instr}}^{(i)} - g_{\phi}(f_{\theta}(\mathbf{E}_{\text{instr}}^{(i)}))\|^2 \quad (2)$$

where N is the number of training samples, f_{θ} is the encoder function, and g_{ϕ} is the decoder function. Once trained, the encoder reduces the 438-dimensional instruction vector into a compact 64-dimensional feature vector used as the final node feature embedding for the GNN model:

$$\mathbf{E}'_{\text{node}} = f_{\theta}(\mathbf{E}_{\text{node}}) \quad (3)$$

where $\mathbf{E}'_{\text{node}} \in \mathbb{R}^{64}$.

B. Graph Classification Using GNNs

After generating node feature embeddings, the next step is to classify the graph using a GNN model. The classification process involves three key stages: node embedding through GNN layers, graph-level representation generation using a readout layer, and final classification using a downstream classifier.

1) Node Embedding with GNN Layers: A GNN model generates node embeddings by iteratively aggregating information from neighboring nodes through K layers [28]. The node embedding at layer l is computed using the following general update rule:

$$h_i^{(l)} = \text{UPDATE} \left(h_i^{(l-1)}, \text{AGG} \left(\{h_j^{(l-1)} : j \in \mathcal{N}(i)\} \right) \right) \quad (4)$$

where $h_i^{(l)}$ is the embedding of node i at layer l , $h_i^{(0)}$ represents the initial node feature vector, $\mathcal{N}(i)$ is the set of neighboring nodes of node i , $\text{AGG}(\cdot)$ is an aggregation function that collects information from neighboring nodes, and $\text{UPDATE}(\cdot)$ is an update function that combines the aggregated information with the node's current embedding.

The aggregation function $\text{AGG}(\cdot)$ and update function $\text{UPDATE}(\cdot)$ can vary depending on the specific GNN model. For instance, in the case of Graph Convolutional Networks (GCNs), the aggregation is based on a normalized summation

of neighboring node features, and the update step applies a linear transformation followed by a non-linearity:

$$h_i^{(l)} = \sigma \left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i d_j}} W^{(l)} h_j^{(l-1)} \right)$$

where $\mathcal{N}(i)$ denotes the set of neighboring nodes of node i , $\sigma(\cdot)$ is a non-linear activation function such as ReLU, $W^{(l)}$ is the learnable weight matrix at layer l , and d_i and d_j denote the degrees of nodes i and j , respectively. The term $\frac{1}{\sqrt{d_i d_j}}$ represents a normalization factor that accounts for the degree of each node to ensure stability during training.

This message-passing mechanism allows the node embeddings to incorporate information from neighboring nodes within a K -hop neighborhood, enabling the model to capture local graph structure and node attribute interactions.

2) Readout Layer: Once the node embeddings are computed through k GNN layers, a readout layer generates a fixed-size graph-level representation by aggregating the final node embeddings. A general readout function can be defined as:

$$h_G = \mathcal{R}(\{h_i^{(k)} : i \in V\}) \quad (5)$$

where h_G is the graph-level representation, V is the set of nodes in the graph, and $\mathcal{R}(\cdot)$ represents the readout function. Common readout functions include Mean pooling, Sum pooling, Max pooling, Attention-based pooling, Set2Set, and Sort Pooling.

3) Graph Classification: The graph-level representation h_G is passed to a classifier, which is typically a fully connected neural network (FCNN) followed by a softmax activation to produce class probabilities:

$$\hat{y} = \text{softmax}(W_c h_G + b_c) \quad (6)$$

where \hat{y} is the predicted class probability vector, W_c is the weight matrix for the classifier, and b_c is the bias term for the classifier.

This process enables the GNN model to learn a graph-level representation that captures both node features and structural information.

C. GNN Explanation Techniques

While GNNs have demonstrated strong performance in graph-based learning tasks, their decision-making processes often remain opaque. Explainability methods aim to interpret how GNN models arrive at their predictions by identifying the most influential nodes, edges, or subgraphs within the input graph. In this study, we employ five state-of-the-art explainers: GNNExplainer, PGExplainer, and CaptumExplainer with three attribution methods: Integrated Gradients, Guided Backpropagation, and Saliency. All these explainability techniques assign weights to the edges based on their importance to the decision made by the GNN model. After evaluating their performance, we select the two best-performing explainers and propose a new explainer based on an aggregation method that combines the edge rankings from these two explainers to generate more consistent and informative subgraphs.

