Breaking Obfuscation: Cluster-Aware Graph with LLM-Aided Recovery for Malicious JavaScript Detection⋆

Zhihong Liang[a,b], Xin Wang[c], Zhenhuang Hu[c], Liangliang Song[c], Lin Chen[a,b], Jingjing Guo[c], Yanbin Wang[c,*] and Ye Tian[c,*]

[a]Electric Power Research Institute, CSG, Guangzhou, Guangdong, China
[b]Guangdong Provincial Key Laboratory of Power System Network Security, Guangzhou, Guangdong, China
[c]Hangzhou Institute of Technology, Xidian University, Hangzhou, China

ARTICLE INFO

Keywords:
Malicious Code Detection
JavaScript
Graph Neural Networks
Deobfuscation
Large Language Models

ABSTRACT

With the rapid expansion of web-based applications and cloud services, malicious JavaScript code continues to pose significant threats to user privacy, system integrity, and enterprise security. However, detecting such threats remains challenging due to sophisticated code obfuscation techniques and JavaScript's inherent language characteristics, particularly its nested closure structures and syntactic flexibility. In this work, we propose DeCoda, a hybrid defense framework that combines large language model (LLM)-based deobfuscation with code graph learning: (1) We first construct a sophisticated prompt-learning pipeline with multi-stage refinement, where the LLM progressively reconstructs the original code structure from obfuscated inputs and then generates normalized Abstract Syntax Tree (AST) representations; (2) In JavaScript ASTs, dynamic typing scatters semantically similar nodes while deeply nested functions fracture scope capturing, introducing structural noise and semantic ambiguity. To address these challenges, we then propose to learn hierarchical code graph representations via a Cluster-wise Graph that synergistically integrates graph transformer network, node clustering, and node-to-cluster attention to simultaneously capture both local node-level semantics and global cluster-induced structural relationships from AST graph. Experimental results demonstrate that our method achieves F1-scores of 94.64% and 97.71% on two benchmark datasets, demonstrating absolute improvements of 10.74% and 13.85% over state-of-the-art baselines. In false-positive control evaluation at fixed FPR levels (0.0001, 0.001, 0.01), our approach delivers 4.82×, 5.91×, and 2.53× higher TPR respectively compared to the best-performing baseline. These results highlight the effectiveness of LLM-based deobfuscation and underscore the importance of modeling cluster-level relationships in detecting malicious code. Our code is available at the following link: https://github.com/zer0p0intvvv/DeCoda.

1. Introduction

JavaScript Wirfs-Brock and Eich (2020) is a foundational technology for modern web development, powering dynamic and interactive web applications. However, its widespread adoption and dynamic nature also make it a frequent target for malicious exploitation. Attackers inject obfuscated scripts into vulnerable websites and web servicesTian et al. (2025); Li et al. (2016); Malik et al. (2019); Lee and Son (2023); Liu et al. (2025a, 2024); Wang et al. (2023); Song et al. (2025); Wang et al. (2022); Liu et al. (2025a), aiming to steal sensitive data, hijack user sessions, or deploy further payloads. These threats extend beyond traditional web environments, impacting emerging domains Zhang et al. (2024) such as IoT devices, cloud platforms, and even autonomous systems where JavaScript-based interfaces are increasingly utilized.

To evade detection, these scripts are often obfuscated using techniques Schrittwieser et al. (2016); Behera and Bhaskari (2015) such as variable renaming, control flow distortion, string encoding, and dynamic function calls, which obscure both their syntactic form and semantic intent Skolka et al. (2019); Wei et al. (2025). While traditional detection approaches such as signature-based scanning and sequence-based machine learning models (e.g., BERT Koroteev (2021), LSTM Greff et al. (2016), and CNN Chua and Roska (2002)) Sun et al. (2021); Zhang et al. (2024) have proven effective against simpler threats, their reliance on linear or localized code representations fundamentally limits their ability to model the hierarchical and relational dependencies essential for analyzing obfuscated code.

In contrast, Graph Neural Networks (GNNs) Corso et al. (2024); Xiao et al. (2025) provide a distinct paradigm by representing source code as structured graphs, such as abstract syntax trees (ASTs) Yamaguchi et al. (2014). These representations inherently capture syntactic and structural relationships, enabling models to encode dependencies effectively. However, existing methods fail to account for JavaScript-specific language characteristics, such as syntactic flexibility and deep closure nesting structures.

Overall, malicious JavaScript detection faces two critical challenges: (1) The language's flexibility facilitates diverse obfuscation techniques to evade pattern-matching detectors, while current deobfuscation approaches relying on static rule-based transformations cannot handle JavaScript's polymorphic code variations. (2) Prior GNN implementations overlook fundamental JavaScript attributes: (a) they ignore

⋆This document is the result of a research project supported by the Guangdong Provincial Key Laboratory of Power System Network Security.
*Corresponding author: Yanbin Wang (wangyanbin15@mails.ucas.ac.cn), Ye Tian (tianye@xidian.edu.cn)
ORCID(s): 0009-0004-4555-3773 (L. Song); 0000-0003-1682-5712 (Y. Wang); 0000-0003-0608-8544 (Y. Tian)

AST node clustering properties - where JavaScript's syntactic flexibility causes semantically equivalent nodes to scatter in feature space, inducing semantic interpretation errors, and (b) their message passing mechanisms disrupt critical scope-chain dependencies, while deeper GNN architectures exacerbate the oversmoothing of closure hierarchies, progressively losing granular scope information across network layers.

To address these challenges, we propose a novel hybrid framework for malicious JavaScript detection that synergizes LLM-guided deobfuscation with cluster-aware graph learning. Our primary innovation leverages LLM's semantic reconstruction capability to restore syntactic clarity and structural coherence through a multi-stage refinement pipeline, where the language model progressively recovers original code semantics from obfuscated inputs and generates normalized AST representations. By organizing these ASTs into hierarchical graph structures, we deploy a graph transformer with node-to-cluster attention that simultaneously captures: (a) semantically consistent node-level features, and (b) cluster-induced structural patterns, effectively modeling both lexical syntax and program-wide dependency relationships unique to JavaScript's execution context.

Our key contributions include:

- Our approach achieves 94.64% and 97.71% F1-scores on the benchmark datasets, with absolute performance gains of 10.74% and 13.85% over existing methods. For security-critical low-FPR scenarios (0.0001, 0.001, 0.01), the method demonstrates substantial TPR improvements of 4.82×, 5.91×, and 2.53× relative to the strongest baseline.

- Our multi-stage LLM deobfuscation pipeline, guided by structured prompt engineering (including string decoding, semantic variable renaming, dynamic invocation reconstruction, and control flow simplification), empirically demonstrates significant deobfuscation improvements. The systematic prompt design ensures: (1) complete payload unpacking, (2) behavioral equivalence preservation, and (3) explanatory metadata generation.

- We uses a robust dual-scale graph learning framework for JavaScript ASTs that simultaneously models node-level features and cluster-induced structural patterns through node-to-cluster attention, effectively addressing: (1) semantic equivalence dispersion in feature space, and (2) scope chain dependency breakdowns in deep closure nesting.

2. Related Work

As JavaScript dominates web development, malicious script detection remains an essential security challenge. This work provides a methodological taxonomy of learning-based detectors, analyzing representative approaches across different modeling paradigms while positioning our framework's structural innovations in context.

2.1. Sequence Model

Sequence-based models employ dominant sequential learning algorithms (LSTMs, CNNs, and Transformers) to extract patterns from code sequences. While all process code as linear sequences, their inductive biases lead to fundamentally different feature representations.

