Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.086163

ARTICLE

ASTBertX: Multilingual Sequence–Structure Fusion for Exploit Type Identification in Malware Detection

Xinglong Cao, Cong Wang*, Jie Yan, Songcan Yu and Mingze He

Police Integration Computing Key Laboratory of Sichuan Province, Sichuan Police College, Chengdu, China

*Corresponding Author: Cong Wang. Email: cong-wang@foxmail.com

Received: 25 May 2026; Accepted: 13 July 2026; Published: 15 September 2026

ABSTRACT: There is currently a lack of systematic research on the fine-grained detection of multi-language and multi-type exploit scripts. To address this gap, this study proposes a model named ASTBertX (AST + BERT + XGBoost) for identifying the specific exploit types of malicious scripts; the model organically integrates code sequence semantics with structural semantics. First, the model utilizes the pre-trained model GraphCodeBERT to extract contextual semantic representations of the scripts; simultaneously, it introduces semantic enhancement nodes into the Abstract Syntax Tree (AST) and employs GATv2 to learn the AST's structural representation. These two representations are mapped into intermediate vectors of uniform dimensionality via multi-layer perceptrons, followed by feature alignment training through a projection layer containing a temporary proxy classifier; finally, the aligned features are fed into an XGBoost classifier to output predictions regarding the exploit type. Upon completion of training, the proxy classifier is discarded, retaining only the parameters of the projection layer. Experimental results demonstrate that the method achieves an accuracy of 95% in five-class classification tasks and 99% in binary classification tasks (malicious vs. benign). The model not only matches baseline methods in binary detection performance but also effectively identifies specific exploit types.

KEYWORDS: Exploit type; cross language; malware detection; malicious code

1 Introduction

The pervasiveness of software security vulnerabilities has rendered them a critical source of risk in cyberspace. Despite notable advancements in vulnerability discovery, patch management, and runtime protection achieved by both academia and industry, the temporal gap between vulnerability disclosure and the deployment of patches continues to afford attackers a window of opportunity. Concurrently, the rapid evolution of large language models has substantially lowered the barrier to the automated generation and widespread deployment of exploit code. For instance, models such as Claude and GPT-4 exhibit strong capabilities in code comprehension and generation, enabling attackers with limited security expertise to launch high-impact attacks. This poses a severe threat to national critical infrastructure, enterprise information systems, and even the privacy of individual citizens.

The typical workflow of exploit script execution is illustrated in Fig. 1. Attackers first identify vulnerabilities on the target server, including overflow-type issues (e.g., buffer overflow, stack overflow, heap overflow), injection-type issues (e.g., command injection, SQL injection, cross-site scripting), DoS-type issues (e.g., denial of service, resource exhaustion), and file-path-related issues (e.g., bypass, unauthorized access, logical flaws). Subsequently, they develop exploit code tailored to the specific vulnerability class. Upon successful

exploitation, attackers gain unauthorized control over the target system and may execute further malicious actions, including privilege escalation, data exfiltration, backdoor persistence, and lateral movement, etc.

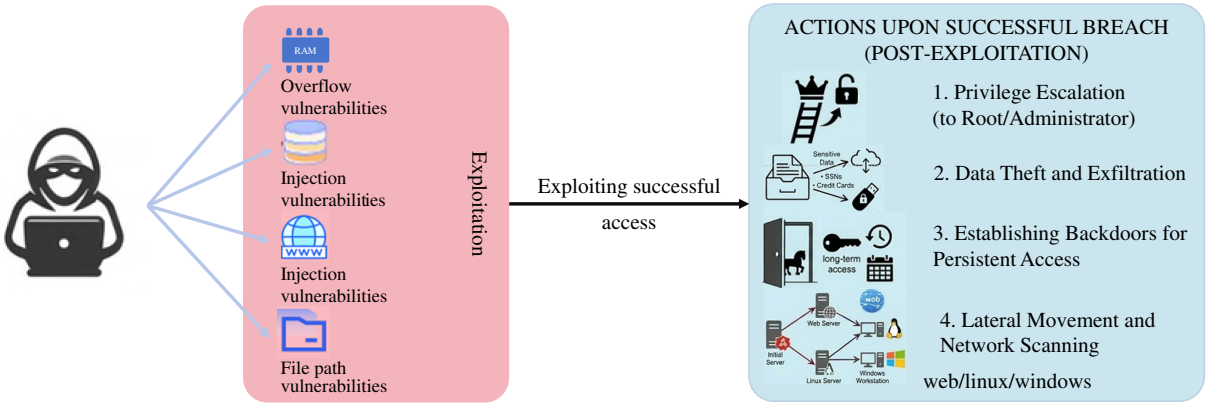

Figure 1: Typical process of vulnerability exploitation.

In recent years, large-scale cybersecurity incidents driven by exploit scripts have become increasingly frequent. For example, in November 2023, the hacker group CyberAv3ngers launched attacks against industrial control devices manufactured by Unitronics in Israel, compromising more than 100 units that are widely deployed in water and wastewater treatment systems [1]. In February 2024, ransomware attacks targeted IoT medical devices in several hospitals across the United States, forcing portions of the healthcare infrastructure to revert to manual operation [2]. In May 2024, a major smart city project in Asia experienced a distributed denial-of-service attack, resulting in service disruptions across IoT systems such as traffic lights, surveillance cameras, and waste-management infrastructure [3]. Security researchers identified a long-standing unpatched vulnerability in AVTECH IP cameras, which attackers exploited to propagate the Mirai malware. Although the vulnerability had been publicly known since 2019, it was not assigned a CVE identifier until 2024 [4]. In July 2025, Google, in collaboration with Human Security and Trend Micro, disclosed the BadBox 2.0 botnet incident. More than 10 million devices—including smart TVs, digital projectors, and in-vehicle infotainment systems—were compromised, making it one of the largest known IoT botnets targeting smart televisions [5]. Thus, in some cases, attackers merely need to execute a publicly available or automatically generated exploit script to trigger large-scale cascading cybersecurity incidents.