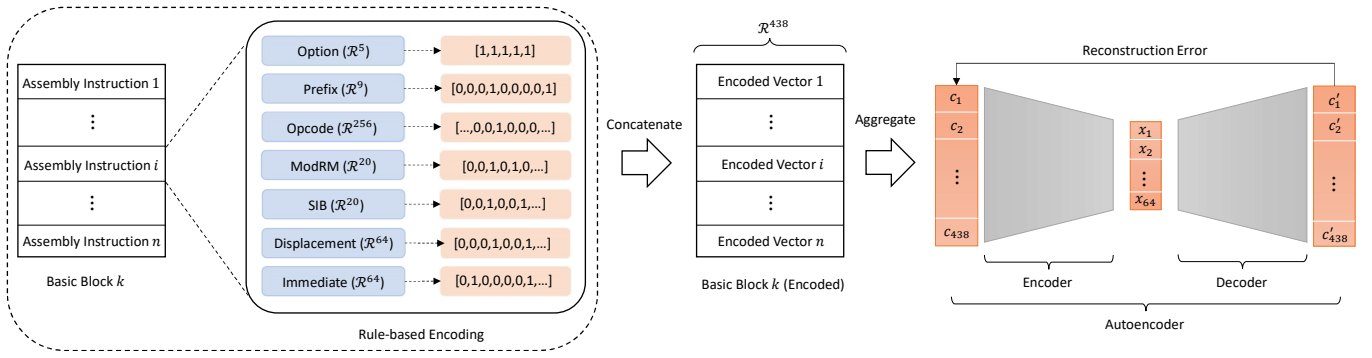


Fig. 2: Embedding process for assembly instructions.

GNNExplainer and PGExplainer are based on the key idea of identifying a subgraph that maximizes the mutual information (MI) with the model's prediction. This objective can be formulated as:

$$\max_{G_s} MI(Y, (G_s, X_s)) = H(Y) - H(Y | G = G_s, X = X_s) \quad (7)$$

where G_s and X_s represent the explanatory subgraph and its associated node features, respectively, Y is the prediction of the GNN model, and $H(\cdot)$ denotes the entropy function. The goal is to find the most informative subgraphs and node features that contribute to the model's prediction. Since $H(Y)$ is constant for a trained GNN, maximizing the mutual information reduces to minimizing the conditional entropy. GNNExplainer proposes approximating the conditional entropy using the cross-entropy loss between the true class label and the model's prediction, resulting in the following optimization objective for binary classification, which can be optimized using gradient descent:

$$\min_M - \sum_{c=1}^2 \mathbb{1}[y=c] \log P_{\Phi}(Y=y | G=A_c \odot \sigma(M), X=X_c) \quad (8)$$

where c is the class label, y is the predicted label, A_c is the adjacency matrix of the input graph, M is the learnable mask that assigns a weight to each edge of the input graph based on its importance for the model's prediction, σ is the sigmoid function that maps the weight to the range $[0, 1]$, and X_c is the corresponding node feature set. This optimization is performed separately for each input sample to generate its corresponding explanation. After optimization, the top-ranked edges based on their weights are selected to form the important subgraph.

PGExplainer uses the following cross-entropy loss to train the explainer on multiple samples. Once trained, the explainer model can be applied to other samples without requiring retraining. For binary classification, the objective is formulated as:

$$\min_{\Psi} - \sum_{i \in I} \sum_{k=1}^K \sum_{c=1}^2 P_{\Phi}(Y=c | G=G_o^{(i)}) \log P_{\Phi}(Y=c | G=\hat{G}_s^{(i,k)}) \quad (9)$$

where $G_o^{(i)}$ is the original input graph for sample i , $\hat{G}_s^{(i,k)}$ is the sampled explanatory subgraph for sample i in the k -th sampling step, and Ψ represents the parameters of the explainer model.

In addition to GNNExplainer and PGExplainer, we employed CaptumExplainer to enhance the interpretability of GNN models. CaptumExplainer provides a framework for attributing the model's predictions to individual nodes, edges, and features, allowing a detailed understanding of how the model processes graph data. We utilized three key attribution methods within CaptumExplainer: Integrated Gradients, Saliency, and Guided Backpropagation.

Integrated Gradients computes the contribution of each input feature by integrating the gradients of the model's output with respect to the input along a path from a baseline to the actual input. This method satisfies two key axioms: sensitivity, which ensures that attributions reflect the difference in the model's outputs between the baseline and the input, and implementation invariance, which guarantees consistent attributions across functionally equivalent models.

Saliency measures the influence of each input feature by calculating the gradient of the model's output with respect to the input, effectively performing a first-order Taylor expansion. The magnitude of these gradients reflects the importance of each feature in the model's decision.

Guided Backpropagation refines the saliency method by modifying the backpropagation process to propagate only non-negative gradients through ReLU activations, thereby highlighting the most impactful and positively contributing features. These three methods provide complementary insights into the model's decision-making process, enabling a deeper understanding of the structural and feature-based factors that drive GNN predictions.

