Explainable Ensemble Learning for Graph-Based Malware Detection

Hossein Shokouhinejad[1], Roozbeh Razavi-Far[1], Griffin Higgins[1], Ali A Ghorbani[1]

[1*]University of New Brunswick, Faculty of Computer Science, 3 Bailey Dr., Fredericton, E3B5A3, New Brunswick, Canada.

Abstract

Malware detection in modern computing environments demands models that are not only accurate but also interpretable and robust to evasive techniques. Graph neural networks (GNNs) have shown promise in this domain by modeling rich structural dependencies in graph-based program representations such as control flow graphs (CFGs). However, single-model approaches may suffer from limited generalization and lack interpretability, especially in high-stakes security applications. In this paper, we propose a novel stacking ensemble framework for graph-based malware detection and explanation. Our method dynamically extracts CFGs from portable executable (PE) files and encodes their basic blocks through a two-step embedding strategy. A set of diverse GNN base learners, each with a distinct message-passing mechanism, is used to capture complementary behavioral features. Their prediction outputs are aggregated by a meta-learner implemented as an attention-based multilayer perceptron, which both classifies malware instances and quantifies the contribution of each base model. To enhance explainability, we introduce an ensemble-aware post-hoc explanation technique that leverages edge-level importance scores generated by a GNN explainer and fuses them using the learned attention weights. This produces interpretable, model-agnostic explanations aligned with the final ensemble decision. Experimental results demonstrate that our framework improves classification performance while providing insightful interpretations of malware behavior.

Keywords: Graph Neural Networks, Malware Detection, Control Flow Graphs, Stacking Ensemble Learning, Explainable AI.

1 Introduction

Malware, or malicious software, continues to pose a significant threat to modern computing environments, particularly in enterprise systems, critical infrastructure, and cloud platforms. As malware evolves in complexity and evasion techniques, conventional signature-based detection methods struggle to keep pace. To address these limitations, researchers have turned to machine learning and deep learning techniques, which offer the potential to detect novel or obfuscated malware through behavioral and structural analysis [1]. However, the effectiveness of such models largely depends on the quality of features and the representation of program behavior, necessitating advanced techniques capable of capturing the intricate dependencies within program execution.

Graph Neural Networks (GNNs) have recently emerged as powerful tools for learning from graph-structured data and have demonstrated considerable success in a range of cybersecurity tasks, particularly malware detection [2–7]. Unlike traditional neural networks, GNNs are specifically designed to operate on graph data by iteratively aggregating information from neighboring nodes and edges, allowing them to capture intricate structural and relational dependencies across the graph. This capability makes GNNs especially well-suited for modeling software behavior, which is often naturally represented as a graph [8]. In malware detection, GNNs have been applied to various graph-based representations of programs, including function call graphs [9], API call graphs [10], and most importantly, control flow graphs (CFGs) [11], each of which captures different behavioral characteristics. Among these representations, CFGs are particularly valuable because they illustrate the execution flow of a program in a structured and analyzable form. A CFG models basic blocks of instructions as nodes and the possible control transfers between them as directed edges. This abstraction captures both high-level logic and low-level execution patterns, enabling the identification of structural anomalies and irregular control paths that are often indicative of malicious behavior.When combined with GNNs, CFGs enable the extraction of rich representations that capture both local execution behaviors and global structural context, thereby improving the detection of sophisticated and evasive malware.

While individual GNN models have shown promising performance, leveraging ensemble learning can further improve generalization, robustness, and predictive performance [12–16]. Ensemble techniques combine the outputs of multiple base learners to reduce variance and avoid overfitting [17]. Among these methods, stacking stands out as a powerful strategy that trains a meta-learner to combine the predictions of multiple base GNN models. Each GNN in the ensemble may capture different aspects of the input graph due to architectural or training diversity, and the meta-learner can synthesize these diverse outputs into a more accurate and reliable prediction. This hierarchical learning paradigm has been underexplored in malware detection with GNNs and offers a promising direction for improving detection accuracy and resilience against evasive techniques [18–22].

In addition to achieving high classification performance, the explainability of malware detection models has become a critical research focus, particularly for applications in high-assurance systems where transparency and trust are essential [23, 24]. Understanding which parts of the CFG contribute most to a detection decision not

only helps validate the model's outputs but also provides actionable insights for security analysts and incident responders. GNNs, while effective in modeling structural patterns in program behavior, often lack interpretability due to their complex architecture. To address this, eXplainable Artificial Intelligence (XAI) techniques such as GNNExplainer [25], PGExplainer [26], SubgraphX [27], and CaptumExplainer [28] have been proposed to highlight influential nodes, edges, or subgraphs that drive predictions. When applied to a stacking ensemble (SE), explanation becomes more challenging, as the meta-learner integrates outputs from multiple base models, each potentially capturing different aspects of the graph.

In this study, we address the challenge of graph-based malware detection and interpretability through a SE learning framework. The approach begins with the dynamic extraction of CFGs from Portable Executable (PE) files. Each basic block within the CFG is then embedded using a two-step feature representation process. For the classification task, we employ a GNN-based SE in which the base learners consist of diverse GNN models, each utilizing a distinct message-passing strategy to ensure representational diversity. The prediction outputs of these base models serve as inputs to a meta-learner, implemented as a multi-layer perceptron (MLP) enhanced with an attention mechanism. This meta-learner not only performs the final classification but also produces attention scores that reflect the relative contribution of each base learner to the final decision. For model explainability, we adopt a post-hoc explanation approach by integrating a state-of-the-art GNN explainer to assign importance scores to nodes and edges within each base model. These individual explanations are then aggregated using the attention scores from the meta-learner, leading to the development of a novel explanation method tailored for the stacking framework. The main contributions of this study are as follows:

- We propose a novel SE framework for graph-based malware detection, where diverse GNN base learners capture complementary structural and semantic information from control flow graphs extracted from PE files.
- A meta-learner equipped with an attention mechanism is introduced to aggregate predictions from base learners, enabling both accurate classification and interpretable attribution of each model's contribution to the final decision.
- We develop a new post-hoc explanation method that combines state-of-the-art GNN explanation techniques with the attention scores from the meta-learner to generate enhanced, ensemble-aware interpretations of malware predictions.

The remainder of this paper is structured as follows: Section 2 reviews prior work on GNN-based malware detection, ensemble learning, and explainable GNNs. Section 3 presents the proposed SE framework, including the dynamic CFG extraction process, node feature embedding, base learner architecture, meta-learner design, and the aggregation-based explanation method. Section 4 reports the experimental setup, datasets, performance evaluation, and explainability analysis. Finally, Section 5 concludes the paper and discusses directions for future work.