Recurrent Neural Networks, particularly LSTM Zaremba et al. (2014), have been adopted for modeling sequential dependencies in malicious JavaScript detection. Fang et al. Fang et al. (2018) used static analysis by learning opcode sequences from compiled JavaScript bytecode using LSTM networks. Subsequent enhancements by Song et al. Song et al. (2020) integrated bidirectional LSTMs with program dependence graphs to analyze semantically sliced execution paths, improving resilience against basic obfuscation. Further improvements incorporated attention mechanisms and semantic embeddings Fang et al. (2020) to identify critical code segments in tokenized JavaScript. However, these approaches fundamentally suffer from three limitations: (1) linear processing constraints that prevent hierarchical relationship modeling, (2) limited context windows for long-range dependency analysis, and (3) inability to effectively represent non-sequential program structures.

CNNs have been extensively applied to malicious JavaScript detection through various code representations, including syntax trees, bytecode, and token sequences. The JSAC framework Liang et al. (2019) employs parallel CNNs to process both abstract syntax trees (ASTs) and control flow graphs (CFGs), capturing complementary syntactic and semantic features. Rozi et al. Rozi et al. (2020) advanced this paradigm by introducing a deep pyramid CNN architecture operating on V8 engine bytecode sequences, augmented with recurrent layers for improved obfuscation resilience. Alternative implementations include Sheneamer's stacked CNN ensemble Sheneamer (2024) for vulnerability detection and ScriptNet's hierarchical CNN Stokes et al. (2019) for byte-level sequence analysis. While CNNs demonstrate exceptional proficiency in extracting local code patterns and achieving strong static analysis performance, their fundamental architectural constraints—particularly limited receptive fields and absence of explicit structural modeling—severely impair their capability to analyze long-range dependencies or complex program semantics.

Transformer architectures, particularly BERT-based models, have demonstrated promising results in malicious JavaScript detection through contextual token embeddings. While hybrid approaches like the BERT-BiLSTM model Abadeer et al. (2022) improve semantic understanding, they fundamentally lack mechanisms to capture the localized syntactic patterns and hierarchical structural relationships critical for effective malware detection beyond pure semantic analysis.

2.2. Code Graph learning

Source code's intrinsic hierarchical graph structure - encompassing control flows, data dependencies, and lexical scopes - renders GNNs particularly suitable for learning

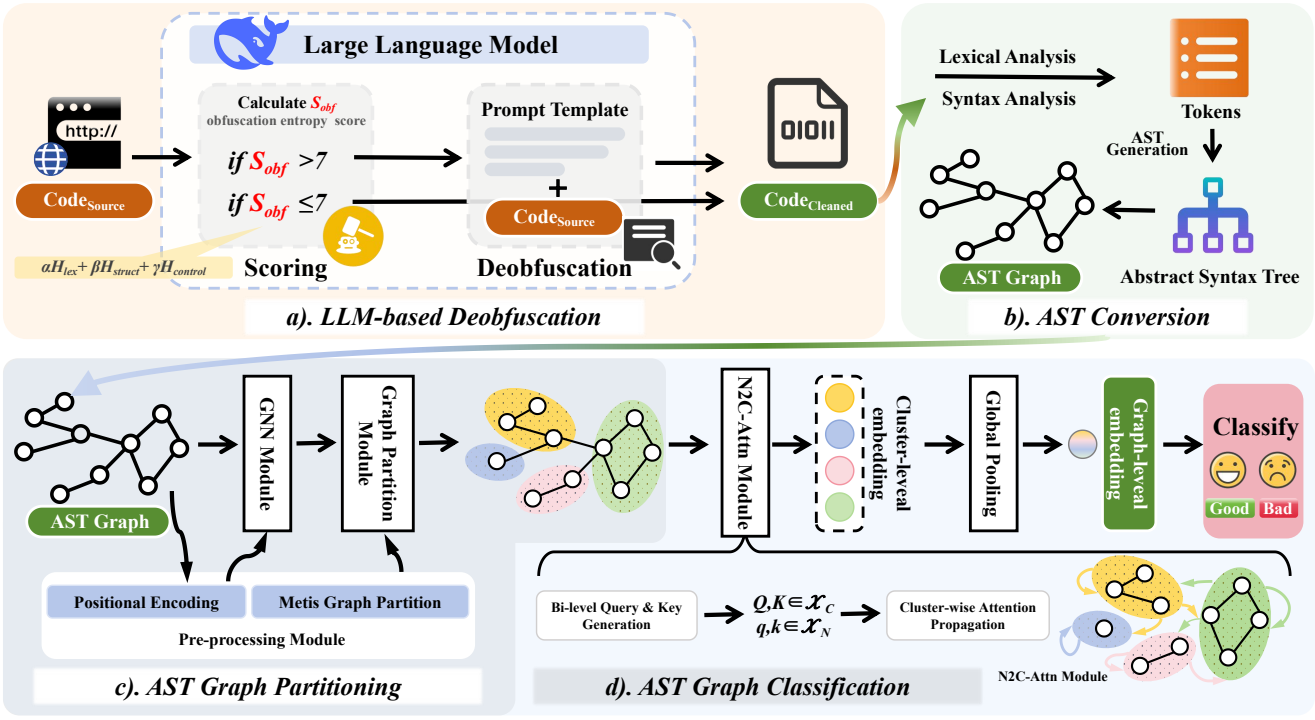


Figure 1: Architecture of the our method.

code garph. As demonstrated by Sheng et al. (2025); Liu et al. (2025b); Sun et al. (2025), GNNs excel at modeling complex structural patterns in graph data. JStrong Fang et al. (2022) apply graph convolutional networks to learn joint structural-semantic features, and JStrack Rozi et al. (2021) employs hierarchical GNNs to preserve syntactic information while analyzing nested code structures.

While GNN applications in code analysis are well-established, their adoption for malicious JavaScript detection remains limited. Existing approaches often employ generic graph architectures (e.g., flat message passing or coarse pooling) that neglect JavaScript-specific characteristics, potentially losing critical syntactic details or facing scalability issues. Our method addresses these limitations through: (1) AST partitioning into meaningful substructures, and (2) node-to-cluster attention for joint hierarchical reasoning, enabling multi-granularity feature integration while preserving structural fidelity. This design demonstrates superior resilience and practical scalability compared to conventional GNN implementations.

3. Method

Our proposed method for detecting malicious obfuscated JavaScript code integrates LLM-based deobfuscation with advanced GNN classification. The overall framework is designed to be modular and extensible, allowing it to adapt effectively to diverse obfuscation patterns and attack scenarios. LLMs are leveraged to restore the syntactic and semantic clarity of obfuscated code, after which structural representations such as ASTs are extracted and transformed into graph form. A graph-based classifier, built upon a cluster-aware transformer architecture, is subsequently applied to capture the structural relationships among nodes and clusters, enabling robust and accurate malicious code detection. An overview of the entire detection pipeline is illustrated in Fig. 1.

3.1. LLM-based Deobfuscation

We utilize DeepSeek-R1, a cost-efficient state-of-the-art large language model, for automated JavaScript deobfuscation. The process incorporates three key components:

3.1.1. Semantic Representation and Feature Extraction

The LLM maps obfuscated code X_{obf} to a semantic space through its pre-trained Transformer architecture:

$$h = \text{Transformer}(X_{\text{obf}}; \theta) \tag{1}$$

where h represents the semantic representation vector, and θ denotes the model parameters. DeepSeek-R1Guo et al. (2025)'s Multi-head Latent Attention (MLA) mechanism extracts key features:

$$c_{KV_t} = W_{D_{KV}} h_t, \quad k_t^C = W_{U_K} c_{KV_t}, \quad v_t^C = W_{U_V} c_{KV_t} \tag{2}$$

This allows the model to capture deep semantic relationships within obfuscated code patterns.