D. RankFusion Explainer: An Edge Ranking Aggregation Strategy

To enhance the accuracy of explainers, we propose a novel aggregation method called RankFusion explainer that combines the edge rankings from two explainers. The goal is to improve the quality of explanations across different sparsity levels.

Algorithm 1 RankFusion Explainer

```
1: Input: Graph  $G = (V, E)$ , Edge weights from Explainer 1 ( $W_1$ ), Edge weights from Explainer 2 ( $W_2$ ), Accuracy of Explainer 1 ( $A_1$ ) and Explainer 2 ( $A_2$ ), Threshold percentage ( $T$ )
2: Output: New edge weights ( $W_{\text{fusion}}$ )
3:  $n \leftarrow |E|$ 
4: Threshold  $\tau \leftarrow T \cdot n/100$ 
5:  $W_{\text{fusion}} \leftarrow \emptyset$ 
6: for each edge  $e$  in  $W_1 \cap W_2$  do
7:    $d \leftarrow |W_1[e] - W_2[e]|$ 
8:   if  $d \leq \tau$  then
9:      $W_{\text{fusion}}[e] \leftarrow \max(W_1[e], W_2[e])$ 
10:  else
11:    if  $A_1 \geq A_2$  then
12:       $W_{\text{fusion}}[e] \leftarrow W_1[e]$ 
13:    else
14:       $W_{\text{fusion}}[e] \leftarrow W_2[e]$ 
15:    end if
16:  end if
17: end for
18: return  $W_{\text{fusion}}$ 
```

The RankFusion process involves reassigning edge weights to their descending rank order for each explainer independently. For each pair of similar edges present in both explainers, we compute the absolute difference between their ranks. If this difference is smaller than a predefined threshold, set as a percentage of the total number of edges in the graph, we assign the rank of the edge as the maximum of the two ranks. However, if the difference exceeds the threshold, the rank of the edge is assigned based on the rank from the explainer with the highest accuracy at the corresponding explanation percentage.

To evaluate the RankFusion explainer, we retain a certain percentage of edges with the highest aggregated weights, reconstruct the graph, and feed it into a trained GNN model. The accuracy of the GNN on this reduced graph serves as the performance measure of the RankFusion explainer. This strategy ensures that the RankFusion explainer retains the most meaningful edges, combining the strengths of both explainers to produce more consistent and accurate explanations across varying sparsity levels. The RankFusion algorithm is presented in Algorithm 1.

IV. EXPLAINER SUBGRAPH EVALUATION AND CONSTRUCTION

An effective GNN explainer should identify a subgraph that preserves the model’s predictive capability while revealing the key structural features driving the prediction. Ideally, an important subgraph should produce a prediction outcome close to that of the original input. At the same time, the model’s performance should drop significantly, when it processes the remaining part of the graph after the important subgraph is removed. This behavior confirms that the identified subgraph

contains the critical features influencing the model’s decision-making. To assess the quality of explainer-generated subgraphs, we rely on two primary evaluation metrics: explainer accuracy and fidelity.

Explainer accuracy measures the model’s performance, when using only the subgraph identified by the explainer as input. A well-constructed subgraph explanation should preserve the model’s original classification performance and accuracy. This metric operates under the assumption that the identified subgraph retains the most influential features necessary for accurate predictions.

Fidelity measures the contribution of the important subgraph to the model’s prediction. It evaluates how much the prediction changes, when the important or unimportant parts of the graph are removed. Fidelity is defined two complementary forms:

$$Fidelity^+ = \frac{1}{N} \sum_{i=1}^N (f(G) - f(G - G_s)), \quad (10)$$

$$Fidelity^- = \frac{1}{N} \sum_{i=1}^N (f(G) - f(G_s)), \quad (11)$$

where G_s is the important subgraph, G is the original input, and f is the trained GNN model. $Fidelity^+$ measures the prediction difference between the original input and its unimportant part, capturing the contribution of the removed edges to the model’s decision. Conversely, $Fidelity^-$ compares the prediction between the original graph and the important subgraph, reflecting the predictive capacity of the extracted subgraph.

In addition to accuracy and fidelity, consistency is an essential property for evaluating the reliability of explainers. It reflects the stability of an explanation, when the input graph is subjected to small structural perturbations that do not alter the model’s prediction [29]. A consistent explainer should yield similar explanations for slightly modified graphs, thereby enhancing interpretability and trustworthiness.