2 Related Works

Recent research has increasingly focused on advancing malware detection through graph-based learning techniques, particularly those that model program behavior using CFGs. GNNs have shown strong performance in capturing the structural dependencies within such graphs, enabling robust classification of malicious software. At the same time, ensemble learning approaches, particularly stacking-based frameworks, have been explored to combine the strengths of multiple models for improved robustness and accuracy. As these models grow in complexity, the need for explainability has become more pronounced, especially for deployment in security-critical systems. This section reviews prior work across three major areas: GNN-based malware detection, stacking-based ensemble methods for malware analysis, and explainable GNN frameworks for interpreting malicious behavior.

Several studies have leveraged GNNs for malware detection using CFGs as input representations. Peng et al. [3] introduced MalGNE, a node embedding framework that begins by encoding assembly instructions using a rule-based vectorization scheme to handle the out-of-vocabulary issue. It employs aggregation and attention-based bidirectional LSTM layers to capture the sequential and semantic structure of instructions within basic blocks. These representations are then passed to a GNN for classification. In a related effort, Zhang et al. [11] proposed a few-shot malware classification model based on a triplet-trained graph transformer. Their method processes each malware sample as a CFG and learns embeddings that capture structural and semantic relationships among basic blocks. Using triplet loss, the model is trained to place similar samples closer in the embedding space while distancing dissimilar ones, thereby improving generalization in data-scarce scenarios.

In addition to GNN-based methods, several studies have employed SE techniques to enhance detection performance and address challenges such as data imbalance and feature redundancy. Li et al. [18] proposed a hybrid detection approach that combines information gain and principal component analysis (IG-PCA) for feature selection, followed by an SE framework. The model uses an attention-based meta-learner to integrate diverse base classifiers and adaptively weight their contributions. Similarly, Naeem et al. [19] presented a malware detection technique that transforms memory dumps into images and extracts handcrafted features, which are then classified using an ensemble of convolutional neural networks. A fully connected neural network serves as the meta-learner, enhancing the final decision. Another example is SEDMDroid [20], which combines static and dynamic features—such as permissions, API calls, network behavior, and system activity—using an SE of decision trees, support vector machines, and random forests, with a neural network acting as the meta-classifier. Vasan et al. [21] further explored ensemble learning for cross-architecture malware detection on IoT devices. Their method extracts opcode sequences from binaries compiled for various architectures, converts them into images, and processes them using deep learning models. The final predictions are aggregated using a fully connected neural network to improve detection performance across platforms.

As model interpretability becomes increasingly important, recent studies have proposed explanation techniques tailored for GNN-based malware classifiers. Herath et al. [29] presented a model-agnostic framework that identifies influential subgraphs

within CFGs and ranks node importance using a surrogate model trained on node embeddings. This approach helps visualize which parts of the graph contribute most to classification. Building on this line of work, the study in [2] introduced a dynamic CFG-based malware detection framework that employs a hybrid node embedding method combining rule-based encoding with autoencoder-derived features. After classification via a GNN, the framework applies various explanation techniques, including GNNExplainer, PGExplainer, and Captum with different attribution strategies. The authors also proposed a new explanation method, RankFusion, which aggregates scores from multiple explainers to generate more stable and informative attributions.

3 Proposed Method

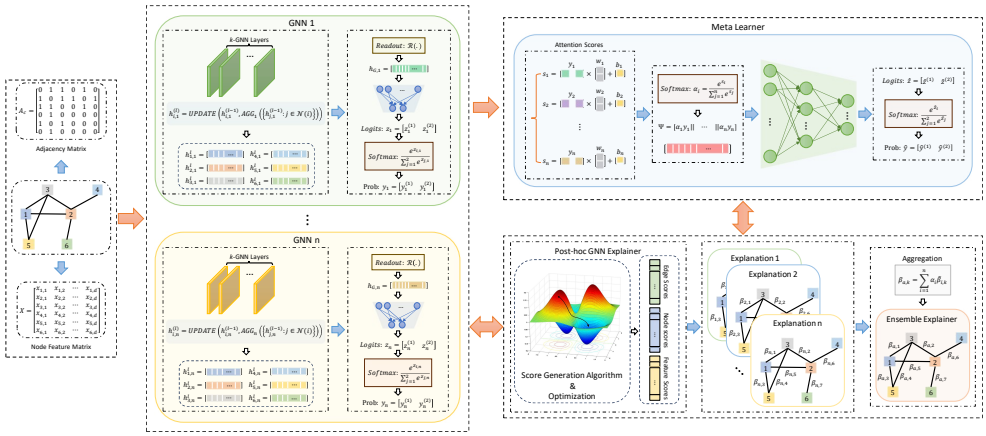


Fig. 1: Proposed framework for explainable malware detection using a stacking ensemble of GNNs.

This section presents our proposed framework for explainable malware detection using a GNN-based SE approach. The overall architecture is illustrated in Figure 1. Our method consists of four key components. First, CFGs are dynamically extracted from PE files, and a two-step embedding strategy is applied to encode semantic and structural features of each basic block. Second, multiple GNN models with distinct message-passing mechanisms are used as base learners to capture complementary aspects of the graph data. Third, the outputs of these base learners are fed into an attention-enhanced MLP that serves as the meta-learner, responsible for final classification and generating model-level attribution scores. Finally, we introduce a novel post-hoc explainability method that aggregates the edge-level explanation from individual base models, weighted by the attention scores produced by the meta-learner, to produce ensemble-aware interpretations. The following subsections provide a detailed description of each component.

3.1 Dynamic CFG Extraction and Node Feature Embedding

CFGs model the execution flow of a program, where nodes represent basic blocks, which are sequential groups of instructions with a single entry and a single exit point. Edges indicate possible transitions in control between these blocks. Static CFGs, constructed through disassembly, often miss important execution paths due to obfuscation, indirect jumps, or dynamic code loading. In contrast, dynamic CFGs are generated by tracing the actual runtime behavior of a program, capturing execution paths that may not be visible through static analysis. This makes them more effective for analyzing advanced or evasive malware. In our framework, we extract dynamic CFGs from PE files to build accurate graph representations for downstream feature embedding and classification.

To prepare each node in the dynamically extracted CFG for downstream learning, we employ a two-step feature embedding strategy, illustrated in Figure 2. The first step involves rule-based encoding of assembly instructions within each basic block, and the second step applies unsupervised dimensionality reduction using an autoencoder to obtain compact representations.

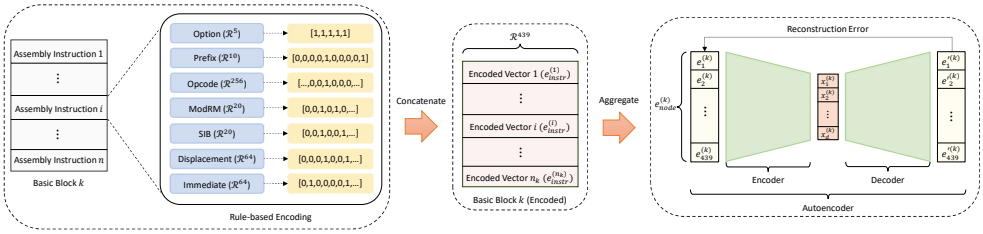


Fig. 2: The two-step node feature embedding process, including rule-based instruction encoding and autoencoder-based dimensionality reduction.