3.1.2. Obfuscation Entropy Scoring

Before applying deobfuscation, we evaluate the complexity of obfuscation using an entropy-based scoring mechanism. The obfuscation entropy score S_{obf} is calculated as:

$$S_{\text{obf}} = \alpha H_{\text{lex}} + \beta H_{\text{struct}} + \gamma H_{\text{control}} \tag{3}$$

where:

- $H_{\text{lex}} = -\sum_{i=1}^{|V|} p_i \log p_i$ represents lexical entropy based on token distribution
- $H_{\text{struct}} = \frac{1}{|N|} \sum_{n \in \text{AST}} \log(\text{depth}(n) \cdot \text{children}(n))$ measures structural complexity
- $H_{\text{control}} = \frac{|\text{branches}|+|\text{loops}|}{|\text{statements}|}$ quantifies control flow complexity
- $\alpha = 0.4$, $\beta = 0.4$, $\gamma = 0.2$ are weighting parameters

Only code samples with $S_{\text{obf}} > 7$ undergo deobfuscation, optimizing computational resources while ensuring that genuinely obfuscated code is processed.

3.1.3. Prompt Engineering for Deobfuscation

We design a specialized prompt template that guides the LLM to systematically deobfuscate JavaScript code:

JavaScript Deobfuscation Prompt Template

Task Description:
Analyze the following obfuscated JavaScript code and provide a clean, readable version while preserving exact functionality.

Instructions:

1. Decode all string obfuscations (hex, base64, unicode escapes)
2. Replace meaningless variable names with descriptive ones
3. Unpack compressed/encoded payloads (e.g., eval expressions)
4. Simplify control flow (remove dead code, flatten conditionals)
5. Reconstruct function calls from dynamic invocations
6. Preserve original logic and behavior exactly

Input: Obfuscated Code

`{obfuscated_code}`

Expected Output:
Provide the deobfuscated version with explanatory comments that describe:

- The original obfuscation techniques detected
- The deobfuscation steps applied
- The restored functionality and control flow

3.1.4. Probabilistic Generation and Optimization

The deobfuscation process is modeled as a sequence-to-sequence transformation, where the model predicts the probability distribution of clean code Y_{clean}:

$$P(Y_{\text{clean}}|X_{\text{obf}}) = \prod_{t=1}^{T} P(y_t|y_{<t}, X_{\text{obf}}; \theta) \tag{4}$$

DeepSeek-R1's Mixture of Experts (MoE) architecture enhances generation capability:

$$h'_t = u_t + \sum_{i=1}^{N_s} \text{FFN}_i^{(s)}(u_t) + \sum_{i=1}^{N_r} g_{i,t} \text{FFN}_i^{(r)}(u_t) \tag{5}$$

where $g_{i,t}$ is the gating function selecting appropriate experts for deobfuscation patterns.

3.1.5. Training and Loss Function

The model is optimized using a deobfuscation-specific loss function:

$$L_{\text{deobf}} = -\sum_{i=1}^{N} \log P(Y_{\text{clean},i}|X_{\text{obf},i}; \theta) + \lambda L_{\text{semantic}} \tag{6}$$

where L_{semantic} ensures semantic preservation between obfuscated and deobfuscated versions.

3.1.6. Code Structure Restoration

The model restores obfuscated structures through syntax tree parsing and semantic inference:

$$\text{AST}_{\text{clean}} = f_{\text{decode}}(h; \theta_{\text{dec}}) \tag{7}$$

where $\text{AST}_{\text{clean}}$ is the restored abstract syntax tree, and f_{decode} is the decoding function leveraging the model's generation capabilities.

The deobfuscation pipeline is automated via scripts that interact with the DeepSeek-R1 API, supporting batch processing with configurable parameters (temperature=0.1, max_tokens=4096) to ensure deterministic and complete deobfuscation. Quality filtering ensures that only successfully deobfuscated samples with semantic equivalence are retained for downstream analysis.

3.2. AST Conversion

The deobfuscated code is parsed into ASTs using the Esprima parser for JavaScript. ASTs serve as a natural bridge between source code and graph neural networks by encoding the syntactic structure of programs in a hierarchical graph format.

Given deobfuscated code Y_{clean}, we construct an AST $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ represents AST nodes (e.g., FunctionDeclaration, VariableDeclarator) and $\mathcal{E}$ represents parent-child relationships. The AST is then transformed into a graph representation:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}_{\text{ext}}, \mathbf{X}, \mathbf{A}) \tag{8}$$

where $\mathcal{E}_{\text{ext}} = \mathcal{E} \cup \mathcal{E}_{\text{data}} \cup \mathcal{E}_{\text{control}}$ includes extended edges for data flow and control flow.

Node features are encoded through multi-component embedding:

$$\mathbf{x}_i = \mathbf{e}_{\text{type}}(t_i) \oplus \mathbf{e}_{\text{value}}(s_i) \oplus \mathbf{e}_{\text{pos}}(l_i, c_i) \tag{9}$$

where $\mathbf{e}_{\text{type}}$, $\mathbf{e}_{\text{value}}$, and $\mathbf{e}_{\text{pos}}$ embed node types, values, and positions respectively.

Finally, the AST graph is converted to PyG format:

$$\text{Data} = \{\mathbf{x}, \text{edge\_index}, \text{edge\_attr}, y\} \tag{10}$$

This representation preserves the rich structural information inherent in code while enabling direct application of GNN architectures.

3.3. AST Graph Partitioning

To handle large-scale AST graphs and enhance model expressiveness, we apply graph partitioning techniques to divide each AST into multiple patches or subgraphs. This partitioning enables multi-granularity analysis while maintaining computational efficiency.

We employ the METIS algorithm to partition the AST graph into m clusters. The cluster assignment is defined by matrix $\mathbf{C} \in \mathbb{R}^{n \times m}$:

$$\mathbf{C}_{nm}^{\text{Metis}} = \begin{cases} \frac{1}{|\mathcal{V}_m|} & \text{if the } n\text{-th node is in the } m\text{-th cluster} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where $|\mathcal{V}_m|$ denotes the number of nodes in cluster m, ensuring balanced representation.

The graph coarsening process transforms the original node features and adjacency matrix to cluster-level representations:

$$\mathbf{X}^P = \mathbf{C}^T \mathbf{X}, \quad \mathbf{A}^P = \mathbf{C}^T \mathbf{A} \mathbf{C} \tag{12}$$

where $\mathbf{X}^P \in \mathbb{R}^{m \times d}$ is the cluster-level feature matrix and $\mathbf{A}^P \in \mathbb{R}^{m \times m}$ is the cluster-level adjacency matrix.

For our Cluster-GT architecture, we compute cluster-level queries and keys using the assignment matrix:

$$K_j = \mathbf{W}'_k \left(\sum_s \mathbf{C}_{sj} h_s \right), \quad Q_i = \mathbf{W}'_v \left(\sum_s \mathbf{C}_{si} h_s \right) \tag{13}$$

This formulation enables the Node-to-Cluster Attention mechanism to integrate both node-level and cluster-level information, capturing hierarchical patterns essential for detecting malicious code structures. The partitioned graphs retain both local syntactic details and global structural patterns, supporting robust classification even in the presence of obfuscation.

3.4. Cluster-wise Code Graph Transformer

In JavaScript ASTs, dynamic typing scatters semantically similar nodes while deeply nested functions fracture scope capturing, introducing structural noise and semantic ambiguity.To address these challenges, we propose a cluster-based AST graph learning framework employing Cluster-wise Graph Transformer Huang et al. (2024), that employs node-to-cluster attention(N2C-Attn) and cluster-guided message passing to capture consistent linguistic patterns across node types and bridge semantic relationships fractured by scope boundaries.