Current mainstream defense strategies primarily follow two directions. The first category focuses on identifying and patching the vulnerabilities themselves, employing techniques such as static code analysis, dynamic fuzz testing, and formal verification, with the goal of reducing exploitable security flaws during software development and deployment [6–9]. However, before vulnerabilities are effectively discovered and patched, attackers can still exploit them to launch targeted attacks. The second category emphasizes malicious attack code detection, where current research typically focuses on post-intrusion behavioral analysis, such as backdoor installation, ransomware execution, and cryptomining hijacking [10–14]. In contrast, automated identification and classification of exploit scripts remain relatively underexplored, even though such scripts often serve as the primary entry point for triggering vulnerability exploitation. However, most existing detection methods are limited to specific programming languages, such as PowerShell [10,11], PHP [12,13], and JavaScript [14]. These approaches generally operate as post-incident detection mechanisms, offering limited value for timely prevention before or during an attack. Meanwhile, research on detecting specific attack behaviors has made progress, but it remains largely focused on single attack types, such

as buffer-overflow attacks [15,16], code-injection attacks [17,18], denial-of-service attacks [19,20], and file-upload attacks [21,22]. Such "fragmented" detection strategies struggle to cope with the increasingly complex and continuously evolving attack techniques observed in real-world environments. More critically, the emergence of Automated Exploit Generation (AEG) technologies [23] has further lowered the threshold for exploitation. For instance, in the case of CVE-2025-4255 [24], attackers need only craft a small amount of code to trigger a buffer overflow vulnerability and subsequently gain control over the target server. Narrowing the scope to exploit-script detection and identification, existing research predominantly targets a single programming language and a single exploit type. However, the languages used in modern malicious scripts have become increasingly diverse, ranging from C to Python, JavaScript, PowerShell, and beyond. With the rapid advancement of generative artificial intelligence, semantic-level automatic translation of exploit scripts has become practically feasible. Using large language models (LLMs), attackers can quickly "translate" an exploit script from one programming language into another, thereby circumventing detection mechanisms designed for a single language. However, semantic code understanding or attack-intent inference via LLMs typically requires generating a large number of contextual tokens, resulting in substantial computational and inference overhead during real-world deployment. Therefore, a key challenge is how to effectively handle attack payloads written in different languages and driven by different vulnerability types, and how to develop script-based attack-type identification techniques that can operate across multiple programming languages and support diverse exploit categories. Such capabilities are essential for detecting and defending against unknown or mutated attack payloads.

In real-world environments, completely eliminating system vulnerabilities is nearly impossible. Therefore, this work shifts the defense strategy earlier by focusing on the detection of exploit scripts. By identifying and blocking malicious scripts that are language-agnostic and driven by diverse exploit types, the proposed approach aims to reduce exploitation risks before attacks occur. We propose ASTBertX, a classification model for exploit scripts that integrates two types of semantic information. The model first encodes exploit scripts using the pre-trained GraphCodeBERT to extract contextual sequence-semantic representations. It then employs the Tree-sitter toolchain to construct multilingual AST parsers and extracts key semantic nodes associated with exploit types. These AST structures are subsequently modeled using a GATv2 graph neural network to learn structural-semantic representations. Finally, the sequence-semantic and structural-semantic representations are concatenated and fused, and a temporary MLP proxy head is used during training to guide feature alignment, after which an XGBoost classifier is employed to produce the final attack-type classification results. Experimental results on the Exploit-DB [25] dataset show that the proposed method achieves 99% accuracy on the traditional binary classification task (malicious vs. benign) and 95% accuracy on the multi-class classification task constructed in this study. The method enables rapid detection and interception at the entry point before attack payloads enter the system, effectively mitigating security risks caused by delayed, missing, or ineffective patches. It provides proactive, efficient, and programming-language-agnostic pre-emptive protection for systems.

2 Related Work

In recent years, several studies have begun integrating machine learning models with malicious code detection, categorizing research targets along two dimensions: programming language type and exploit type. An overview of these studies is presented in Table 1. Existing research generally suffers from three major limitations: (1) limited language coverage, with most studies focusing on a single or a small number of scripting languages; (2) coarse-grained characterization of attack methods, as most works do not perform fine-grained attack-type classification; and (3) relatively narrow task formulations, predominantly centered on binary malicious/benign classification, with limited research on multi-class classification or semantic

analysis of attack behaviors. These limitations hinder the adaptability of existing models to real-world threat environments, where multiple programming languages and diverse exploit types coexist. To address this gap, we provide a categorized summary of recent literature based on programming languages and exploit types (Table 1).

Table 1: Research on malware classification and detection.

	Supported Target Languages	Behavioral Stage	Multi-Type Recognition Supported
Zhang et al. [10]	PowerShell	Post-intrusion	No
Alahmadi et al. [11]	PowerShell	Post-intrusion	No
Hannousse et al. [12]	WebShell	Post-intrusion	No
Feng et al. [13]	PHP	Post-intrusion	No
Pereira et al. [14]	JavaScript	Post-intrusion	No
Chenet et al. [15]	C	During-intrusion	No
Youssef et al. [16]	C	During-intrusion	No
Vu et al. [26]	Python	Post-intrusion	No
Ladisa et al. [27]	Python, JavaScript	Post-intrusion	Yes
Erdemir et al. [28]	Bash, Python, Perl	Post-intrusion	No
Ohm et al. [29]	JavaScript, Java, Python, PHP, Ruby	Post-intrusion	No

2.1 Malicious Script Detection across Different Programming Languages

Detection for PowerShell: Zhang et al. [10] performed malicious PowerShell script detection by applying de-obfuscation techniques based on AMSI memory dumps and fine-grained subtree decomposition of the AST. They further employed a two-layer neural architecture combining multi-head self attention and bidirectional GRU for feature learning. Alahmadi et al. [11] proposed a detection model based on stacked denoising autoencoders and XGBoost, achieving a 98% detection rate with only 0.6% false positives.

Detection for WebShells: Hannousse et al. [12] achieved over 98% detection accuracy for WebShells in PHP, JSP, and ASP environments using deep-learning-based methods. However, their approach remains vulnerable to sample duplication and advanced obfuscation, indicating limited robustness. Graph-based and syntax-aware methods have also emerged: Feng et al. [13] introduced a PHP WebShell detection framework leveraging word embeddings, risk-weight assignment, and graph neural networks.

Detection for JavaScript: Pereira et al. [14] employed a weighted Deterministic Finite Automaton (DFA) to dynamically model JavaScript behaviors, enabling differentiation between partially malicious and fully malicious actions. Their method demonstrated stronger detection capability in complex scenarios.

2.2 Intrusion Attack Detection across Different Vulnerability Types

Detection of Buffer-Overflow Attacks: Chenet et al. [15] proposed an anomaly-detection method based on RISC-V hardware performance counters, which monitors low-level processor events and employs machine-learning models to detect stack-based buffer overflows. Youssef et al. [16] introduced a method combining runtime instruction-trace analysis with ensemble learning. Through refined feature engineering and in-depth analysis of data-aggregation parameters, their approach achieved exceptionally high detection performance (100% recall with 0% false positives) and strong generalization capability, including the ability to detect previously unseen ROP attacks.