Each basic block consists of a sequence of x86-64 assembly instructions. Following a modified version of the approach proposed in [3], each instruction is decomposed into up to seven components: *prefix*, *opcode*, *ModRM*, *SIB*, *displacement*, *immediate*, and an *option* flag that indicates the presence of these fields. Each component is encoded using a fixed-length binary or one-hot representation, depending on its value space. For example, the opcode is encoded using a 256-dimensional one-hot vector, ModRM and SIB components are broken into subfields and encoded accordingly, and the displacement and immediate values are represented as 64-dimensional binary vectors. The concatenation of all encoded components results in a 439-dimensional feature vector for each instruction. This fine-grained encoding ensures full coverage of the x86-64 instruction set and resolves the out-of-vocabulary problem often encountered with text-based instruction embeddings.

Since a basic block may contain multiple instructions, we apply an aggregation function over their encoded vectors to produce a unified high-dimensional

representation for the node:

$$e_{\text{node}}^{(k)} = \text{AGG}\left(e_{\text{instr}}^{(1)}, e_{\text{instr}}^{(2)}, \ldots, e_{\text{instr}}^{(n_k)}\right) \tag{1}$$

where n_k is the number of instructions in the k-th node (basic block), $e_{\text{instr}}^{(i)} \in \mathbb{R}^{439}$ is the encoded vector of the i-th instruction, and $\text{AGG}(\cdot)$ denotes an aggregation function such as mean or max pooling. The result $e_{\text{node}}^{(k)} \in \mathbb{R}^{439}$ is the aggregated high-dimensional representation of node k.

To reduce the dimensionality of these vectors while preserving essential information, we train an autoencoder in an unsupervised manner. The autoencoder consists of an encoder-decoder architecture and is optimized using the mean squared error (MSE) loss:

$$L_{\text{MSE}} = \frac{1}{M} \sum_{i=1}^{M} \left\| e_{\text{instr}}^{(i)} - g_\phi(f_\theta(e_{\text{instr}}^{(i)})) \right\|^2 \tag{2}$$

where f_θ and g_ϕ denote the encoder and decoder functions, respectively, and M is the number of instruction samples used for training.

After training, the encoder maps each node's high-dimensional vector into a lower-dimensional latent space:

$$x^{(k)} = f_\theta(e_{\text{node}}^{(k)}) \tag{3}$$

where $x^{(k)} \in \mathbb{R}^d$, with $d \ll 439$, represents the compact embedding of the k-th node.

This two-step embedding process enables the generation of expressive and compact node features, which are well-suited for downstream graph learning tasks, particularly in settings where node-level supervision is unavailable.

By collecting the embeddings of all nodes, we construct the node feature matrix $X \in \mathbb{R}^{N \times d}$, where N is the total number of nodes in the graph. Each row of this matrix corresponds to one node's embedding:

$$X = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(N)} \end{bmatrix} \in \mathbb{R}^{N \times d} \tag{4}$$

3.2 Graph Neural Networks as Base Learners

GNNs are a class of neural models designed to learn from graph-structured data. In our setting, the input is a CFG in which each node represents a basic block with an associated feature vector $x^{(i)} \in \mathbb{R}^d$, and edges reflect control-flow relationships. GNNs operate through a process known as message passing, in which each node iteratively updates its representation by aggregating information from its neighbors.

The general message passing framework for a GNN layer is expressed as:

$$h_i^{(l)} = \text{UPDATE}^{(l)}\left(h_i^{(l-1)}, \text{AGG}^{(l)}\left(\left\{h_j^{(l-1)} : j \in \mathcal{N}(i)\right\}\right)\right) \tag{5}$$

where $h_i^{(l)}$ denotes the representation of node i at layer l, $\mathcal{N}(i)$ denotes the set of neighboring nodes of i, and AGG$(\cdot)$ and UPDATE$(\cdot)$ are differentiable functions responsible for aggregating and updating node features. At the initial layer, $h_i^{(0)} = x^{(i)}$, the input node feature vector.

In this work, we leverage multiple types of GNN models as base learners to introduce representational diversity, which is essential for the effectiveness of the SE. Different GNN architectures adopt distinct aggregation and update strategies, resulting in varied learning biases and node embeddings. By combining these diverse models, the meta-learner can learn complementary patterns that improve malware classification performance.

Among the many GNN variants proposed in the literature, three widely recognized and influential models are the Graph Convolutional Network (GCN) [30], Graph Isomorphism Network (GIN) [31], and Graph Attention Network (GAT) [32]. These models serve as prominent examples, each characterized by a distinct message passing mechanism, as outlined below.

Graph Convolutional Network (GCN)

GCN applies a normalized aggregation of neighboring features followed by a linear transformation:

$$h_i^{(l)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i d_j}} W^{(l)} h_j^{(l-1)}\right) \tag{6}$$

where $h_i^{(l)}$ is the embedding of node i at layer l, d_i and d_j denote the degrees of nodes i and j, respectively, $W^{(l)}$ is the learnable weight matrix at layer l, and $\sigma(\cdot)$ is a non-linear activation function such as ReLU.

Graph Isomorphism Network (GIN)

GIN is designed for high discriminative power and follows an MLP-based aggregation scheme:

$$h_v^{(l)} = \text{MLP}^{(l)}\left((1+\epsilon) \cdot h_v^{(l-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(l-1)}\right) \tag{7}$$

where $h_v^{(l)}$ is the embedding of node v at layer l, $u \in \mathcal{N}(v)$ refers to the neighbors of node v, ϵ is either a learnable parameter or a fixed scalar, and $\text{MLP}^{(l)}(\cdot)$ denotes a multi-layer perceptron. The use of summation ensures injective aggregation, making GIN highly expressive in distinguishing graph structures.

Graph Attention Network (GAT)

GAT introduces attention mechanisms to learn the importance of neighboring nodes:

$$h_v^{(l)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l)} W^{(l)} h_u^{(l-1)}\right) \tag{8}$$

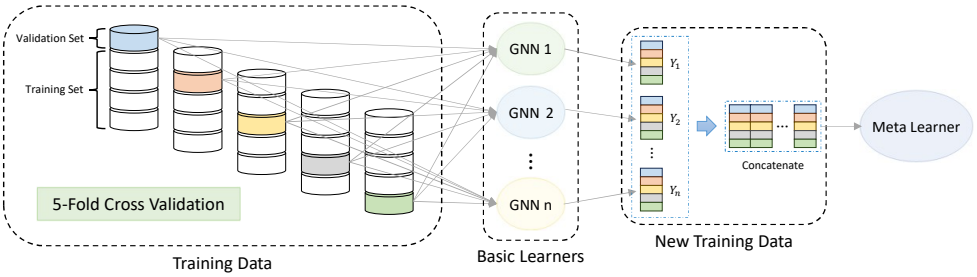


Fig. 3: SE model training process.

where $\alpha_{vu}^{(l)}$ is the attention coefficient between node v and neighbor u, computed based on their features, and $W^{(l)}$ is a learnable weight matrix. The attention mechanism enables the network to assign varying influence to different neighbors during aggregation.