3.4.1. Node-to-Cluster Attention

The general form of the N2C-Attn is defined as:

$$\text{N2C-Attn}(X)_i = \frac{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj}\, \kappa_{\text{B}}(\{Q_i, q_i\}, \{K_j, k_t\}) v_t}{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj}\, \kappa_{\text{B}}(\{Q_i, q_i\}, \{K_j, k_t\})} \tag{14}$$

where $\mathbf{A}^P$ is the cluster-level adjacency matrix, $\mathbf{C}$ is the cluster assignment matrix, κ_{B} is the dual-granularity kernel function, and Q_i, q_i, K_j, k_t are the cluster-level and node-level queries and keys, respectively.

For the tensor product kernel N2C-Attn-T:

$$\kappa_{\text{B}}(\{Q_i, q_i\}, \{K_j, k_t\}) = \kappa_C(Q_i, K_j)\, \kappa_N(q_i, k_t) \tag{15}$$

$$\text{N2C-Attn-T}(X)_i = \frac{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj}\, \kappa_C(Q_i, K_j)\, \kappa_N(q_i, k_t) v_t}{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj}\, \kappa_C(Q_i, K_j)\, \kappa_N(q_i, k_t)} \tag{16}$$

For the convex linear combination kernel (N2C-Attn-L):

$$\kappa_{\text{B}}(\{Q_i, q_i\}, \{K_j, k_t\}) = \alpha\, \kappa_C(Q_i, K_j) + \beta\, \kappa_N(q_i, k_t) \tag{17}$$

$$\text{N2C-Attn-L}(X)_i = \frac{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj} (\alpha\, \kappa_C(Q_i, K_j) + \beta\, \kappa_N(q_i, k_t)) v_t}{\sum_j \mathbf{A}^P_{i,j} \sum_t \mathbf{C}_{tj} (\alpha\, \kappa_C(Q_i, K_j) + \beta\, \kappa_N(q_i, k_t))} \tag{18}$$

where $\alpha, \beta \geq 0$ and $\alpha + \beta = 1$ are learnable parameters.

3.4.2. Dual-granularity Kernelized Attention Framework

The general form of the dual-granularity kernel function is:

$$\kappa_{\text{B}}(\{\mathbf{x}^m\}_{m=1}^M, \{\mathbf{y}^m\}_{m=1}^M) = f_\eta(\{\kappa_m(\mathbf{x}^m, \mathbf{y}^m)\}_{m=1}^M) \tag{19}$$

where $M = 2$ in N2C-Attn, corresponding to node-level and cluster-level kernels.

3.4.3. AST Graph-level Embedding

The final graph-level embedding is obtained by average pooling over the cluster outputs of the N2C-Attn module:

$$\mathbf{h}_{\text{graph}} = \frac{1}{|\mathcal{N}^P|} \sum_{i \in \mathcal{N}^P} \text{N2C-Attn}(X)_i \tag{20}$$

where $\mathcal{N}^P$ denotes the set of clusters and $\text{N2C-Attn}(X)_i$ is the attention output for the i-th cluster.

3.4.4. Multi-Layer Perceptron for Classification

A Multi-Layer PerceptronRiedmiller and Lernen (2014) is a feedforward neural network used for classification tasks, consisting of an input layer with m neurons (features), $L-1$ hidden layers with n_l neurons each, and an output layer with K neurons (classes).

For each hidden layer $l \in \{1, \ldots, L-1\}$, the computation is:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \tag{21}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}), \tag{22}$$

where $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix, $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ is the bias vector, and f is an activation function (e.g., ReLU, tanh).

For the output layer ($l = L$):

- **Binary Classification**: The output is

$$\hat{y} = \sigma(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}) = \frac{1}{1 + e^{-z^{(L)}}},$$

  where σ is the sigmoid function.

- **Multi-class Classification**: The output is

$$\hat{y}_i = \text{softmax}(\mathbf{z}^{(L)})_i = \frac{e^{z_i^{(L)}}}{\sum_{j=1}^{K} e^{z_j^{(L)}}}, \quad \text{for } i = 1, \ldots, K.$$

- **Binary Cross-Entropy**: For binary classification, the loss is:

$$\mathcal{L} = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right].$$

- **Categorical Cross-Entropy**: For multi-class classification, the loss is:

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log(\hat{y}_i).$$

4. Experimental Setup

4.1. Dataset

To evaluate our JavaScript detector, we constructed two new datasets to assess its performance. These datasets are designed to provide a balanced and diverse representation of both benign and malicious JavaScript samples. The detailed class distribution of these datasets is summarized in Table 1. The datasets are described as follows:

Table 1
Composition of the Dataset_1 and Dataset_2

Source	Benign	Malicious	Total
JS150	150 000	0	150 000
JS-Malicious-Dataset	0	1 357	1 357
Dataset_1	**150 000**	**1 357**	**151 357**
JavaScript_Datasets	8 079	8 500	16 579
Dataset_2	**8 079**	**8 500**	**16 579**

- **Dataset_1**: This corpus is composed of a large-scale benign dataset, JS150 Raff et al. (2021), and a publicly available malicious corpus, JS-Malicious-Dataset GeeksOnSecurity (2022). A subset of samples from JS150 is included to ensure balanced coverage of both classes while preserving real-world diversity.

- **Dataset_2**: Sourced from a GitHub repository ZZN0508 (2023), this dataset separates benign (`goodjs`) and malicious (`badjs`) scripts. It includes all 8,079 benign and 8,500 malicious samples.

We employ the DeepSeek code model to automatically deobfuscate obfuscated JavaScript code. We utilize carefully crafted prompt templates to detect and decode string encodings (e.g., Hex or Base64), standardize variable names, and reconstruct control flows while preserving the original functionality. The deobfuscated code is then transformed into an Abstract Syntax Tree (AST) with data flow annotations using the Esprima parser. For sequence-based models (e.g., BERT or LSTM), we perform language feature segmentation, while for graph-based models (e.g., GNN), we apply the METIS algorithm to partition the AST into 8 subgraphs compatible with the PyG format. For model development, we randomly partition the combined corpus, allocating 80% for training, 10% for validation, and 10% as a held-out test set, ensuring the benign-to-malicious ratio is maintained across all splits.

For model development we randomly sample 80% of the combined corpus for training, 10% for validation, and retain 10% as a held-out test set whilst preserving the benign/malicious ratio.

4.2. Baselines

We compare our method against the following baseline models:

- **BERT**: A Transformer-based pre-trained language model that revolutionized natural language processing through bidirectional contextual awareness. Its core architecture employs 12 Transformer encoder layers (768 hidden dimensions), pre-trained on text corpora (e.g., Wikipedia) using Masked Language Modeling (MLM) and Next Sentence Prediction (NSP) objectives.

- **CodeBERT**Feng et al. (2020): A specialized pre-trained language model designed for programming-related tasks, building upon the Transformer architecture. Unlike general-purpose language models like BERT, CodeBERT is uniquely trained on both programming languages and natural language text, enabling it to understand the nuanced relationship between code and human language. The model employs innovative training objectives including MLM and RTD, which specifically enhance its ability to comprehend code syntax and semantics.
- **LSTM**: A bidirectional LSTM with 2 layers, 256 hidden dimensions, dropout of 0.2, and an embedding dimension of 128. The model is trained using a batch size of 32 and an initial learning rate of 1e-3 with a step-wise decay schedule.
- **CNN**: A 1D convolutional architecture with 3 layers, kernel sizes of [3, 5, 7], 128 filters per layer, ReLU activations, and global max pooling. The model is trained with a batch size of 32 and a learning rate of 5e-4.
- **GCN**Kipf and Welling (2016): A Graph Convolutional Network with 3 layers, 128 hidden dimensions, ReLU activations, and dropout of 0.5. The model processes AST graphs using message passing between adjacent nodes, with graph-level pooling for final classification. We train with a batch size of 16 and a learning rate of 5e-4 with weight decay of 1e-5.