Let $G = (V, E)$ be an input graph, with $\mathcal{M}(G)$ denoting the prediction of a trained GNN model. To evaluate consistency, we generate a set of perturbed graphs $\mathcal{G}_{\text{pert}} = \{G'_1, G'_2, \dots, G'_m\}$, where each G'_i is created by randomly removing a small subset of nodes or edges from G .

The number of removed elements is computed as:

$$|S| = \lceil \log(|X|) + |X| \cdot p \rceil, \quad (12)$$

where $X \in \{V, E\}$, $S \subseteq X$, and $p \in (0, 1)$ is a perturbation ratio.

Each perturbed graph G'_i is retained for consistency evaluation if it meets the following conditions:

- The model prediction is unchanged:

$$\mathcal{M}(G'_i) = \mathcal{M}(G) \quad (13)$$

- The graph-level embeddings remain similar:

$$\cos(\phi(G), \phi(G'_i)) < \tau, \quad (14)$$

where $\phi(\cdot)$ denotes the model's embedding function, and τ is a threshold for cosine similarity.

For each valid G'_i , we compute the explainer's fidelity scores $\mathcal{F}^+(G'_i)$ and $\mathcal{F}^-(G'_i)$. The consistency of the explainer is measured by the variability of these scores:

$$\Delta^+ = \max_i \mathcal{F}^+(G'_i) - \min_i \mathcal{F}^+(G'_i) \quad (15)$$

$$\Delta^- = \max_i \mathcal{F}^-(G'_i) - \min_i \mathcal{F}^-(G'_i) \quad (16)$$

Smaller values of Δ^+ and Δ^- indicate greater consistency, as they reflect less variation in the explanation quality under small, non-disruptive perturbations.

1) *Top-Edge Selection (TES)*: Conventional methods for subgraph extraction rank edges according to their weights and select the highest-ranked edges to construct a subgraph of a specific size relative to the original graph. We refer to this approach as TES and use it as our baseline method.

However, extracting subgraphs using TES often results in explanations composed of multiple disconnected components, whereas original graphs typically consist of a single connected component. This discrepancy can reduce the reliability of evaluation metrics such as accuracy and fidelity, since the target GNN is trained on connected graphs and may not perform well on fragmented subgraphs.

2) *Greedy Edge-wise Composition (GEC)*: To address the limitations of TES, we propose Greedy Edge-wise Composition (GEC), an alternative subgraph extraction technique designed to construct strongly connected subgraphs with high cumulative edge importance. This method operates on a weighted graph representation $G' = (V', E', \mathbf{S}_E)$, where \mathbf{S}_E is the edge importance scores.

The process begins by selecting the edge with the highest importance weight:

$$e_{\max} = \arg \max_{e \in E'} \alpha_e, \quad (17)$$

where $\alpha_e \in \mathbf{S}_E$ denotes the importance weight of edge e . The two nodes incident to e_{\max} are added to the selected node set V_{selected} , and the edge itself is added to the selected edge set E_{selected} .

GEC then proceeds iteratively, at each step selecting the next highest-weight edge that connects to the current set of selected nodes:

$$e_{\text{next}} = \arg \max_{\substack{e \in E' \\ e \text{ connects to } V_{\text{selected}}}} \alpha_e. \quad (18)$$

The corresponding node(s) from e_{next} not already in V_{selected} are added to the node set, and e_{next} is added to the edge set. This greedy procedure is repeated until a predefined number of edges is included.

Let k be the target number of edges to select. The final extracted subgraph is:

$$G_{\text{extracted}} = (V_{\text{selected}}, E_{\text{selected}}), \quad \text{where } |E_{\text{selected}}| = k. \quad (19)$$

By prioritizing both edge weight and structural connectivity, GEC ensures that the selected subgraph retains the

most informative and influential components of the original graph. This results in more faithful and robust explanations, ultimately supporting a more reliable interpretation of the model's decision-making process.

V. RESULTS AND ANALYSIS

For our experiments, we selected 1,117 malicious samples from BODMAS [30], 1,029 malicious samples from PMML [31], and 510 benign samples from DikeDataset [32], providing a balanced dataset for evaluating the effectiveness of the explainers across various malware families and sample types. To recover CFGs dynamically, we employ the *angr* library [33]–[35], a Python-based binary analysis tool. *angr* constructs graphs by integrating both symbolic execution and constraint solving, enabling comprehensive and precise CFG recovery. The dynamically generated CFGs used in our experiments are available for public access here.

We use a three-layer GCN with a 20% dropout rate for malware detection. The detection model achieved strong performance, with an accuracy of 94.74% and an F1 score of 96.82%. These results highlight the effectiveness of the proposed framework, which combines dynamically generated CFGs, node feature embeddings based on assembly instructions, and a GNN-based detection model to accurately differentiate between benign and malicious samples.