Detection of Code-Injection Attacks: Lo et al. [17] achieved effective SQL-injection detection by filtering noise during preprocessing, emphasizing semantic information, and employing an efficient self-category attention architecture. Their lightweight design enables deployment in resource-constrained edge environments. Gowtham and Pramod [18] proposed an SQL-injection detection model that integrates advanced

semantic-feature extraction with powerful heterogeneous ensemble learning. By leveraging Word2Vec to capture deep semantic patterns in queries and systematically addressing class imbalance, feature redundancy, and model overfitting, their method significantly improved detection robustness.

Detection of DoS-Type Attacks: Santhosh et al. [19] proposed a machine learning based DDoS detection method combining random forests with an improved XGBoost classifier, achieving 97% accuracy. Li et al. [20] introduced HDA IDS, a hybrid intrusion detection system that uses stacking ensemble learning to efficiently detect known DoS and botnet attacks. They further developed a semi supervised deep learning model integrating CNN, LSTM, and GAN to detect unknown and 0-day attacks.

Detection of File-Upload Exploitation: Wichmann et al. [21] designed a file-upload detection and sanitization system based on web framework middleware. Through multi-level validation strategies and flexible sanitization mechanisms, the system effectively mitigates most historically observed file-upload vulnerabilities while maintaining a low false-positive rate. Han et al. [22] analyzed user upload-behavior patterns rather than directly inspecting file content, and used historical statistical data to automatically generate membership functions. Their approach aims to address the challenges of rule construction, annotation dependence, and limited detection dimensions in traditional methods.

Although the above studies have advanced malicious-code research, existing methods, as summarized in Table 1, remain limited to a single scripting language, a single attack target, or a single vulnerability, and typically rely on binary malicious/benign classification. With the development of generative AI [23], APTs [30], automated threat-intelligence mining [31], and automated vulnerability discovery and exploitation [32], both attack techniques and payloads have diversified rapidly. The cost for attackers to combine multiple attack vectors and exploit multiple vulnerabilities simultaneously has significantly decreased. Consequently, existing detection approaches struggle to meet the requirements of multi-language, multi-vulnerability, and multi-platform detection. Moreover, fine-grained classification research for different exploit types remains limited, making it difficult to distinguish the characteristics of various attack categories effectively.

The main contributions of this paper are as follows: We construct a multilingual and multi-type enhanced dataset, Exploit-DB, covering multiple programming languages such as C, Python, and Perl, and annotate four major exploit categories: Overflow, Injection, DoS, and File-path attacks. The dataset has been open-sourced [33] to address the limitations of existing datasets that focus on a single language or a single exploit type. We further propose a language-agnostic exploit-script detection method that fuses sequence and structural semantic features: contextual semantic features are extracted using the pre-trained GraphCodeBERT, while language-independent ASTs are generated via tree-sitter, and structural semantics are captured using a GATv2 graph neural network. The fused representations are then fed into a XGBoost classifier to obtain the final classification results.

3 Methods

This work first employs GraphCodeBERT to extract sequence features from code, representing each script as a set of high-dimensional vectors. Meanwhile, tree-sitter is used to generate the abstract syntax tree (AST), which is then fed into an enhanced GATv2 graph neural network to extract structural features. Subsequently, a fusion strategy is designed to integrate contextual semantic features with structural semantic features, enabling multi-class detection of multilingual and multi-type malicious exploit scripts. The overall architecture of the proposed algorithm is illustrated in Fig. 2.

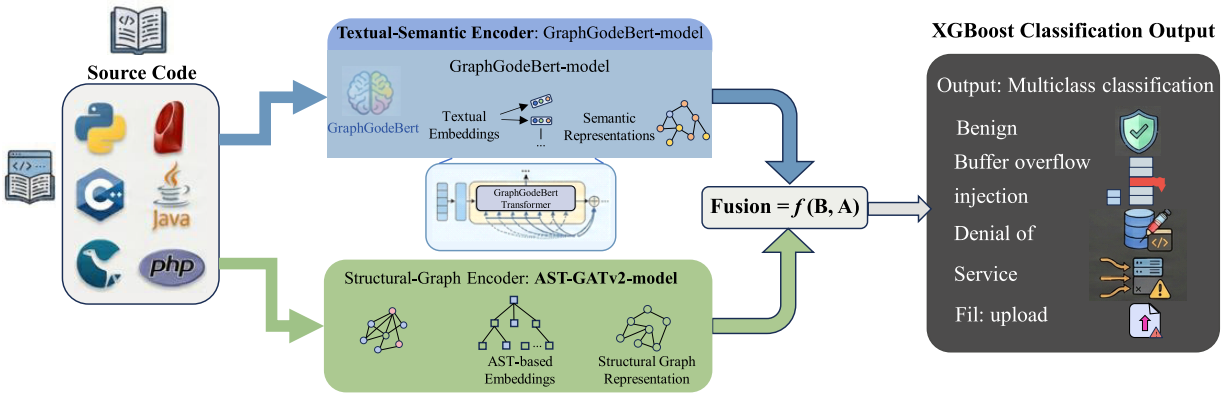

Figure 2: Overall architecture diagram of the model.

3.1 GraphCodeBERT Model Design

To extract numerical features that accurately capture the sequential semantic information of source code, this study deploys the pre-trained GraphCodeBERT model locally as a sequence semantic encoder. Note that although GraphCodeBERT supports data-flow graph features in its original design, this study utilizes only its sequence encoding capability via masked mean pooling, as structural semantics are independently captured by the AST-GATv2 branch described in Section 3.2. The module is illustrated in Fig. 3.

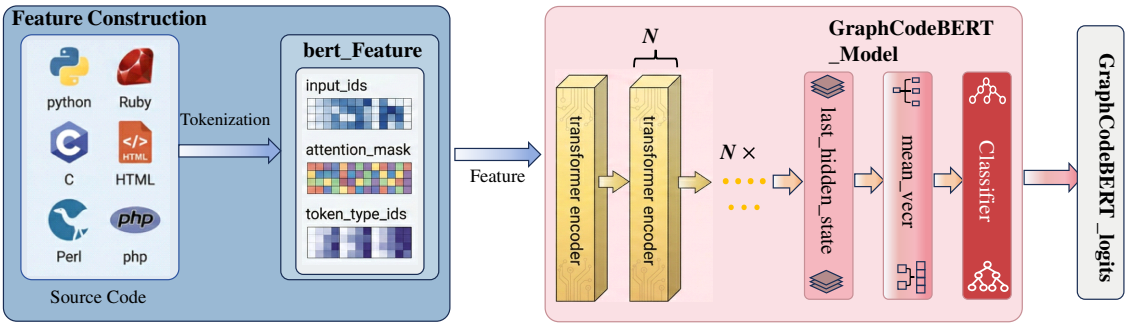

Figure 3: GraphCodeBERT model design.