All baseline models process tokenized code sequences and are trained using the Adam optimizer with weight decay of 1e-5. Early stopping with a patience of 10 epochs based on validation F1-score is employed across all models.

4.3. Evaluation Metrics

We evaluate model performance using the following metrics:

- **Accuracy**: The proportion of correctly classified samples.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{23}$$

- **Precision**: The proportion of true positive predictions among all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{24}$$

- **Recall**: The proportion of true positive predictions among all actual positives.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{25}$$

- **F1-score**: The harmonic mean of precision and recall.

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{26}$$

- **AUC-ROC**: The area under the Receiver Operating Characteristic curve, which plots the true positive rate against the false positive rate at various threshold settings.

$$\text{AUC-ROC} = \int_0^1 TPR(FPR^{-1}(t))\, dt \tag{27}$$

where TPR is the true positive rate and FPR is the false positive rate.

For malicious code detection, we consider the malicious class as the positive class. These metrics provide a comprehensive evaluation of model performance, with F1-score being particularly important due to the security implications of both false positives and false negatives in malware detection.

4.4. Implementation Details

All experiments were conducted using PyTorch 1.12.0 and PyTorch Geometric 2.2.0 on an NVIDIA RTX 3090 GPU with 24GB RAM. Our proposed Cluster-GT model uses 3 Cluster-GT layers with 128-dimensional hidden states, 8 attention heads per layer, and dropout rate of 0.1. We employed the METIS algorithm to partition each AST into 8 clusters depending on graph size. Hyperparameters for all models were optimized via grid search on the validation set using a 10-fold cross-validation strategy. For the dual-granularity kernelized attention mechanism, we used the tensor product kernel (N2C-Attn-T). The DeepSeek-Coder-7B-InstructGuo et al. (2024) model was accessed through an API interface for the deobfuscation pipeline.

5. Results and Discussion

5.1. Overall Performance Comparison

To evaluate the comprehensive performance of our proposed method, we conducted a comparative analysis against a suite of baseline models on Dataset_1 and Dataset_2, with the results presented in Table 2. In this evaluation, our method leverages its integrated LLM-based deobfuscation pipeline, whereas all baseline models (BERT, CodeBERT, LSTM, CNN, and GCN) were evaluated on the original, potentially obfuscated datasets. This experimental design aims to simulate a realistic scenario, pitting our end-to-end solution against existing standard methodologies.

The results clearly demonstrate the decisive superiority of our method. On Dataset_1, our method achieves an F1-score of 0.9464 and an AUC of 0.9750, significantly outperforming the best baseline, CodeBERT, which recorded an F1-score of 0.8551 and an AUC of 0.9242. This equates to a relative improvement of approximately 9.13% in F1-score and 5.08% in AUC. The performance gap widened on the more challenging Dataset_2, where our method attained an F1-score of 0.9771 and an AUC of 0.9818. In contrast, CodeBERT achieved an F1-score of 0.8605 and an AUC of 0.9430, marking a relative improvement for our method of approximately 11.66% in F1-score and 3.88% in AUC.

The baseline models exhibit pronounced limitations when processing unprocessed data. Sequence-based models

Table 2
Performance Comparison of Our Method Against Baselines on $Dataset_1$ and $Dataset_2$. Note: Values in parentheses () indicate the performance gap between each baseline method and our proposed approach

Model	$Dataset_1$				$Dataset_2$			
	Precision	Recall	F1-Score	AUC	Precision	Recall	F1-Score	AUC
BERT	0.8216 (-13.10)	0.8571 (-8.32)	0.8390 (-10.74)	0.9150 (-6.00)	0.8316 (-13.95)	0.8771 (-10.61)	0.8537 (-12.34)	0.9242 (-5.76)
LSTM	0.7552 (-19.74)	0.8650 (-7.53)	0.8064 (-14.00)	0.8700 (-10.50)	0.8000 (-17.11)	0.8811 (-10.21)	0.8386 (-13.85)	0.8911 (-9.07)
CodeBERT	0.8489 (-10.37)	0.8615 (-7.88)	0.8551 (-9.13)	0.9242 (-5.08)	0.8701 (-10.10)	0.8511 (-13.21)	0.8605 (-11.66)	0.9430 (-3.88)
GCN	0.8325 (-12.01)	0.8623 (-7.80)	0.8471 (-9.93)	0.9189 (-5.61)	0.8737 (-9.74)	0.8611 (-12.21)	0.8674 (-10.97)	0.9480 (-3.38)
CNN	0.8071 (-14.55)	0.8485 (-9.18)	0.8273 (-11.91)	0.8350 (-14.00)	0.8471 (-12.40)	0.8585 (-12.47)	0.8528 (-12.43)	0.9150 (-6.68)
Our Method	**0.9526**	**0.9403**	**0.9464**	**0.9750**	**0.9711**	**0.9832**	**0.9771**	**0.9818**

Table 3
Performance Impact of LLM-based Deobfuscation Across All Models

Model	Condition	$Dataset_1$				$Dataset_2$			
		Precision	Recall	F1-Score	AUC	Precision	Recall	F1-Score	AUC
BERT	Obfuscated	0.8216	0.8571	0.8390	0.9150	0.8316	0.8771	0.8537	0.9242
	Deobfuscated	0.8914	0.8732	0.8822	0.9112	0.9014	0.8832	0.8922	0.9400
	Improvement (%)	+6.98	+1.61	+4.32	-0.38	+6.98	+0.61	+3.85	+1.58
LSTM	Obfuscated	0.7552	0.8650	0.8064	0.8700	0.8000	0.8811	0.8386	0.8911
	Deobfuscated	0.8026	0.9212	0.8578	0.9090	0.8516	0.9351	0.8914	0.9332
	Improvement (%)	+4.74	+5.62	+5.14	+3.90	+5.16	+5.40	+5.28	+4.21
CodeBERT	Obfuscated	0.8489	0.8615	0.8551	0.9242	0.8701	0.8511	0.8605	0.9430
	Deobfuscated	0.9087	0.8882	0.8983	0.9422	0.9172	0.8963	0.9066	0.9601
	Improvement (%)	+5.98	+2.67	+4.32	+1.80	+4.71	+4.52	+4.61	+1.71
GCN	Obfuscated	0.8325	0.8623	0.8471	0.9189	0.8737	0.8611	0.8674	0.9480
	Deobfuscated	0.9323	0.8611	0.8952	0.9512	0.9223	0.8582	0.8885	0.9611
	Improvement (%)	+9.98	-0.12	+4.81	+3.23	+4.86	-0.29	+2.11	+1.31
CNN	Obfuscated	0.8071	0.8485	0.8273	0.8350	0.8471	0.8585	0.8528	0.9150
	Deobfuscated	0.8836	0.8936	0.8886	0.9350	0.9036	0.8936	0.8986	0.9450
	Improvement (%)	+7.65	+4.51	+6.13	+10.00	+5.65	+3.51	+4.58	+3.00
Our Method	Obfuscated	0.9100	0.9000	0.9050	0.9423	0.9201	0.9311	0.9256	0.9542
	Deobfuscated	**0.9526**	**0.9403**	**0.9464**	**0.9750**	**0.9711**	**0.9832**	**0.9771**	**0.9818**
	Improvement (%)	**+4.26**	**+4.03**	**+4.14**	**+3.27**	**+5.10**	**+5.21**	**+5.15**	**+2.76**

like BERT and LSTM are particularly vulnerable, as code obfuscation techniques such as variable renaming and control flow flattening disrupt the sequential semantics upon which they depend. Even the graph-based GCN model, despite utilizing Abstract Syntax Trees (ASTs), only achieved an F1-score of 0.8674 on $Dataset_2$, indicating its inability to capture the multi-granularity structural information that our node-to-cluster attention mechanism effectively extracts.