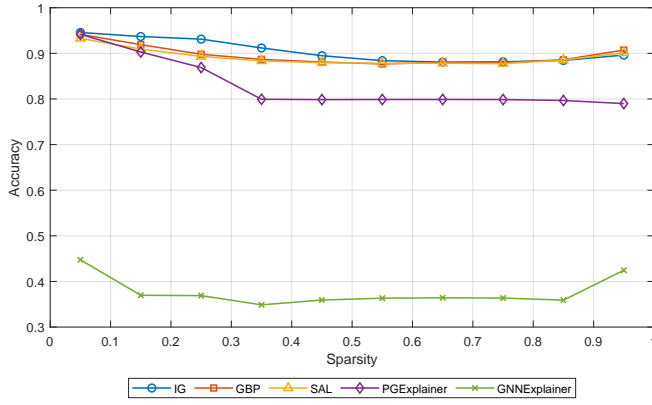
As previously discussed, the performance of the explainers is evaluated using the Accuracy, Fidelity, and Consistency metrics. The remainder of this section presents a detailed comparison of five explainers: GNNExplainer, PGExplainer, and three variants of CaptumExplainer based on different attribution methods: Integrated Gradients, Guided Backpropagation, and Saliency. The evaluation is conducted through two different subgraph extraction techniques: TES and GEC, based on these three metrics. Additionally, our proposed RankFusion explainer is evaluated by selecting the top two best-performing explainers.

Figure 3 presents a comparative analysis of model accuracy based on different explainability techniques and subgraph extraction methods. The x-axis denotes subgraph sparsity, defined as the percentage of edges retained from the original graph, ranging from 5% to 95%. The y-axis indicates the accuracy of the trained GCN model, when evaluated on the extracted subgraphs at each sparsity level.

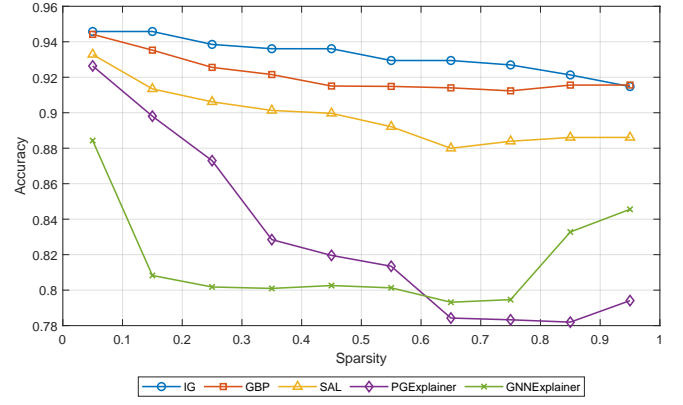
For clarity, the abbreviations used in all figures are as follows: IG (Integrated Gradients), GBP (Guided Backpropagation), SAL (Saliency), and RF (RankFusion explainer).

As illustrated in Figure 3, the GEC-based subgraph extraction method consistently outperforms the TES approach across all explainers. The performance improvement is particularly notable in the case of GNNExplainer, where the GEC method achieves nearly double the accuracy of TES, when the subgraph retains less than 90% of the original edges.

Moreover, regardless of the extraction method (TES or GEC), the ranking of explainers in terms of classification accuracy remains consistent. CaptumExplainer variants (specifically IG, GBP, and SAL) demonstrate the highest performance,



(a) TES-based subgraph extraction.



(b) GEC-based subgraph extraction.

Fig. 3: Experimental comparison of explainers.

followed by PGExplainer, with GNNExplainer yielding the lowest accuracy.

We assess the performance of the explainers using the $Fidelity^-$, $Fidelity^+$, and Consistency metrics across different sparsity levels. Figure 4 presents the $Fidelity^-$ and $Fidelity^+$ results under the GEC approach. In each subplot, the x-axis represents the sparsity level, defined as the percentage of edges retained from the original graph, and the y-axis indicates the fidelity score. The solid lines show the fidelity values computed from the original, unperturbed graphs for each explainer and sparsity level.

To assess consistency in practice, we generated 20 perturbed versions of each input graph: 10 with random node removal and 10 with random edge removal. In our implementation, the perturbation ratio was set to $p = 0.01$. Each perturbed graph was retained for analysis only if it preserved the model’s original prediction and the cosine similarity between the graph-level embeddings of the original and perturbed graphs was below a selected threshold. We performed a sweep over similarity thresholds τ ranging from 0.01 to 0.03 in increments of 0.0025 to identify valid perturbations. For each explainer, we visualized the variability of $Fidelity^+$ and $Fidelity^-$ scores across all valid perturbations. The solid lines represent fidelity values computed on the original (unperturbed) graphs, while the shaded regions reflect the range between the minimum and maximum fidelity scores observed across the valid perturbations.