After several rounds of message passing, node embeddings capture local structural and feature information. To perform graph classification, a readout function is applied to aggregate node-level embeddings into a single graph-level vector. Common readout strategies include global mean pooling, max pooling, sum pooling, and more advanced techniques such as Set2Set or attention-based pooling. The resulting graph embedding, denoted as h_G, serves as a compact representation of the entire graph.

This graph-level representation is then fed into a classifier, which is typically implemented as a simple MLP. For binary classification tasks such as malware detection, the final layer of the MLP outputs a two-dimensional logit vector $z = [z^{(1)}, z^{(2)}] \in \mathbb{R}^2$, where each component corresponds to one of the two classes (benign or malicious). A softmax function is subsequently applied to convert these logits into a probability distribution $y = [y^{(1)}, y^{(2)}]$, where $y^{(1)}$ and $y^{(2)}$ indicate the predicted probabilities for the benign and malicious classes, respectively. The final predicted label corresponds to the class with the higher probability.

Using different GNN architectures as base learners enriches the feature space available to the meta-learner. Each GNN type captures distinct structural and semantic patterns due to differences in aggregation functions, update mechanisms, and learning biases. For malware detection, where subtle behavioral variations in control flow graphs can be indicative of malicious activity, such architectural diversity enhances both robustness and classification performance. The meta-learner can leverage these complementary perspectives to improve generalization and decision accuracy.

3.3 Attention-Based Stacking Ensemble Architecture

Stacking ensemble (SE) learning is a powerful strategy for improving prediction accuracy and robustness by leveraging the complementary strengths of multiple models. Rather than relying on a single model's output, stacking combines the predictions of several base learners through a secondary model, known as the meta-learner, to produce a more reliable final decision. This approach enhances generalization and reduces overfitting by mitigating individual model biases and variances. In our work,

we adopt an SE framework that integrates multiple GNN-based base learners with an attention-based meta-learner. The training procedure consists of three main phases: (i) independently training each base learner using cross-validation, (ii) generating a new meta-training dataset from the base learners' validation predictions, and (iii) training the meta-learner using this aggregated information. The overall process is illustrated in Figure 3.

Let $\mathcal{D}_{\text{train}}$ denote the original training dataset, and suppose we use 5-fold cross-validation. The dataset is partitioned into 5 non-overlapping subsets $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(5)}$. Let GNN_i represent the i-th base learner. The following steps are applied to construct the training set for the meta-learner:

1. **Fold-wise training:** For each base learner GNN_i, and each fold $f = 1, \ldots, 5$, a model is trained on training set $\mathcal{D}_{\text{train}} \setminus \mathcal{D}^{(f)}$ and evaluated on validation set $\mathcal{D}^{(f)}$.
2. **Prediction on validation fold:** The trained model $\text{GNN}_i^{(f)}$ produces predicted probability vectors for samples in $\mathcal{D}^{(f)}$. Let:

$$\widehat{Y}_i^{(f)} = \text{GNN}_i^{(f)} \left(\mathcal{D}^{(f)} \right) \in \mathbb{R}^{|\mathcal{D}^{(f)}| \times 2} \tag{9}$$

3. **Row-wise aggregation:** After all 5 folds, the predictions are stacked row-wise to form the complete prediction matrix for i-th base learner :

$$Y_i = \begin{bmatrix} \widehat{Y}_i^{(1)} \\ \widehat{Y}_i^{(2)} \\ \vdots \\ \widehat{Y}_i^{(5)} \end{bmatrix} \in \mathbb{R}^{|\mathcal{D}_{\text{train}}| \times 2} \tag{10}$$

After constructing $\widehat{Y}_i$ for all n base learners, we horizontally concatenate them to form the input matrix for the meta-learner:

$$Y = [Y_1 \,\|\, Y_2 \,\|\, \cdots \,\|\, Y_n] \in \mathbb{R}^{|\mathcal{D}_{\text{train}}| \times 2n} \tag{11}$$

Each row of Y is the concatenation of predicted probabilities from all base learners for a single sample, and serves as input to the meta-learner.

After hyperparameter tuning and construction of the meta-training dataset through 5-fold cross-validation, each GNN base learner is retrained on the entire original training set $\mathcal{D}_{\text{train}}$ (i.e., the union of all folds) to produce its final model. These final models are then used to generate predictions on the test set for evaluation by the meta-learner.

Let the input vector for a given sample be denoted as:

$$y = [y_1 \,\|\, y_2 \,\|\, \cdots \,\|\, y_n] \in \mathbb{R}^{2n} \tag{12}$$

where $y_i \in \mathbb{R}^2$ is the prediction vector from base learner GNN_i.

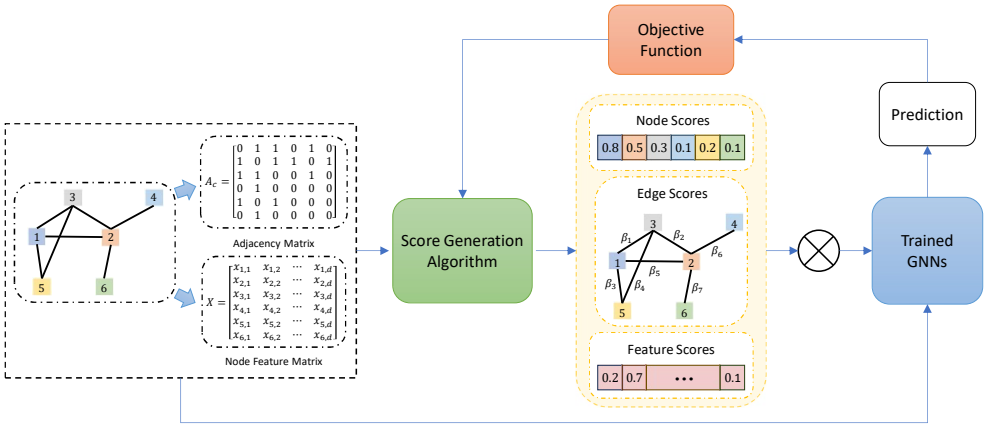


Fig. 4: Post-hoc explainability process for interpreting GNN predictions.