To further investigate the models' behavior under strict false positive constraints, we evaluated the True Positive Rate (TPR) at various low False Positive Rate (FPR) levels. As shown in Table 4, our method consistently outperforms both CodeBERT and GCN across all FPR thresholds, particularly excelling in extremely low-FPR regimes (e.g., 0.2473 TPR at 0.0001 FPR). The corresponding ROC curves in Figure 2 further illustrate this advantage, with our method achieving the highest AUC of 0.9818 and dominating other baselines across the entire FPR range, especially in the low-FPR region relevant to security-critical scenarios.

The exceptional performance of our method is attributable to its core design principles:

- **Integrated Deobfuscation Pipeline:** By normalizing code representations via an LLM prior to feature extraction, our method fundamentally reduces the complexity of the detection task.
- **Robust Graph-based Representation:** The AST-based graph structure preserves critical syntactic and logical relationships, offering resilience against structural obfuscation that sequence-based models lack.

Table 4
Controlled FPR Evaluation (TPR @ FPR Levels) on $Dataset_2$. Note: Values in parentheses indicate the performance multiplier of our method versus each baseline

Model	TPR @ FPR Level			
	0.0001	0.001	0.01	0.1
CodeBERT	0.0513 (4.82×)	0.0655 (5.91×)	0.3171 (2.53×)	0.8864 (1.09×)
GCN	0.0189 (13.09×)	0.0425 (9.11×)	0.3813 (2.10×)	0.8816 (1.09×)
Our Method	**0.2473**	**0.3872**	**0.8008**	**0.9624**

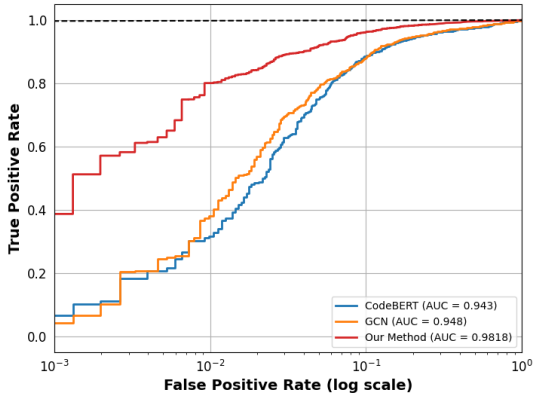


Figure 2: ROC curves

- **Multi-granularity Feature Fusion:** The node-to-cluster attention mechanism enables the model to capture features at multiple levels of abstraction, from local code constructs to global patterns, facilitating a deeper understanding of the code's behavior.

These advantages establish our method as a superior solution in terms of precision and generalization, making it an ideal choice for demanding security applications.

5.2. Impact of Deobfuscation

Table 3 illustrate the performance of models on $Dataset_1$ and $Dataset_2$ before and after applying our LLM-based deobfuscation pipeline, revealing substantial improvements that highlight the effectiveness of this approach in normalizing code structures and enhancing feature extraction.

On $Dataset_1$, BERT's F1-score rises from 0.8390 in obfuscated code to 0.8822 after deobfuscation, a 4.32% improvement, while LSTM shows a more pronounced gain, increasing from 0.8064 to 0.8578, or 5.14%. CodeBERT improves from 0.8551 to 0.8983, a 4.32% boost, and CNN advances from 0.8273 to 0.8886, achieving a 6.13% enhancement. On $Dataset_2$, similar trends emerge: BERT's F1-score grows from 0.8537 to 0.8922, a 3.85% increase; LSTM improves from 0.8386 to 0.8914, a 5.28% gain; CodeBERT rises from 0.8605 to 0.9066, up 4.61%; and CNN increases from 0.8528 to 0.8986, a 4.58% improvement.

Graph-based models also benefit, though to a lesser extent. GCN's F1-score on $Dataset_1$ improves from 0.8471 to 0.8952, a 4.81% increase, and on $Dataset_2$ from 0.8674 to 0.8885, a 2.11% gain. Our method, already robust, sees its F1-score rise from 0.9050 to 0.9464 on $Dataset_1$, a 4.14% improvement, and from 0.9256 to 0.9771 on $Dataset_2$, a 5.15% enhancement.

Sequence-based models collectively achieve an average F1-score improvement of 4.98% on $Dataset_1$ and 4.58% on $Dataset_2$, significantly higher than the 4.48% and 3.63% averages for graph-based models on the respective datasets. This disparity underscores the greater sensitivity of sequence-based models to obfuscation techniques, such as variable renaming and control flow manipulation, which disrupt sequential token relationships. Our method, leveraging robust graph-based representations and multi-granularity feature integration, maintains superior performance in both obfuscated and deobfuscated scenarios, with minimal performance gaps. The consistent enhancements across all models affirm the critical role of our LLM-based deobfuscation pipeline in boosting detection accuracy, particularly for sequence-based models, while our method's inherent resilience ensures unmatched reliability.

5.3. Analysis of Architectural Advantage

Table 3 demonstrates that our method's superior performance is not solely dependent on its deobfuscation pipeline but is also rooted in its advanced architectural design.

When all models were evaluated on the original obfuscated datasets, our method achieved F1-scores of 0.9050 and 0.9256 on $Dataset_1$ and $Dataset_2$, respectively. These results significantly surpass those of the best-performing baseline, CodeBERT, which scored 0.8551 and 0.8605. This indicates that our model's graph-based structure is inherently more resilient to common obfuscation techniques. Furthermore, this performance gap persists on the deobfuscated datasets. Our method obtains F1-scores of 0.9464 and 0.9771, while CodeBERT reaches 0.8983 and 0.9066, confirming our model's superior capability to learn from normalized code representations.

This consistent outperformance is attributable to several core architectural features:

- **Preservation of hierarchical code structure:** By leveraging ASTs, our model effectively preserves the code's hierarchical integrity, enabling a more robust detection of malicious patterns that are agnostic to superficial changes in code syntax.
- **Multi-granularity feature integration:** The Node-to-Cluster Attention mechanism facilitates the integration of features at various abstraction levels, allowing the model to capture both fine-grained local patterns and coarse-grained global patterns.
- **Superior precision and generalization:** The architecture demonstrates excellent generalization, with the

Table 5
Performance of Our Method Across Different Node Counts and Cluster Kernels

Cluster Kernels	1000 Nodes			2000 Nodes			4000 Nodes		
	Accuracy	Recall	F1-Score	Accuracy	Recall	F1-Score	Accuracy	Recall	F1-Score
1	0.9346	0.9281	0.9314	0.9346	0.9192	0.9268	0.9542	0.9242	0.9379
4	0.9474	0.9374	0.9425	0.9673	0.9573	0.9620	0.9671	0.9671	0.9670
8	0.9671	0.9539	0.9607	0.9769	0.9605	0.9686	0.9868	0.9871	0.9870

F1-score difference between Dataset_1 and Dataset_2 for our final model being minimal. This showcases its ability to maintain high performance on diverse codebases.

These design choices collectively create a more powerful and reliable model for malicious code detection, independent of the benefits provided by the deobfuscation pre-processing.

5.4. Scalability Analysis

Table 5 presents the performance of our method across varying numbers of nodes (1000, 2000, 4000) and cluster kernels (1, 4, 8) in the Abstract Syntax Tree (AST) graphs, evaluated using accuracy, recall, and F1-score. These results demonstrate the scalability of our method, highlighting its ability to effectively integrate structural information as graph size increases and its enhanced performance with finer cluster granularity.