First, the source code is encoded as model input. The Tokenizer provided by GraphCodeBERT converts the code text into a sequence of token IDs (*input_ids*). Since GraphCodeBERT inherits from CodeBERT, its maximum positional-encoding length is 512; tokens beyond this limit cannot be encoded. Therefore, sequences that are too long or too short are truncated or padded (with token IDs = 0), respectively, ensuring that the input length does not exceed 512 tokens.

To further validate the rationality of this truncation strategy, we conducted a comparative experiment using a sliding-window chunking approach (window size 512, stride 256, up to 8 chunks averaged) to process scripts exceeding 512 tokens. The results show that the direct truncation strategy achieves an accuracy of 91.55% and a Macro-F1 of 88.23%, slightly outperforming the chunking approach (91.26% accuracy, 85.62% Macro-F1). This indicates that the key discriminative features of exploit scripts—such as dangerous function calls, payload construction, and vulnerability-triggering logic—are predominantly concentrated in the leading portion of the code, and that averaging across chunks introduces noise from less informative

trailing segments. These results confirm that 512-token truncation is a justified lightweight design choice for this task.

Meanwhile, an attention_mask is used to distinguish valid tokens from padding tokens (1 for valid tokens, 0 for padding). The token_type_ids are used to differentiate between sentence segments; however, for code-related tasks, this sequence is defined as an all-zero tensor to remain consistent with the pretrained model configuration.

Finally, a 512-dimensional tensor is obtained and fed into the GraphCodeBERT model for inference, producing the contextual semantic representation of the source code.

The last_hidden_state output by the model after inference is a tensor of shape [batch_size, seq_len, hidden_size], where: batch_size denotes the number of samples. In this study, a single-sample feature representation is used; therefore, batch_size equals 1. seq_len refers to the initial input sequence length of 512 tokens. hidden_size represents the final semantic vector for each token, with each vector comprising 768 dimensions. Mean_vec is employed as the semantic feature input. mean_vec is obtained by averaging the hidden_state representations of all valid tokens, yielding a fixed-length vector of 768 dimensions that encodes the semantics of the entire code snippet.

$$\boldsymbol{v}_{\boldsymbol{mean}} = \frac{\sum_{i=1}^{N} m_i \cdot h_i}{\sum_{i=1}^{N} m_i} \tag{1}$$

here, h_i denotes the contextual semantic vector of the i-th token, m_i represents the attention mask, and N is the sequence length.

However, this study also evaluates different BERT-based models, and the results are summarized in Table 2. Therefore, GraphCodeBERT, which achieves the highest accuracy and F1 score, is selected as the semantic-feature extractor in this study.

Table 2: Comparison of inference performance for various models.

Model	Val Acc.	Macro-F1
BERT-Base	0.897	0.853
DistilBERT	0.895	0.849
TinyBERT	0.833	0.745
CodeBERT	0.8800	0.8400
GraphCodeBERT	0.9100	0.8649

3.2 AST-GATv2 Model Design

This work first extracts fine-grained topological features of nodes and edges from the AST to enhance code-classification performance [34]. To capture the structural semantic information of script code, the Tree-sitter toolchain is used to construct dynamic language parsers that generate ASTs for different programming languages. Each node in the AST corresponds to a syntactic element representing a fundamental program structure, such as conditional statements, loops, function calls, or variable declarations. Although AST node types vary across programming languages, their underlying syntactic semantics remain consistent. Through AST parsing, source code is uniformly mapped into a structured representation composed of syntactic nodes, enabling language-agnostic modeling. In addition to the built-in node labels provided by Tree-sitter, this study further augments node information by extracting special attributes associated with different exploit behaviors. The design of these special nodes is summarized in Table 3.

Table 3: Extraction of node attributes.

Node Attribute	Description
type	The syntactic type of the AST node, as defined by the parser (e.g., Tree-sitter).
depth	The hierarchical depth of the node within the AST, where the root node has depth 0.
contains_path_traversal_tokens	Indicates whether the node contains tokens such as ../ or ..\\, used to assist in identifying potential file-system access or path-traversal risks.
is_loop	Indicates whether the node belongs to a loop structure, used to help distinguish DoS-related attack patterns.
is_in_script_context	Script execution or command parsing context for identifying potential attacks.
is_in_script_context	Script execution or command parsing context for identifying potential attacks.

The AST extracted by Tree-sitter is defined as $G = (V, E, W)$, where V denotes the set of nodes, and E represents the set of edges. In this work, a depth-first search (DFS) is applied to traverse the AST, recursively connecting each node to its parent to form edges. W corresponds to the labels of special node attributes.

Each node type is then mapped as follows:

$$x_i = type: NodeTypes \rightarrow \{0, 1, \ldots, T-1\} \tag{2}$$

The node sequence is obtained through this mapping process:

$$x_i = [x_1, x_2, \ldots, x_n]\ T \in Z_n \tag{3}$$

Invalid edges are filtered using the node count n, which improves model efficiency and stability.

The constraints $0 \leq s_j < n$ and $0 \leq t_j < n$ ensure valid mappings, where s_j, t_j denote the mapped indices of the parent and child nodes, respectively, and j represents the current node ID. An index matrix is then constructed.

$$A = \begin{bmatrix} s_1 & t_1 \\ s_2 & t_2 \\ \ldots & \ldots \\ s_3 & t_3 \end{bmatrix} \tag{4}$$

A randomly initialized embedding matrix $E \in R^{n \times d_{emb}} (d_{emb} = 64)$ is then generated, and each node is vectorized through this embedding matrix.

$$V_i = E[x_i] row_{xi}(E)\ [e_{x_i,0}, e_{x_i,1}, \ldots e_{x_i,d_{emb-1}}] \in R^{d_{emb}} \tag{5}$$

here, $e_{x_i,d_{emb-1}}$ denotes the value of the d_{emb-1}-th dimension in the embedding vector of the i-th node x_i. All node embedding vectors are then stacked.

$$X = \begin{bmatrix} E[x_1]^T \\ E[x_2]^T \\ ..... \\ E[x_n]^T \end{bmatrix} \in R^{n \times d_{emb}} \tag{6}$$

The final feature-extraction function is defined as follows:

$$\phi(G) = (X, W, A, y) \in R^{d_t} \tag{7}$$

here, y denotes the mapped label of the exploit type, and $d_t = 67$, consisting of the 64-dimensional embedding vector plus the three special-attribute dimensions.

This study employs an improved Graph Attention Network (GATv2) to extract features from AST nodes and capture structural relationships among them. Unlike the traditional GAT architecture, GATv2 introduces a dynamic attention mechanism that overcomes the static attention limitation of standard GAT. Building on GATv2, this study further designs a class-aware multi-head attention mechanism, in which each head focuses on a different structural characteristic, to more effectively model complex node interactions that more effectively models complex interactions between nodes in the graph. The technical architecture of the AST-GATv2 module is illustrated in Fig. 4.