The results show that $Fidelity^-$ values are generally low and stable for explainers, when subgraphs are extracted using GEC. This is particularly evident for GNNExplainer, whose performance stabilizes significantly under this approach. According to the definition of $Fidelity^-$, lower values are desirable, and this trend is clearly observed for the best-performing explainers (IG, GBP, and SAL), which consistently yield lower $Fidelity^-$ scores.

In contrast, $Fidelity^+$ values demonstrate a consistent pattern across sparsity levels. The top explainers (IG, GBP, and SAL) achieve higher scores, indicating a substantial change

in model prediction when the important subgraph is removed. This suggests that the identified important subgraph indeed contains the critical predictive structure. Furthermore, the consistency bands in the $Fidelity^+$ plot are generally narrow, indicating that explanations remain stable across perturbations. Although $Fidelity^-$ bands are somewhat wider, the results still indicate reasonable stability, particularly for the top explainers. These findings highlight the advantage of using GEC in conjunction with effective explainers for generating reliable and robust explanations.

To assess the effectiveness of the proposed RankFusion explainer, we investigate whether selecting and combining the outputs of the top-performing explainers can lead to improved subgraph quality and model performance. Based on the previous analysis across accuracy, fidelity, and consistency metrics, the two most effective explainers identified were Integrated Gradients and Guided Backpropagation. RankFusion is applied by aggregating the edge importance rankings produced by these two explainers using the methodology described earlier.

Figure 5 compares the model’s accuracy when using subgraphs generated by IG, GBP, and their aggregated result. All subgraphs in this evaluation are extracted using the GEC method. The x-axis represents the sparsity level, and the y-axis shows the accuracy obtained by feeding the resulting subgraph into the pre-trained GCN. Across all sparsity levels, the aggregated explanation consistently achieves higher accuracy than either of the individual explainers. While the improvement over the best individual explainer (IG) is modest, the results clearly support the hypothesis that selecting and combining the outputs of strong explainers leads to more informative subgraphs.

Figure 6 visually compares the explanations produced by the RankFusion explainer, IG explainer, and GBP explainer on the same CFG of a malicious sample. In each subfigure, the colored region highlights the top 5% of ranked edges along with their corresponding nodes, forming a connected subgraph that is considered most informative by the respective explainer.

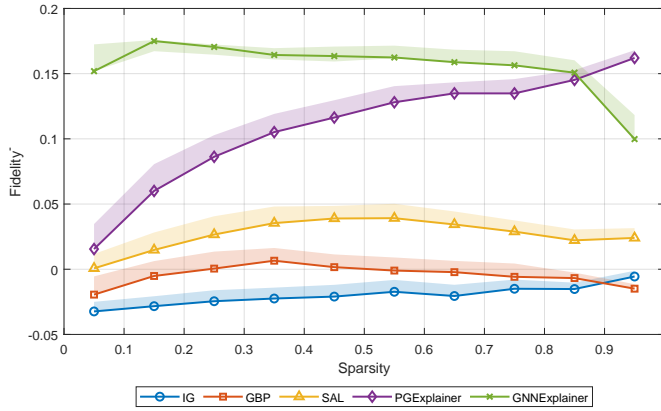


Fig. 4: Fidelity performance of explainability methods under the GEC approach.

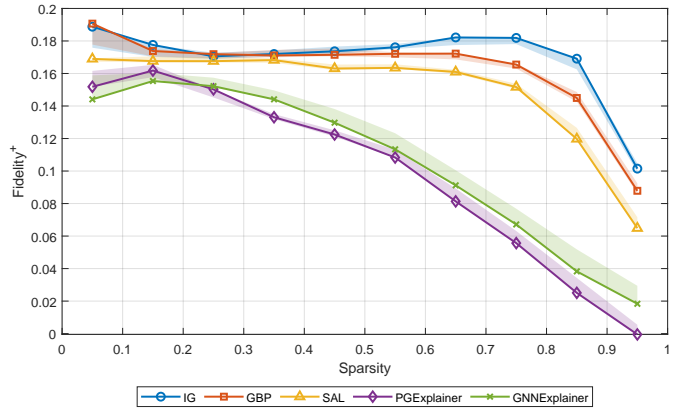


Fig. 5: Accuracy comparison of individual top explainers and the RankFusion explainer.