As the number of nodes in the AST graph increases from 1000 to 4000, our method consistently improves across all metrics. For instance, with eight cluster kernels, the F1-score rises from 0.9607 at 1000 nodes to 0.9870 at 4000 nodes, a 2.7% improvement. This trend underscores the method's ability to leverage richer structural information in larger graphs, enabling more precise detection of malicious code patterns. Similarly, accuracy and recall exhibit significant gains, reaching 0.9868 and 0.9871, respectively, at 4000 nodes with eight clusters, reflecting a balanced performance that minimizes both false positives and false negatives.

The impact of cluster granularity is particularly pronounced in larger graphs. At 4000 nodes, increasing the number of cluster kernels from one to eight boosts the F1-score from 0.9379 to 0.9870, a 5.2% enhancement, compared to a 3.1% improvement (0.9314 to 0.9607) at 1000 nodes. This indicates that finer cluster divisions are more effective in large-scale graphs, where diverse structural patterns require nuanced feature integration. In contrast, with a single cluster kernel, performance remains suboptimal across all node counts, with an F1-score of only 0.9379 at 4000 nodes, highlighting the critical role of the Node-to-Cluster Attention mechanism in capturing multi-granularity features.

Furthermore, the performance gains from increasing cluster kernels diminish slightly at smaller node counts. For example, at 1000 nodes, the F1-score improves by 1.9% from four to eight clusters (0.9425 to 0.9607), compared to 2.1% at 4000 nodes (0.9670 to 0.9870). This suggests that our method's scalability is most pronounced in larger graphs, where additional clusters yield greater benefits. The balanced performance across metrics at 4000 nodes and eight clusters, with near-identical accuracy, recall, and F1-score, further demonstrates the method's robustness in handling complex graph structures.

These findings confirm that our method scales effectively with increasing graph size and cluster granularity, making it well-suited for real-world applications where AST graphs may vary significantly in scale and complexity.

6. Limitations and Future Work

The experimental results demonstrate that our method achieves significant performance improvements over the baseline approaches. In this section, we analyze the limitations of the proposed method and discuss potential opportunities for future enhancements:

- **Limitations of AST-Based Representation**: Our approach primarily leverages Abstract Syntax Trees (ASTs) for code representation. While ASTs capture rich structural and syntactic information, alternative representations such as Control Flow Graphs (CFGs), Program Dependency Graphs (PDGs), and bytecode/IR-based features can provide complementary semantic and behavioral insights. Future work should explore hybrid representations combining ASTs with these techniques to enhance detection accuracy and robustness.

- **Limitations in Dynamic Feature Analysis**: While our method incorporates two JavaScript-specific characteristics, its handling of JavaScript's dynamic nature remains incomplete. Unlike other programming languages where malicious behavior is often statically identifiable, JavaScript's malicious patterns frequently manifest during dynamic execution (e.g., through eval(), dynamic property access, or runtime code generation). Future work could integrate dynamic analysis through browser instrumentation or execution tracing to capture runtime behaviors, subsequently encoding these features as additional node/edge attributes in our graph representation.

- **Limitations in Model Selection**: Our approach primarily employs DeepSeek's base version for deobfuscation tasks, with our core contribution lying in the refined prompt engineering for large language

models. While utilizing more advanced LLM versions (such as DeepSeek-V2 or GPT-4) could potentially yield better deobfuscation results, we deliberately focused on prompt optimization and were constrained by the prohibitive costs of premium LLM APIs. Future work should systematically evaluate the cost-benefit tradeoffs of state-of-the-art LLMs across different JavaScript obfuscation patterns, particularly for dynamic features like just-in-time compilation and prototype pollution that challenge current static analysis approaches.

7. Conclusion

The proliferation of web applications has intensified security risks from malicious JavaScript, where advanced obfuscation techniques and dynamic language features (e.g., nested closures) challenge conventional detection methods. Our work introduces a novel defense framework that synergizes LLM-based semantic deobfuscation with hierarchical graph learning: (1) A multi-stage prompt engineering pipeline reconstructs original code semantics from obfuscated inputs, generating normalized AST representations; (2) To overcome structural noise from JavaScript's dynamic typing and scope fragmentation, we develop a Cluster-wise Graph Transformer that jointly models node-level semantics and cluster-induced relationships through innovative node-to-cluster attention. Experimental validation shows our method achieves 94.64% and 97.71% F1-scores (10.74%/13.85% absolute gains over SOTA) with 4.82×-5.91× higher TPR at critical FPR thresholds, while maintaining exceptional cross-dataset consistency (3.07% F1 variance). This hybrid paradigm of semantic-aware LLMs and structure-preserving GNNs establishes a new foundation for robust malware detection across evolving threat landscapes.

References

Abadeer, M., Moeini, B., Sewell, E., Branco, P., Ventura, F., Shi, W., 2022. Dynamic extraction of bert-based embeddings for the detection of malicious javascript, in: Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering, pp. 110–119.

Behera, C.K., Bhaskari, D.L., 2015. Different obfuscation techniques for code protection. Procedia Computer Science 70, 757–763.

Chua, L.O., Roska, T., 2002. The cnn paradigm. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications 40, 147–156.

Corso, G., Stark, H., Jegelka, S., Jaakkola, T., Barzilay, R., 2024. Graph neural networks. Nature Reviews Methods Primers 4, 17.

Fang, Y., Huang, C., Liu, L., Xue, M., 2018. Research on malicious javascript detection technology based on lstm. IEEE Access 6, 59118–59125.

Fang, Y., Huang, C., Su, Y., Qiu, Y., 2020. Detecting malicious javascript code based on semantic analysis. Computers & Security 93, 101764.

Fang, Y., Huang, C., Zeng, M., Zhao, Z., Huang, C., 2022. Jstrong: Malicious javascript detection based on code semantic representation and graph neural network. Computers & Security 118, 102715.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 .

GeeksOnSecurity, 2022. JS-Malicious-Dataset: A Collection of Malicious JavaScript for Security Research. https://github.com/geeksonsecurity/js-malicious-dataset. GitHub repository, Accessed: 2025-07-07.

Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J., 2016. Lstm: A search space odyssey. IEEE transactions on neural networks and learning systems 28, 2222–2232.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al., 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948 .

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y.K., Luo, F., Xiong, Y., Liang, W., 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. URL: https://arxiv.org/abs/2401.14196, arXiv:2401.14196.

Huang, S., Song, Y., Zhou, J., Lin, Z., 2024. Cluster-wise graph transformer with dual-granularity kernelized attention. Advances in Neural Information Processing Systems 37, 33376–33401.

Kipf, T.N., Welling, M., 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 .

Koroteev, M.V., 2021. Bert: a review of applications in natural language processing and understanding. arXiv preprint arXiv:2103.11943 .

Lee, C., Son, S., 2023. Adcpg: Classifying javascript code property graphs with explanations for ad and tracker blocking, in: Proceedings of the 2023 ACM SIGSAC conference on computer and communications security, pp. 3505–3518.

Li, X., Wang, Z., Wang, Q., Yan, S., Xie, T., Mei, H., 2016. Relationship-aware code search for javascript frameworks, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 690–701.

Liang, H., Yang, Y., Sun, L., Jiang, L., 2019. Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg, in: 2019 International Joint Conference on Neural Networks (IJCNN), IEEE. pp. 1–8.

Liu, R., Wang, Y., Guo, Z., Xu, H., Qin, Z., Ma, W., Zhang, F., 2024. Transurl: Improving malicious url detection with multi-layer transformer encoding and multi-scale pyramid features. Computer Networks 253, 110707.