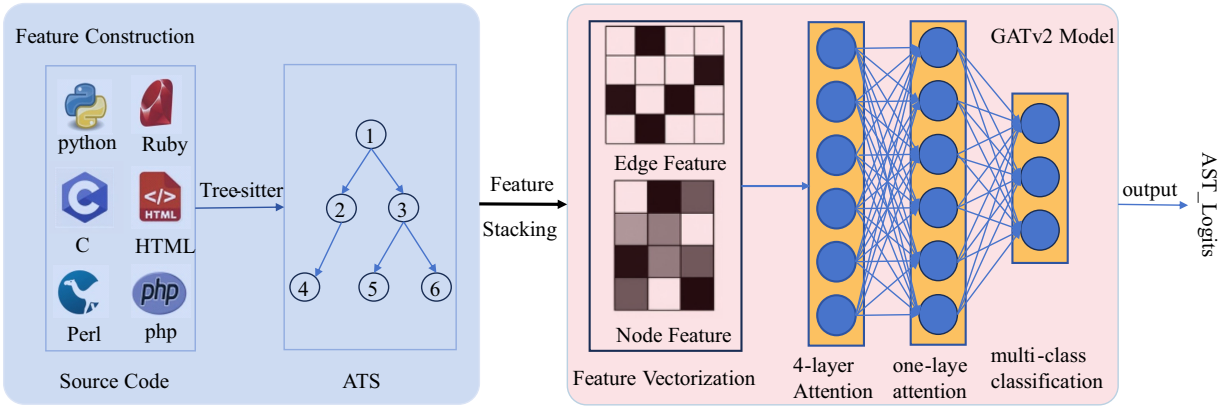

Figure 4: AST-GATv2 model design diagram.

The architecture of GATv2 consists of two stacked graph-attention layers:

The first layer adopts a four head class aware attention mechanism (heads = 4). Experimental results show that using four heads achieves a good balance between accuracy and training efficiency, effectively capturing local dependencies across different semantic subspaces. By learning node relationships in multiple neighborhood subspaces in parallel, the model obtains fine-grained representations of local syntactic information. Each class-aware attention head focuses on different structural characteristics—such as control flow, data flow, or identifier dependencies—thereby enhancing the diversity and robustness of node representations.

For each class-aware attention head $k \in \{1, 2, 3, 4\}$, the input node features are first projected into a lower-dimensional subspace through a linear transformation:

$$\tilde{h}_i^{(k)} = W^{(k)} h_i{}^{(0)} \tag{8}$$

here, $W^{(k)} \in R^{d_{out} \times d_t}$ is the learnable weight matrix of the k-th class-aware attention head, where $d_{out} = 128$, $d_t = 67$ denote the output and input dimensions, respectively. This configuration achieves a good balance between model expressiveness and computational efficiency. Given that the input dimension of GATv2 is 67, a 128-dimensional hidden representation is sufficient to capture complex syntactic and semantic patterns in code structures while avoiding overfitting caused by excessive parameters. The vector $h_i{}^{(0)}$ denotes the initial input feature of node i.

GATv2 computes the importance of neighboring nodes using a learnable class-aware attention vector:

$$\alpha_{ij(k)} = \frac{\exp(e_{il}^{(k)})}{\sum_{l\in N(i)} \exp(e_{il}^{(k)})} \tag{9}$$

Softmax normalization is applied over all neighboring nodes to obtain the class-aware attention weights, where $N(i)$ denotes the neighbor set of node i, and $\alpha_{ij(k)}$ represents the attention weight of the k-th head.

Here, σ denotes the activation function, and h_i^k represents the updated embedding of node i produced by the k-th attention head. Each head then performs weighted aggregation over the features of neighboring nodes, and the outputs of all heads are concatenated to form the final node representation, resulting in a total dimensionality of 512.

$$h_i{}^{(1)} = ||_{k=1}^{4} h_i{}^{k} \in R^{4d_{out}} \tag{10}$$

The second layer extracts higher-level global dependency information from the local features produced by the first layer. It employs a single-head class-aware attention mechanism to fuse information across structurally distant nodes, thereby avoiding unnecessary computational overhead. Compared with the first layer, the attention weights learned in the second layer cover a broader scope, enabling the model to capture global dependencies across statement blocks, across functions, or across multi-level parent–child relationships. After this layer, each node representation incorporates not only local structural information but also the semantic context of the entire graph. After the two GATv2 layers, each node obtains a context-aware feature representation. The final representation $h_i{}^{(2)}$ has a dimensionality of 128.

Finally, we apply global mean pooling over the output representations $h_i{}^{(2)}$:

$$g = \frac{1}{n}\sum_{i=1}^{n} h_i{}^{(2)} \tag{11}$$

In both convolutional layers, the ELU activation function is adopted to obtain smoother gradient behavior, enabling faster convergence during training. Moreover, Focal Loss is employed as the loss function to strengthen the model's ability to learn from minority-class samples.

3.3 Design of the XGBoost Module

The classification stage adopts a two-phase framework: in Phase 1, a temporary linear proxy head is appended to the projection layers to provide backpropagation gradients for feature alignment training; in Phase 2, the proxy head is discarded, the projection layers are frozen, and the concatenated fused vectors are fed into an XGBoost classifier for final prediction.

In the feature-fusion and classification stage, XGBoost is adopted in place of a traditional MLP classifier, forming a two-stage fusion-and-classification framework. The motivation is that the two heterogeneous feature streams, code contextual semantics and AST structural semantics, originate from different modalities. Direct concatenation followed by end-to-end classification often leads to distribution mismatch and degraded performance. By decoupling feature alignment from classification and optimizing them in separate stages, each stage obtains a clearer training objective, avoiding conflicts between multiple learning goals. The logits produced by the BERT branch and the AST graph neural network are independently projected into a unified latent space, after which they are concatenated to form a fused representation. To encourage the

projection layers to learn modality-aligned representations, the training objective combines the classification cross-entropy loss with a feature-alignment loss:

$$\mathcal{L} = \mathcal{L}_{CE} + 0.1 \times \mathcal{L}_{align} \tag{12}$$

here, $\mathcal{L}_{CE}$ denotes the cross-entropy loss of the proxy classifier, which evaluates the class discriminability of the fused representations after projection and drives the projection layers to learn semantically distinctive feature mappings. $\mathcal{L}_{align}$ is the feature-alignment loss, defined as the mean squared error between the two projected vectors after ℓ_2 normalization:

$$\mathcal{L}_{align} = \| \frac{f_{BERT}}{\| f_{BERT} \|_2} - \frac{f_{AST}}{\| f_{AST} \|_2} \|_2^2 \tag{13}$$

This loss constrains the two vectors on a normalized hypersphere, encouraging the BERT semantic modality and the AST structural modality to become consistent in the feature space, thereby alleviating distribution discrepancies between heterogeneous features. The coefficient 0.1 balances the magnitudes of the two losses, ensuring that the alignment constraint functions as an auxiliary regularization term without disrupting the optimization direction of the main classification task. A temporary linear proxy head is introduced in this stage solely to provide back-propagation gradients; it is discarded after training, and only the projection-layer parameters are retained.