Each y_i is passed through a learnable linear transformation to compute an attention score:

$$s_i = w_i^\top y_i + b_i, \quad \text{for } i = 1, \ldots, n \tag{13}$$

where $w_i \in \mathbb{R}^2$ and $b_i \in \mathbb{R}$ are learnable parameters. These scores are normalized via softmax to obtain attention weights:

$$\alpha_i = \frac{\exp(s_i)}{\sum_{j=1}^{n} \exp(s_j)} \tag{14}$$

The final attention-aware input is constructed as:

$$\Psi = [\alpha_1 y_1 \,\|\, \alpha_2 y_2 \,\|\, \cdots \,\|\, \alpha_n y_n] \in \mathbb{R}^{2n} \tag{15}$$

This vector Ψ is then passed through an MLP, which outputs a two-dimensional logit vector $\hat{z} \in \mathbb{R}^2$, followed by a softmax operation to yield the predicted class probabilities $\hat{y} \in \mathbb{R}^2$. The learned attention weights α_i offer interpretability by indicating the relative contribution of each base learner to the ensemble's final decision.

3.4 Aggregation-Driven Explainability for Stacked GNNs

Post-hoc GNN explainability methods aim to interpret the predictions of trained GNN models by identifying important substructures in the input graph. These methods typically consist of three core components: (i) a pre-trained GNN model whose predictions are to be explained, (ii) a score generation mechanism that produces importance scores over graph elements (nodes, edges, or features), and (iii) an objective function that guides the explainer to identify input elements that most significantly influence the model's prediction. A variety of explanation techniques have been proposed in the literature, including perturbation-based methods, gradient-based approaches, surrogate modeling techniques, and decomposition-based methods, each offering distinct

advantages in terms of fidelity, interpretability, and computational cost. An overview of this process is illustrated in Figure 4.

In this work, we focus on edge-level explanations and utilize a post-hoc GNN explainer as base explainer to obtain importance scores for edges in each input graph. Let $\beta_{i,k}$ denote the importance score assigned by the explainer to the k-th edge of a given sample as interpreted by the i-th base learner, where $i = 1, \ldots, n$ and $k = 1, \ldots, m$ (with n being the number of base learners and m the number of edges in the sample).

To ensure consistency across base learners, we first normalize the edge scores from each base model:

$$\widetilde{\beta}_{i,k} = \frac{\beta_{i,k}}{\sum_{j=1}^{m} \beta_{i,j}} \tag{16}$$

Then, leveraging the attention weights $\alpha_i \in [0, 1]$ computed by the attention-based meta-learner, we aggregate the normalized edge scores into a unified importance score for each edge:

$$\beta_{a,k} = \sum_{i=1}^{n} \alpha_i \, \widetilde{\beta}_{i,k}, \quad \text{for } k = 1, \ldots, m \tag{17}$$

Here, $\beta_{a,k}$ represents the aggregated edge importance for the k-th edge, capturing both the local importance from base learners and their global contribution to the ensemble decision through α_i.

This aggregation-driven approach provides a unified and interpretable explanation aligned with the ensemble model's final prediction. It highlights which edges consistently contribute to base learner decisions and are emphasized by the meta-learner's attention mechanism.

3.5 Evaluation Metrics

To comprehensively assess both the classification performance and interpretability of the proposed framework, we employ the following evaluation metrics:

Accuracy

Accuracy measures the proportion of correctly classified samples over the total number of samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{18}$$

where TP, TN, FP, and FN represent the number of true positives, true negatives, false positives, and false negatives, respectively.

Precision

Precision evaluates the correctness of positive predictions, defined as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{19}$$

A high precision indicates a low false positive rate.

Recall

Recall (or sensitivity) assesses the model's ability to identify all relevant instances:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{20}$$

A high recall reflects a low false negative rate, which is particularly important in malware detection.

F1 Score

The F1 Score is the harmonic mean of precision and recall, balancing the trade-off between the two:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{21}$$

Fidelity

Fidelity quantifies the explanatory power of a subgraph by assessing its influence on the model's prediction. It measures how the prediction outcome changes when either the important or unimportant portions of the graph are removed. Fidelity is evaluated in two complementary forms:

$$Fidelity+ = 1 - \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\left(\hat{y}_i^{G_{C\setminus S}} = \hat{y}_i\right), \tag{22}$$

$$Fidelity- = 1 - \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\left(\hat{y}_i^{G_S} = \hat{y}_i\right), \tag{23}$$

where G_S denotes the important subgraph, G_C is the complete original graph, $\hat{y}_i$ is the model's predicted label for the i-th sample, and y_i is the ground truth label.

$Fidelity+$ captures the impact of the removed important subgraph by comparing the model's predictions on the original graph and on the graph with the important part removed ($G_{C\setminus S}$). A higher $Fidelity+$ indicates that the removed subgraph had a substantial influence on the model's decision, thereby validating its importance in the prediction process. Conversely, $Fidelity-$ assesses the predictive sufficiency of the important subgraph alone by comparing the model's prediction on the full graph and on G_S. A lower $Fidelity-$ implies that the identified subgraph retains most of the critical information needed for accurate prediction, thus indicating a more faithful and self-contained explanation. Together, these two metrics provide a comprehensive evaluation of explanation quality by quantifying both the necessity and sufficiency of the identified subgraph in relation to the model's output.

4 Results and Analysis

In our experiments, we utilized malicious samples from the BODMAS [33] and PMML [34] datasets, as well as benign samples obtained from the DikeDataset [35]. A summary of the datasets employed in this study is provided in Table 1. For dynamic

CFG recovery, we leveraged the Angr framework [36–38], a Python-based binary analysis tool. Angr facilitates CFG construction by combining symbolic execution with constraint solving, allowing for precise and in-depth graph generation. All experiments were conducted on a workstation equipped with an Intel Xeon Platinum 8253 CPU (32 cores, 3.0 GHz) and 128 GB RAM. We implemented our framework in Python using PyTorch Geometric for model training and NetworkX v2.8.8 for graph processing and manipulation.

Table 1: Statistics of the evaluated datasets.

Dataset	#Samples	Avg. Nodes	Avg. Edges	Label
BODMAS	122	63,226.44	66,033.72	Malware
DikeDataset	319	9,059.07	15,171.37	Benign
PMML	390	14,246.54	23,977.81	Malware

To reduce the dimensionality of the initial 439-dimensional node feature vectors, we employed a symmetrical autoencoder that projects the data into a 64-dimensional latent space. The encoder consists of three fully connected layers with dimensions 439 $\rightarrow$256$\rightarrow$128$\rightarrow$64, each followed by a ReLU activation function. The decoder replicates this architecture in reverse, using layers of size 64$\rightarrow$128$\rightarrow$256$\rightarrow$ 439, also with ReLU activations. The model was trained for 5000 epochs using the Adam optimizer with a learning rate of 0.0001, aiming to minimize the mean squared error (MSE) between the input features and their reconstructions. Training was terminated at 5000 epochs, as the validation MSE had stabilized below 1×10^{-4} for the final 1000 epochs, indicating convergence. The resulting 64-dimensional representations from the encoder were then used as input node features for subsequent graph learning tasks.