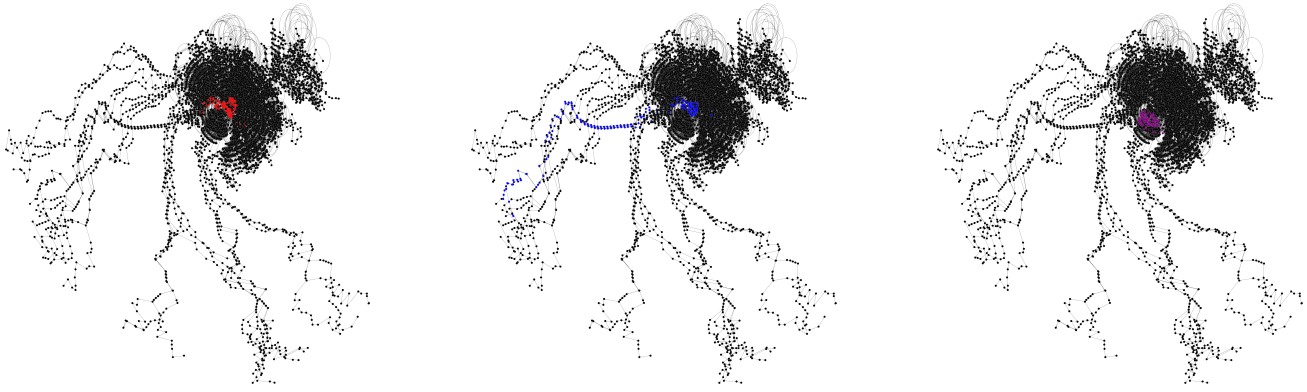
VI. CONCLUSION

This paper presents a comprehensive framework for malware detection using dynamically generated CFGs. Node-level assembly instructions are embedded into a continuous vector space through an embedding approach that combines rule-based encoding with autoencoder-based learning. These representations are then used as input to a GNN-based classifier for malware detection. To enhance the interpretability of the model, five explainers are employed: GNNExplainer, PGExplainer, and CaptumExplainer with three attribution methods, namely Integrated Gradients, Guided Backpropagation, and Saliency. In addition, we propose RankFusion, an aggregated explainer that selects and integrates the outputs of the top-performing explainers to improve the quality of generated explanations. We also introduce GEC as a connectivity-aware subgraph extraction strategy that mitigates the shortcomings of traditional top-edge selection. Experimental results demonstrate the strong detection performance of the framework, with an accuracy exceeding 94% and an F1 score above 96%. Further analysis based on Fidelity, Accuracy, and Consistency metrics shows that the combination of RankFusion and GEC

produces more stable, interpretable, and accurate explanations, reinforcing the value of incorporating both explainer aggregation and structurally coherent subgraph extraction in GNN-based malware detection.

REFERENCES

- [1] Y. Li, K. Xiong, T. Chin, and C. Hu, "A machine learning framework for domain generation algorithm-based malware detection," *IEEE Access*, vol. 7, pp. 32765–32782, 2019.
- [2] F. Manavi and A. Hamzeh, "A new method for ransomware detection based on pe header using convolutional neural networks," in *2020 17th international ISC conference on information security and cryptology (ISCISC)*, pp. 82–87, IEEE, 2020.
- [3] R. Frederick, J. Shapiro, and R. A. Calix, "A corpus of encoded malware byte information as images for efficient classification," in *2022 16th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, pp. 32–36, IEEE, 2022.
- [4] C. Li, Q. Lv, N. Li, Y. Wang, D. Sun, and Y. Qiao, "A novel deep framework for dynamic malware detection based on api sequence intrinsic features," *Computers & Security*, vol. 116, p. 102686, 2022.
- [5] A. Bensaoud and J. Kalita, "Cnn-lstm and transfer learning models for malware classification based on opcodes and api calls," *Knowledge-Based Systems*, p. 111543, 2024.
- [6] E. M. Dovom, A. Azmoodeh, A. Dehghantanha, D. E. Newton, R. M. Parizi, and H. Karimipour, "Fuzzy pattern tree for edge malware detection and categorization in iot," *Journal of Systems Architecture*, vol. 97, pp. 1–7, 2019.
- [7] H. Peng, J. Yang, D. Zhao, X. Xu, Y. Pu, J. Han, X. Yang, M. Zhong, and S. Ji, "Malgne: Enhancing the performance and efficiency of cfg-based malware detector by graph node embedding in low dimension space," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4881–4896, 2024.
- [8] Y. Sun, A. K. Bashir, U. Tariq, and F. Xiao, "Effective malware detection scheme based on classified behavior graph in iiot," *Ad Hoc Networks*, vol. 120, p. 102558, 2021.
- [9] A. Abusnaina, M. Abuhamad, H. Alasmay, A. Anwar, R. Jang, S. Salem, D. Nyang, and D. Mohaisen, "DI-fhmc: Deep learning-based fine-grained hierarchical learning approach for robust malware classification," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3432–3447, 2021.
- [10] X. Deng, Z. Wang, X. Pei, and K. Xue, "Transmalde: An effective transformer based hierarchical framework for iot malware detection," *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 1, pp. 140–151, 2024.
- [11] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, 2021.
- [12] C.-Y. Wu, T. Ban, S.-M. Cheng, T. Takahashi, and D. Inoue, "Iot malware classification based on reinterpreted function-call graphs," *Computers & Security*, vol. 125, p. 103060, 2023.