The optimal projection-layer weights obtained in the first stage are loaded and frozen, after which forward inference is performed on the full dataset to extract the concatenated fused vectors as the input features for XGBoost. The main hyper parameter settings of XGBoost are summarized in Table 4.

Table 4: Hyper parameter settings.

Hyperparameter	Value	Description
n_estimators	300	Number of boosted trees
max_depth	6	Maximum depth of each tree
learning_rate	0.1	Learning rate (shrinkage factor)
subsample	0.8	Row sampling ratio for each tree
colsample_bytree	0.8	Feature sampling ratio for each tree
eval_metric	mlogloss	Multiclass log-loss

Compared with an MLP classifier head, XGBoost does not require back propagation, is inherently robust to high-dimensional sparse features, and avoids gradient-based hyperparameter tuning, resulting in a more stable training process. The core advantage of the two-stage design lies in the fact that the projection-layer pre-training stage fully leverages the alignment loss to regularize the feature distributions of the two modalities, while the XGBoost classification stage exploits the strong discriminative power of ensemble learning to perform the final decision-making. This achieves a decoupled optimization of feature alignment and classification capability.

4 Experiments

4.1 Dataset

We first describe the construction of the Exploit-DB-EX dataset. The Exploit-DB dataset [25] contains 46,921 exploit scripts written in multiple programming languages and targeting various platforms. The

description field of each entry is written by security researchers at submission time and reviewed by the Exploit-DB professional team, ensuring high labeling reliability. In this work, the description field provided by Exploit-DB is used to categorize malicious samples, ensuring high labeling reliability. By extracting characteristic keywords (e.g., Overflow, Injection, DoS, File path), the samples are grouped into categories, and the resulting distribution is visualized as shown in Fig. 5.

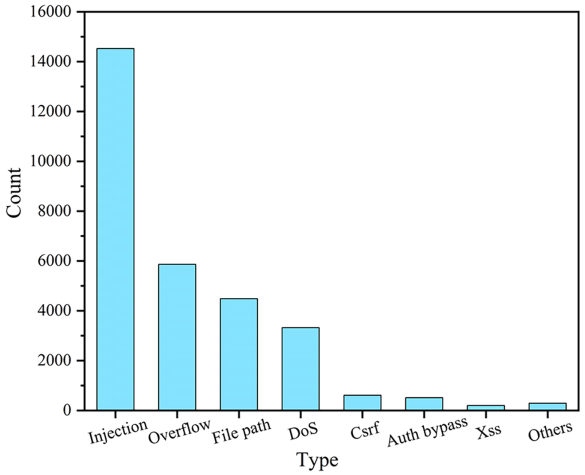

Figure 5: Number of vulnerability exploit types.

Among all samples, the four categories, Overflow, Injection, DoS, and File path, account for approximately 90%. Given the dataset scale, this work focuses on these four attack types together with benign scripts for algorithm design and evaluation. Based on the vulnerability-related keywords commonly appearing in the description field, the exploit descriptions are categorized into ten high-risk vulnerability types, resulting in a total of 29,806 samples. To ensure sufficient sample size, only the four major categories, Overflow, Injection, DoS, and File path, are retained, yielding 27,747 exploit samples. After removing .txt files based on file extensions and performing de-duplication, a total of 9161 valid malicious script samples are obtained.

As shown in Fig. 6, the distribution of the ten most popular programming languages is presented; in this study, the six languages with the highest frequency of occurrence—Python, Perl, C, Ruby, HTML, and PHP—totaling 8499 samples, were selected to serve as the basis for classification analysis.

Since the dataset originates from the public Exploit-DB exploit repository, whose scripts are designed for security researchers with an emphasis on readability and reproducibility, the samples generally do not contain obfuscated malicious code. Additionally, 5000 benign code samples were obtained from Hugging Face [35], consisting of 1000 samples each from five languages: Python, C, Ruby, HTML, and PHP. These benign samples include functionalities such as uploading, downloading, connecting, and storing—operations similar to those in exploit scenarios but without malicious intent. They are incorporated to enhance the model's ability to distinguish functionally similar code and prevent reliance on superficial functional cues. After obtaining the initial dataset, hash-based de-duplication is first performed, followed by language identification based on file extensions. Noise is further removed by stripping single-line and multi-line comments according to the common commenting conventions of each language (e.g., Python uses # or triple quotes \"\"\" ... \"\"\"). It is worth noting that Exploit-DB enforces a strict submission review process conducted by a professional security team, which rejects duplicate and derivative submissions at the entry stage. This mechanism inherently guarantees the uniqueness and independence of samples in the database,

making hash-based deduplication a supplementary rather than primary quality-control measure. This results in a final dataset of 13,499 samples, including both malicious and benign code.