For the graph classification task, we experimented with three GNN architectures: GCN, GAT, and GIN, which served as base learners. All three models shared a common architecture comprising three graph convolutional layers with 64 hidden units each, followed by ReLU activation functions. A global mean pooling layer was applied to aggregate node-level embeddings into a graph-level representation. This representation was then passed through a dropout layer with a rate of 0.2, followed by a fully connected linear layer that produced class scores for binary classification. Each base model was trained using the Adam optimizer with a learning rate of 0.0001 and a weight decay of 0.0005. The training process used the cross-entropy loss function over 50 epochs. These hyperparameters, including learning rate, weight decay, and the number of training epochs, were selected through grid search.Figure 5 presents the class-wise validation performance metrics of the base learners across all five folds.

After performing 5-fold cross-validation and generating new training data using the base learners' predictions, all base models were retrained on the full training set to ensure consistency. For the meta-learner, we employed an attention-based MLP. The architecture consists of three fully connected layers: an input layer of size equal to the concatenated outputs of the base learners, followed by two hidden layers with 128 and 64 units respectively, each with ReLU activations and dropout (rate 0.2), and a final

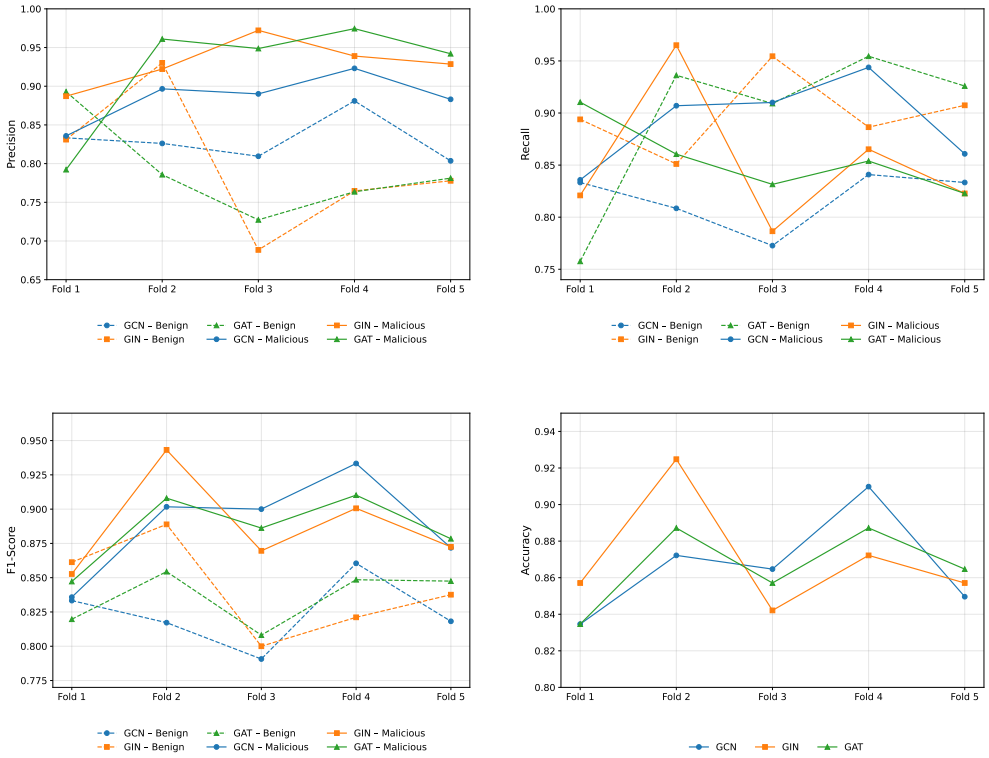


Fig. 5: Validation performance metrics (Precision, Recall, F1-Score, and Accuracy) of the base learners (GCN, GAT, and GIN) across all five folds.

output layer for binary classification. The meta-learner was trained for 100 epochs using the Adam optimizer with a learning rate of 0.001.

Tables 2,3,4, and 5 present the class-wise evaluation metrics, including precision, recall, and F1-score for both benign and malicious classes, along with the overall accuracy of each model. The results cover the performance of the base learners (GCN, GIN, and GAT) as well as the SE model on the test dataset. Furthermore, Figure 6 provides a visual representation of these metrics using a radar plot.

Across all evaluation criteria, including precision, recall, F1-score, and accuracy, the SE model consistently achieves the highest performance. This demonstrates that combining multiple base learners through ensemble learning enhances both generalization and robustness.

The SE model achieves the highest overall accuracy of 86.14%, along with the best macro and weighted averages across all metrics. For the malicious class, which is of primary concern in malware detection, the SE model obtains a recall of 91.18% and an F1-score of 89%, outperforming each individual base learner. The high recall is particularly critical in cybersecurity applications, where false negatives, meaning undetected malware instances, are significantly more damaging than false positives.

Although the GIN model exhibits the highest precision for the malicious class (91.95%), its recall is the lowest (78.43%) among all models. This indicates that while GIN is effective in avoiding false positives, it is more prone to missing actual malicious instances. In malware detection, this trade-off is suboptimal, as failing to identify threats can lead to serious security breaches. Therefore, despite its strong precision, GIN's low recall diminishes its practical reliability as a standalone model.

In contrast, the SE model provides a balanced performance across both classes, achieving the highest F1-scores for benign and malicious samples. These findings underscore the advantage of ensemble strategies that integrate the strengths of diverse GNN architectures, resulting in a more effective and dependable detection framework.

Furthermore, Figure 7 presents the ROC curves for the base learners and the SE model. The Area Under the Curve (AUC) for the SE model reaches 0.9390, which is higher than that of all individual base models, further confirming the superior discriminative capability of the ensemble approach.

Table 2: Test performance of GCN model.

Metric	Precision	Recall	F1-Score	Support
Benign	0.7903	0.7656	0.7778	64
Malicious	0.8558	0.8725	0.8641	102
Accuracy			0.8313	166
Macro Avg	0.8230	0.8191	0.8209	166
Weighted Avg	0.8305	0.8313	0.8308	166

Table 3: Test performance of GIN model.