(a) RankFusion explainer result.

(b) Integrated Gradients explainer result.

(c) Guided Backpropagation explainer result.

Fig. 6: Visual comparison of the top 5% high-ranking subgraphs identified by three explainers on a malicious CFG.

- [13] X. Zhang, M. Zhang, Y. Zhang, M. Zhong, X. Zhang, Y. Cao, and M. Yang, "Slowing down the aging of learning-based malware detectors with api knowledge," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 902–916, 2023.
- [14] E. Amer, I. Zelinka, and S. El-Sappagh, "A multi-perspective malware detection approach through behavioral fusion of api call sequence," *Computers & Security*, vol. 110, p. 102449, 2021.
- [15] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning," *Computers & Security*, vol. 122, p. 102872, 2022.
- [16] H. Shokouhinejad, R. Razavi-Far, H. Mohammadian, M. Rabbani, S. Ansong, G. Higgins, and A. A. Ghorbani, "Recent advances in malware detection: Graph learning and explainability," *arXiv preprint arXiv:2502.10556*, 2025.
- [17] 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*, vol. 8, no. 2, pp. 957–970, 2021.
- [18] Y. Fang, C. Huang, M. Zeng, Z. Zhao, and C. Huang, "Jstrong: Malicious javascript detection based on code semantic representation and graph neural network," *Computers & Security*, vol. 118, p. 102715, 2022.
- [19] Z. Yu, S. Li, Y. Bai, W. Han, X. Wu, and Z. Tian, "Remsf: A robust ensemble model of malware detection based on semantic feature fusion," *IEEE Internet of Things Journal*, vol. 10, no. 18, pp. 16134–16143, 2023.
- [20] Y.-H. Chen, S.-C. Lin, S.-C. Huang, C.-L. Lei, and C.-Y. Huang, "Guided malware sample analysis based on graph neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 4128–4143, 2023.
- [21] S.-J. Bu and S.-B. Cho, "Triplet-trained graph transformer with control flow graph for few-shot malware classification," *Information Sciences*, vol. 649, p. 119598, 2023.
- [22] P. Feng, L. Gai, L. Yang, Q. Wang, T. Li, N. Xi, and J. Ma, "Dawngnn: Documentation augmented windows malware detection using graph neural network," *Computers & Security*, vol. 140, p. 103788, 2024.
- [23] W. Li, H. Tang, H. Zhu, W. Zhang, and C. Liu, "Ts-mal: Malware detection model using temporal and structural features learning," *Computers & Security*, vol. 140, p. 103752, 2024.
- [24] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in Neural Information Processing Systems (NIPS)*, vol. 32, 2019.
- [25] 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*, pp. 12241–12252, PMLR, 2021.
- [26] J. D. Herath, P. P. Wakodkar, P. Yang, and G. Yan, "Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 172–184, 2022.
- [27] D. Luo, W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang, "Parameterized explainer for graph neural network," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.
- [28] H. Shokouhinejad, R. Razavi-Far, G. Higgins, and A. A. Ghorbani, "Node-Centric Pruning: A novel graph reduction approach," *Machine Learning and Knowledge Extraction*, vol. 6, no. 4, pp. 2722–2737, 2024.
- [29] E. Hajiramezanali, S. Maleki, A. Tseng, A. BenTaieb, G. Scalia, and T. Biancalani, "On the consistency of GNN explainability methods," in *XAI in Action: Past, Present, and Future Applications*, 2023.
- [30] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "Bodmas: An open dataset for learning based temporal analysis of pe malware," in *2021 IEEE Security and Privacy Workshops (SPW)*, pp. 78–84, IEEE, 2021.
- [31] Practical Security Analytics LLC, "Pe malware machine learning dataset," <https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/>, 2024. Accessed: 2024-08-06.
- [32] G.-A. Iosif, "Dikedataset," <https://github.com/iosifache/DikeDataset>, 2021. Accessed on February 27, 2024.
- [33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.
- [34] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.
- [35] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmallice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.