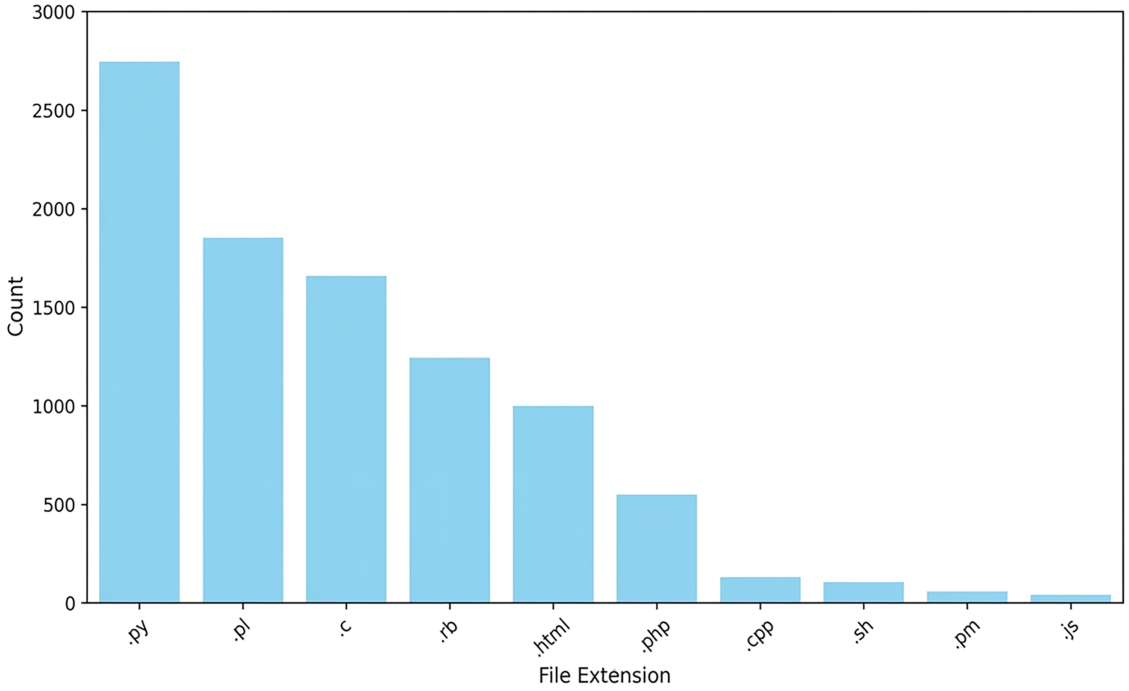

Figure 6: Number of exploit scripts in different languages.

4.2 Experimental Setup

The experiments are conducted on a system equipped with an NVIDIA GeForce RTX 4060 GPU and an Intel Core i7-14650HX CPU running Ubuntu 20.04. The Exploit-DB-EX dataset is used, and after preprocessing, it is split into training, validation, and test sets with a ratio of approximately 70%/10%/20% (9449 training samples, 1350 validation samples, and 2700 test samples), with the three subsets being strictly disjoint. Early stopping is determined solely based on validation set accuracy, with training halted if no improvement is observed for 10 consecutive epochs. The test set is completely isolated throughout the entire training process and is used exclusively for final performance reporting. The model achieving the highest validation accuracy is selected as the final model.

4.3 Experimental Results and Analysis

The performance of the proposed method is evaluated on the validation set, achieving an overall accuracy of 95.07%, with both recall and F1-score exceeding 99%. The classification metrics for different exploit types are summarized in Table 5.

For the Benign class, the precision, recall, and F1-score reach 99.38%, 99.20%, and 99.29%, respectively, indicating that the model performs exceptionally well in distinguishing exploit scripts from normal code and is capable of intercepting malicious scripts in real-world deployment scenarios. The Injection and Overflow classes achieve F1-scores of 93.46% and 94.84%, respectively, reflecting strong overall performance. The DoS and File path classes obtain F1-scores of 83.99% and 87.88%, which are relatively lower. The reasons are analyzed as follows. As shown in the confusion matrix, most misclassifications of the DoS class are toward Overflow, and 31 Overflow samples are also misclassified as DoS (Fig. 7). These two

attack types share intrinsic similarities at the AST structural level: loop constructs and memory-related operations frequently appear in both. Since the current GATv2 relies primarily on node types and topological relationships, it struggles to fully distinguish the deeper semantic difference between "loops used for resource exhaustion" and "loops used for buffer filling." This is the main reason for the mutual confusion between DoS and Overflow.

Table 5: Classification metrics for different vulnerability types.

	Benign (%)	DoS (%)	File Path (%)	Injection (%)	Overflow (%)
Precision	99.38	83.99	86.14	92.40	95.26
Recall	99.20	83.99	89.69	94.55	94.43
F1-score	99.29	83.99	87.88	93.46	94.84

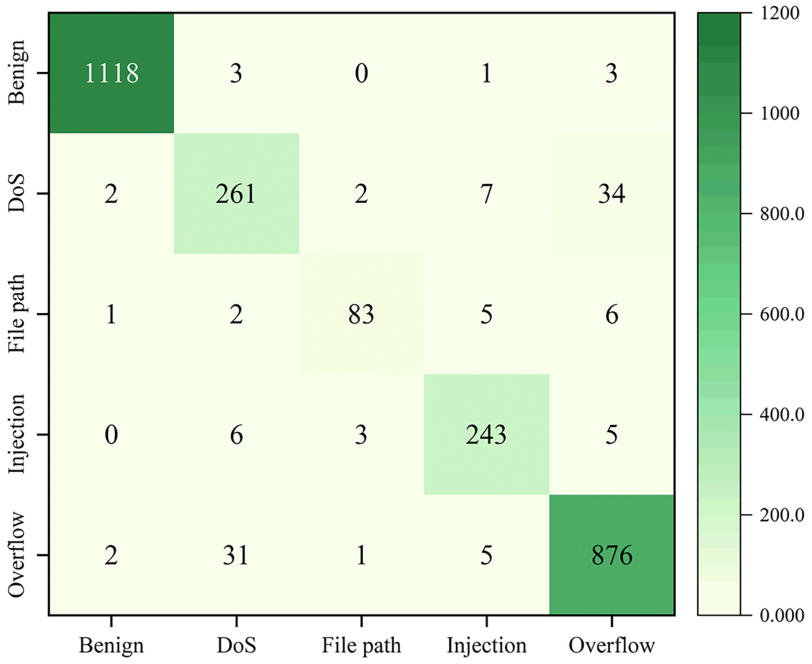

Figure 7: Confusion matrix of different classes.

The cross-language and cross-category accuracy statistics are shown in Fig. 8. From the programming-language perspective, the overall accuracy for Python, C, Ruby, Perl, HTML, and PHP remains above 94%. And from the vulnerability-type perspective, the Benign class achieves over 98% accuracy across all six languages (global accuracy: 98.72%), indicating that benign samples exhibit highly distinguishable semantic and structural characteristics across multilingual settings. The File path (85.57%) and DoS (84.38%) classes show relatively lower accuracy. For File path, the Ruby subset contains only five samples (n = 5), and the limited sample size may introduce fluctuations in the evaluation results for this language-category combination. The misclassification of Overflow in PHP is related to PHP's memory-management mechanism: PHP's stack operations are abstracted by Zend engine, and the AST generated by Tree-sitter lacks low-level memory-operation nodes equivalent to those found in C-language Overflow samples. As a result, the Overflow structural patterns learned from C samples transfer poorly to PHP. Nevertheless, PHP still achieves an overall accuracy of 97.0%, which remains within a high-accuracy range.