Metric	Precision	Recall	F1-Score	Support
Benign	0.7215	0.8906	0.7972	64
Malicious	0.9195	0.7843	0.8466	102
Accuracy			0.8253	166
Macro Avg	0.8205	0.8375	0.8219	166
Weighted Avg	0.8432	0.8253	0.8275	166

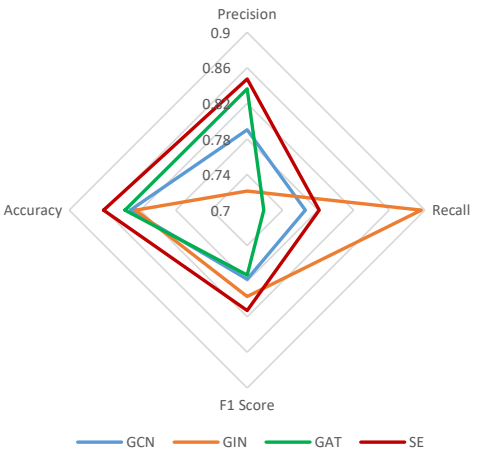
To evaluate explanation Fidelity, we employed Integrated Gradients (IG) and Guided Backpropagation (GBP) as representative post-hoc gradient-based explainers, both of which are widely regarded as state-of-the-art techniques for interpreting GNN predictions. For each base learner (GCN, GIN, and GAT), both IG and GBP were applied independently to generate explanations. In the case of the SE model, IG and GBP were also used as base explainers, and their outputs were further processed using the aggregation-based explanation method introduced in the previous section. Figure 8 presents the Fidelity results for IG and GBP across the GCN, GIN, GAT,

Table 4: Test performance of GAT model.

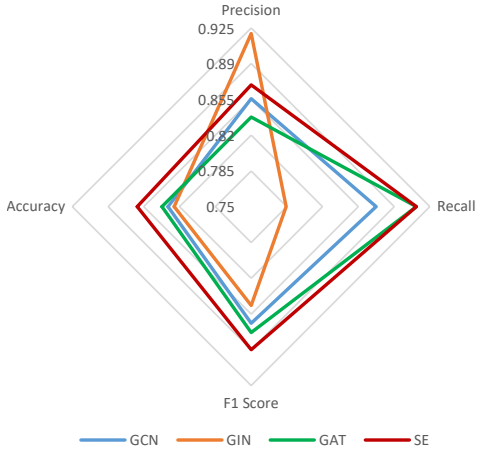
Metric	Precision	Recall	F1-Score	Support
Benign	0.8364	0.7188	0.7731	64
Malicious	0.8378	0.9118	0.8732	102
Accuracy			0.8373	166
Macro Avg	0.8371	0.8153	0.8232	166
Weighted Avg	0.8373	0.8373	0.8346	166

Table 5: Test performance of SE model.

Metric	Precision	Recall	F1-Score	Support
Benign	0.8475	0.7812	0.8130	64
Malicious	0.8692	0.9118	0.8900	102
Accuracy			0.8614	166
Macro Avg	0.8583	0.8465	0.8515	166
Weighted Avg	0.8608	0.8614	0.8603	166



(a) (a) Benign class performance across models.



(b) (b) Malicious class performance across models.

Fig. 6: Class-wise test performance of all models.

and SE models. The left subplot illustrates $Fidelity+$, while the right subplot depicts $Fidelity-$.

The results demonstrate that, for both IG and GBP, our aggregation-based explanation method applied to the SE model consistently achieves lower $Fidelity-$ values across different sparsity levels, indicating a stronger ability to identify and prioritize influential subgraphs. Furthermore, in terms of $Fidelity+$, the proposed explanation

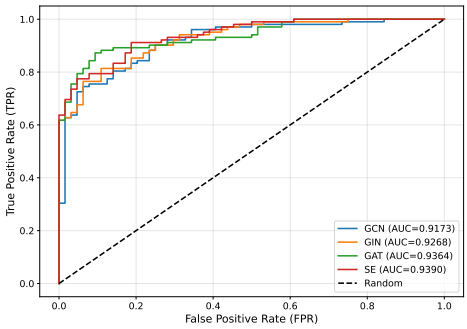


Fig. 7: Comparison of ROC curves for base learners and SE model.

approach yields values that are comparable to, or slightly higher than, those obtained from the base learners. This suggests that the aggregation method effectively preserves critical information while improving interpretability. Overall, these findings confirm the utility and robustness of the proposed explanation framework in the context of ensemble-based GNN models.

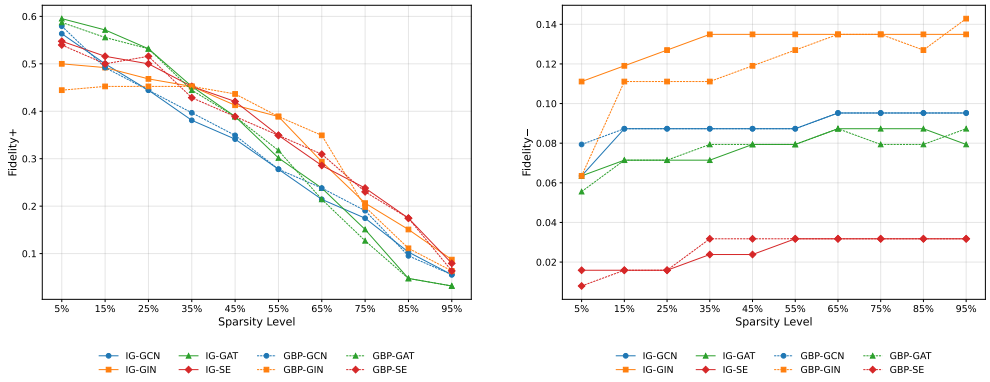


Fig. 8: Fidelity evaluation of IG and GBP explainers across GCN, GIN, GAT, and SE models. For the SE model, IG and GBP explanations are further refined using the proposed aggregation-based method.

5 Conclusion

This paper presented a novel SE framework for explainable malware detection using GNNs on dynamic CFGs. The proposed approach integrates multiple diverse GNN base learners with an attention-based meta-learner to enhance classification performance and provide interpretable predictions. A two-step node embedding strategy was employed to encode semantic and structural information from assembly instructions,

while the meta-learner not only aggregated predictions but also offered insights into the relative contributions of each base model.

To address the challenge of interpretability in ensemble-based GNN models, we introduced a post-hoc explanation technique that fuses edge importance scores from gradient-based explainers using the attention weights of the meta-learner. This aggregation-driven approach generated ensemble-aware explanations aligned with the model's final decision, improving the Fidelity.

Extensive experiments on real-world malware and benign datasets demonstrated that the proposed SE framework outperforms individual GNN models in terms of accuracy, F1-score, and AUC, particularly for the critical malicious class. Moreover, the explanation results revealed the capability of our method to identify influential subgraphs.

Future work may explore the extension of this framework to multi-class malware classification, integration of dynamic features beyond CFGs, and further investigation into explainability techniques tailored for ensemble architectures in graph-based learning contexts.