Liu, R., Wang, Y., Xu, H., Qin, Z., Zhang, F., Liu, Y., Cao, Z., 2025a. Pmanet: Malicious url detection via post-trained language model guided multi-level feature attention network. Information Fusion 113, 102638.

Liu, R., Wang, Y., Xu, H., Sun, J., Zhang, F., Li, P., Guo, Z., 2025b. Vul-lmgnns: Fusing language models and online-distilled graph neural networks for code vulnerability detection. Information Fusion 115, 102748. URL: https://www.sciencedirect.com/science/article/pii/S1566253524005268, doi:https://doi.org/10.1016/j.inffus.2024.102748.

Malik, R.S., Patra, J., Pradel, M., 2019. Nl2type: Inferring javascript function types from natural language information, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 304–315.

Raff, E., et al., 2021. JS150k: A Dataset of JavaScript Programs with Rich and Diverse Semantics. Technical Report. SRI International. URL: https://www.sri.inf.ethz.ch/js150. accessed: 2025-07-07.

Riedmiller, M., Lernen, A., 2014. Multi layer perceptron. Machine learning lab special lecture, University of Freiburg 24.

Rozi, M.F., Ban, T., Ozawa, S., Kim, S., Takahashi, T., Inoue, D., 2021. Jstrack: Enriching malicious javascript detection based on ast graph analysis and attention mechanism, in: Neural Information Processing: 28th International Conference, ICONIP 2021, Sanur, Bali, Indonesia, December 8–12, 2021, Proceedings, Part II 28, Springer. pp. 669–680.

Rozi, M.F., Kim, S., Ozawa, S., 2020. Deep neural networks for malicious javascript detection using bytecode sequences, in: 2020 International Joint Conference on Neural Networks (IJCNN), IEEE. pp. 1–8.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E., 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. 49. URL: https://doi.org/10.1145/2886012, doi:10.1145/2886012.

Sheneamer, A., 2024. Vulnerable javascript functions detection using stacking of convolutional neural networks. PeerJ Computer Science 10, e1838.

Sheng, Z., Song, L., Wang, Y., 2025. Dynamic feature fusion: Combining global graph structures and local semantics for blockchain phishing detection. IEEE Transactions on Network and Service Management .

Skolka, P., Staicu, C.A., Pradel, M., 2019. Anything to hide? studying minified and obfuscated code in the web, in: The world wide web conference, pp. 1735–1746.

Song, L., Ding, S.H., Tian, Y., Li, L.T., Ou, W., Charland, P., Walenstein, A., 2025. Obfuscated clone search in javascript based on reinforcement subsequence learning. ACM Transactions on Software Engineering and Methodology .

Song, X., Chen, C., Cui, B., Fu, J., 2020. Malicious javascript detection based on bidirectional lstm model. Applied Sciences 10, 3440.

Stokes, J.W., Agrawal, R., McDonald, G., Hausknecht, M., 2019. Scriptnet: Neural static analysis for malicious javascript detection, in: MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM), IEEE. pp. 1–8.

Sun, H., Chen, M., Weng, J., Liu, Z., Geng, G., 2021. Anomaly detection for in-vehicle network using cnn-lstm with attention mechanism. IEEE Transactions on Vehicular Technology 70, 10880–10893.

Sun, J., Jia, Y., Wang, Y., Tian, Y., Zhang, S., 2025. Ethereum fraud detection via joint transaction language model and graph representation learning. Information Fusion 120, 103074.

Tian, Y., Yumin, Z., Jia, Y., Sun, J., Wang, Y., 2025. Webguard++: Interpretable malicious url detection via bidirectional fusion of html subgraphs and multi-scale convolutional bert. arXiv preprint arXiv:2506.19356 .

Wang, Y., Xu, H., Guo, Z., Qin, Z., Ren, K., 2022. Snwf: Website fingerprinting attack by ensembling the snapshot of deep learning. IEEE Transactions on Information Forensics and Security 17, 1214–1226.

Wang, Y., Zhu, W., Xu, H., Qin, Z., Ren, K., Ma, W., 2023. A large-scale pretrained deep model for phishing url detection, in: ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE. pp. 1–5.

Wei, L., Wang, Y., Li, X., Li, J., Huang, Y., Liu, Z., 2025. A detection method for malware communication traffic via encrypted traffic analysis. IEEE Internet of Things Journal .

Wirfs-Brock, A., Eich, B., 2020. Javascript: the first 20 years. Proc. ACM Program. Lang. 4. URL: https://doi.org/10.1145/3386327, doi:10.1145/3386327.

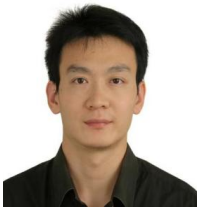
Xiao, W., Shi, C., Chen, M., Liu, Z., Chen, M., Song, H.H., 2025. Graphedge: Dynamic graph partition and task scheduling for gnns computing in edge network. Information Fusion , 103329.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE symposium on security and privacy, IEEE. pp. 590–604.

Zaremba, W., Sutskever, I., Vinyals, O., 2014. Recurrent neural network regularization. arXiv preprint arXiv:1409.2329 .

Zhang, T., Du, C., Zhou, Y., Guan, Q., Liu, Z., Huang, X., Gong, Z., Deng, L., Li, Y., 2024. Hybrid transfer and self-supervised learning approaches in neural networks for intelligent vehicle intrusion detection and analysis. IEEE Internet of Things Journal .

ZZN0508, 2023. JavaScript_Datasets: JavaScript Malicious Code Dataset. https://github.com/ZZN0508/JavaScript_Datasets. GitHub repository, Accessed: 2025-07-07.



Zhihong Liang holds a Master of Engineering in Computer Applications from South China University of Technology. He is a Senior Professor-level Senior Engineer, a member of the Artificial Intelligence Committee of the Chinese Society for Electrical Engineering, and a member of the Computer Security Committee of the China Computer Federation. His long-term research and engineering activities focus on network security, data security, artificial intelligence, and cloud computing. He has authored or co-authored more than ten academic papers.

Wang Xin holds a Master of Engineering degree in Mechanical Design and Theory from Northeast Agricultural University. She is currently employed at Xidian University, where she is engaged in the research and development of industrial software and intelligent security.

Zhenhuang Hu is currently pursuing his graduate studies at the Hangzhou Institute of Technology of Xidian University. He obtained his Bachelor's degree from Harbin University of Science and Technology. His primary research interests encompass malicious code detection, network security, and applications of large language models.

Liangliang Song is a graduate student at the Hangzhou Institute of Technology of Xidian University. He received his Bachelor's degree from Hangzhou Dianzi University. His research focuses on deep learning, network security, and large language models.

Lin Chen received his M.E. in Computer Science and Technology from South China University of Technology. He is an engineer whose research centers on AI security in power systems and network security, with more than ten relevant academic papers published.

Jingjing Guo received her B.S., M.S., and Ph.D. degrees from Xidian University, China. Her research focuses on AI system security, privacy protection, UAV and IoT security. She has published over 50 papers in top journals such as IEEE JSAC and TMC, and holds more than 60 patents.

Yanbin Wang is currently an Associate Professor at the Hangzhou Institute of Technology, Xidian University. He received his Ph.D. in Cybersecurity from Zhejiang University. His research focuses on AI-powered web security, software security, and blockchain security. Dr. Wang has led or participated in over 10 key research projects, including National Key R&D Programs and provincial-level initiatives. With 60+ publications in top-tier international journals and conferences, he previously

served as an editorial board member for three SCI-indexed journals.

Ye Tian is an associate professor at Xidian University. He received Ph.D. from Harbin Engineering University. His current research interests include artificial intelligence security and multimodal information processing.