References

[1] Shokouhinejad, H., Razavi-Far, R., Mohammadian, H., Rabbani, M., Ansong, S., Higgins, G., Ghorbani, A.A.: Recent advances in malware detection: Graph learning and explainability. arXiv preprint arXiv:2502.10556 (2025)

[2] Shokouhinejad, H., Higgins, G., Razavi-Far, R., Mohammadian, H., Ghorbani, A.A.: On the consistency of gnn explanations for malware detection. arXiv, priprint: 2504.16316 (2025) 2504.16316 [cs.CR]

[3] Peng, H., Yang, J., Zhao, D., Xu, X., Pu, Y., Han, J., Yang, X., Zhong, M., Ji, S.: 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 **19**, 4881–4896 (2024)

[4] Li, W., Tang, H., Zhu, H., Zhang, W., Liu, C.: Ts-mal: Malware detection model using temporal and structural features learning. Computers & Security **140**, 103752 (2024)

[5] Feng, P., Gai, L., Yang, L., Wang, Q., Li, T., Xi, N., Ma, J.: Dawngnn: Documentation augmented windows malware detection using graph neural network. Computers & Security **140**, 103788 (2024)

[6] Yu, Z., Li, S., Bai, Y., Han, W., Wu, X., Tian, Z.: Remsf: A robust ensemble model of malware detection based on semantic feature fusion. IEEE Internet of Things Journal **10**(18), 16134–16143 (2023)

[7] Chen, Y.-H., Lin, S.-C., Huang, S.-C., Lei, C.-L., Huang, C.-Y.: Guided malware sample analysis based on graph neural networks. IEEE Transactions on

Information Forensics and Security **18**, 4128–4143 (2023)

[8] Shokouhinejad, H., Razavi-Far, R., Higgins, G., Ghorbani, A.A.: Node-Centric Pruning: A novel graph reduction approach. Machine Learning and Knowledge Extraction **6**(4), 2722–2737 (2024)

[9] Liu, Z., Wang, R., Japkowicz, N., Gomes, H.M., Peng, B., Zhang, W.: Segdroid: An android malware detection method based on sensitive function call graph learning. Expert Systems with Applications **235**, 121125 (2024)

[10] Zhang, X., Zhang, M., Zhang, Y., Zhong, M., Zhang, X., Cao, Y., Yang, M.: Slowing down the aging of learning-based malware detectors with api knowledge. IEEE Transactions on Dependable and Secure Computing **20**(2), 902–916 (2023)

[11] Bu, S.-J., Cho, S.-B.: Triplet-trained graph transformer with control flow graph for few-shot malware classification. Information Sciences **649**, 119598 (2023)

[12] Ben Yahia, N.: Enhancing social and collaborative learning using a stacked gnn-based community detection. Social Network Analysis and Mining **14**(1), 205 (2024)

[13] Venkatapathy, S., Votinov, M., Wagels, L., Kim, S., Lee, M., Habel, U., Ra, I.-H., Jo, H.-G.: Ensemble graph neural network model for classification of major depressive disorder using whole-brain functional connectivity. Frontiers in Psychiatry **14**, 1125339 (2023)

[14] Kim, S., Lee, D., Kang, S., Lee, S., Yu, H.: Learning topology-specific experts for molecular property prediction. In: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence. AAAI Press, ??? (2023)

[15] Hu, F., Wang, L., Liu, Q., Wu, S., Wang, L., Tan, T.: Graphdive: Graph classification by mixture of diverse experts. In: Raedt, L.D. (ed.) Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22, pp. 2080–2086. International Joint Conferences on Artificial Intelligence Organization, ??? (2022)

[16] Wang, H., Jiang, Z., You, Y., Han, Y., Liu, G., Srinivasa, J., Kompella, R.R., Wang, Z.: Graph mixture of experts: learning on large-scale graphs with explicit diversity modeling. In: Proceedings of the 37th International Conference on Neural Information Processing Systems (2023)

[17] Zeng, Y., Wei, Y., Yang, Y., Xu, S., Zhang, H., Jie, Y.: A novel real-time classification method of mixed weathered mudstone-sand-pebble formation ahead of an epbs using dbokm-smote and stacking ensemble learning. Tunnelling and Underground Space Technology **165**, 106870 (2025)

[18] Li, T., Yao, Q., Li, J., Li, Q., Wang, R., Jia, C., Xiao, Y.: Malware detection model based on stacking ensemble technique. Applied Soft Computing **180**, 113338 (2025)

[19] Naeem, H., Dong, S., Falana, O.J., Ullah, F.: Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification. Expert Systems with Applications **223**, 119952 (2023)

[20] Zhu, H., Li, Y., Li, R., Li, J., You, Z., Song, H.: Sedmdroid: An enhanced stacking ensemble framework for android malware detection. IEEE Transactions on Network Science and Engineering **8**(2), 984–994 (2021)

[21] Vasan, D., Alazab, M., Venkatraman, S., Akram, J., Qin, Z.: Mthael: Cross-architecture iot malware detection based on neural network advanced ensemble learning. IEEE Transactions on Computers **69**(11), 1654–1667 (2020)

[22] Joshi, A., Kumar, S.: Stacking-based ensemble model for malware detection in android devices. International Journal of Information Technology **15**(6), 2907–2915 (2023)

[23] Shokouhinejad, H., Razavi-Far, R., Higgins, G., Ghorbani, A.A.: Dual explanations via subgraph matching for malware detection. arXiv, preprint: 2504.20904 (2025) 2504.20904 [cs.CR]

[24] Mohammadian, H., Higgins, G., Ansong, S., Razavi-Far, R., Ghorbani, A.A.: Explainable malware detection through integrated graph reduction and learning techniques. arXiv preprint arXiv:2412.03634 (2024)

[25] Ying, Z., Bourgeois, D., You, J., Zitnik, M., Leskovec, J.: Gnnexplainer: Generating explanations for graph neural networks. Advances in Neural Information Processing Systems (NIPS) **32** (2019)

[26] Luo, D., Cheng, W., Xu, D., Yu, W., Zong, B., Chen, H., Zhang, X.: Parameterized explainer for graph neural network. In: 34th International Conference on Neural Information Processing Systems (2020)

[27] Yuan, H., Yu, H., Wang, J., Li, K., Ji, S.: On explainability of graph neural networks via subgraph explorations. In: International Conference on Machine Learning, pp. 12241–12252 (2021). PMLR

[28] Baldassarre, F., Azizpour, H.: Explainability techniques for graph convolutional networks. In: International Conference on Machine Learning (ICML), Workshop on Learning and Reasoning with Graph-Structured Representations (2019)

[29] Herath, J.D., Wakodikar, P.P., Yang, P., Yan, G.: 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)

[30] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017)

[31] Keyulu Xu, J.L. Weihua Hu, Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (ICLR) (2019)

[32] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph attention networks. In: International Conference on Learning Representations (ICLR) (2018)

[33] Yang, L., Ciptadi, A., Laziuk, I., Ahmadzadeh, A., Wang, G.: Bodmas: An open dataset for learning based temporal analysis of pe malware. In: 2021 IEEE Security and Privacy Workshops (SPW), pp. 78–84 (2021). IEEE

[34] Practical Security Analytics LLC: PE Malware Machine Learning Dataset. https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/. Accessed: 2024-08-06 (2024)

[35] Iosif, G.-A.: DikeDataset. https://github.com/iosifache/DikeDataset. Accessed on February 27, 2024 (2021)

[36] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis (2016)

[37] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution (2016)

[38] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware (2015)