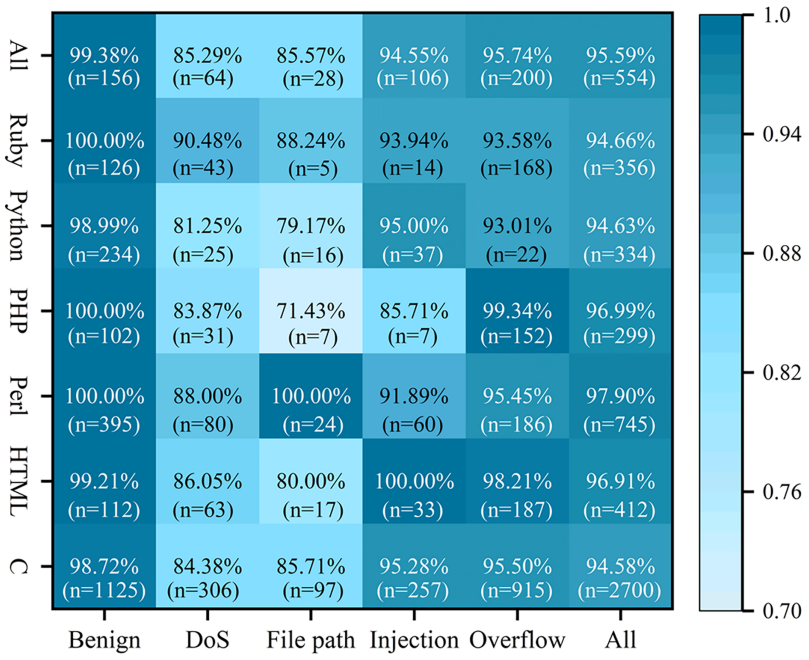

Figure 8: Accuracy statistics across different languages and categories.

4.4 Ablation Study

In this study, AST-GATv2 and GraphCodeBERT are used to conduct ablation experiments, and the resulting performance comparisons are summarized in Table 6.

Table 6: Various indicators for each model.

	Accuracy (%)	Macro F1 (%)	Weighted F1 (%)
AST-GATv2	82.85	71.52	82.44
GraphCodeBERT	91.00	86.00	90.00
All	95.07	91.89	95.08

To verify the effectiveness of each component of the model, an ablation study was conducted. The results show that AST-GATv2 exhibits relatively low performance among the single-modality models, while GraphCodeBERT achieves higher accuracy. The full model outperforms both individual models, indicating that sequence-semantic features significantly enhance the model's ability to understand code. The performance of the single-modality models and the full model across different categories is summarized as Fig. 9.

In the precision comparison across different models, the All model achieves the best performance for all vulnerability types, significantly outperforming both GraphCodeBERT and AST-GATv2. Specifically, the All model reaches a precision of 0.99 for the Benign class and improves the DoS precision by approximately 14% compared with AST-GATv2. GraphCodeBERT shows moderate overall performance, reflecting its strong capability in semantic understanding. In contrast, AST-GATv2 exhibits lower precision for the Injection and DoS classes, indicating the limitations of relying solely on structural features. Overall, the fused model demonstrates superior detection capability compared with single-modality models, validating the effectiveness of multimodal fusion in exploit detection.

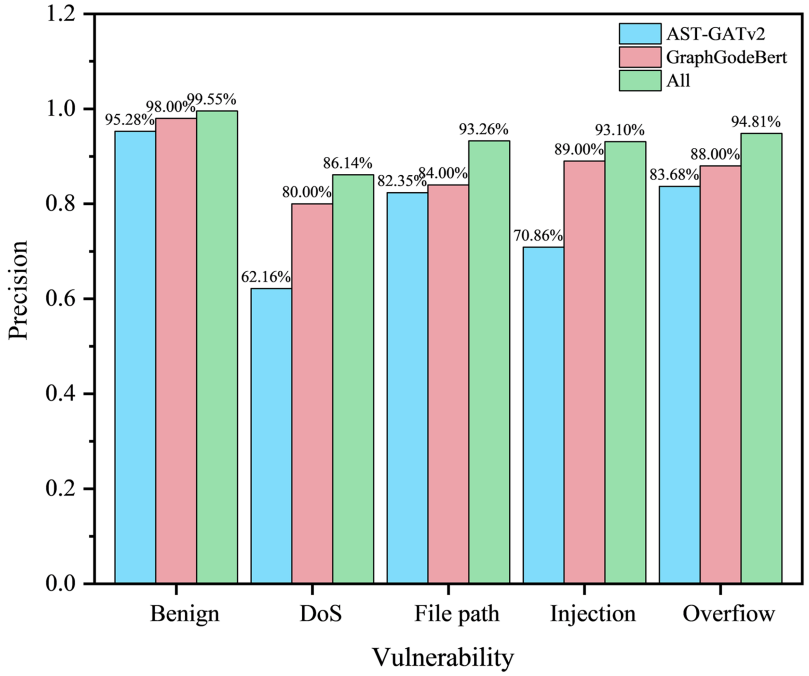

Figure 9: Accuracy comparison of single models.

It should be noted that the relatively lower standalone performance of AST-GATv2 (82.85%) compared to GraphCodeBERT (91.00%) does not imply a limited contribution of structural features. This performance gap primarily reflects the disparity in pre-training resources: GraphCodeBERT benefits from large-scale pre-training on billions of code tokens, whereas AST-GATv2 employs randomly initialized 64-dimensional node embeddings trained from scratch on limited task data. The substantial performance gain achieved by the fused model (95.07%) over GraphCodeBERT alone (91.00%)—particularly in the DoS and File path categories where structural patterns such as loop constructs and path-traversal tokens are most discriminative—demonstrates that the AST branch contributes genuine structural information rather than marginal regularization.

4.5 Comparative Experiments

We focus on fine-grained vulnerability-type identification for multilingual exploit scripts, the task is highly specialized, making it difficult to find directly comparable baseline methods in existing research. The reasons are as follows: (1) most existing malicious-script detection approaches support only a single programming language—such as PowerShell, PHP, or JavaScript—and therefore cannot handle the six languages considered in this study (Python, C, Perl, Ruby, PHP, and HTML). (2) Research specifically targeting fine-grained exploit-type classification remains largely unexplored.

Considering these factors, this study adopts the multilingual cross-language malicious-package detection method proposed by Ladisa et al. [27] as the baseline. This method introduces a language-agnostic static feature system and uses XGBoost as the classifier, making it one of the few multilingual detection approaches with both multi-language support and open-source availability. To ensure fair comparison, the baseline method is evaluated on the Exploit-DB dataset under both binary and five-class settings, using the same evaluation metrics as this work—accuracy and macro-averaged F1-score. The experimental results are presented in Table 7.

Table 7: Comparison of experimental results.

Task	Method	Accuracy	Macro F1
Binary classification	Ladisa et al. [27]	98.44%	98.77%
	ASTBertX	99.38%	99.29%
Five-class classification	Ladisa et al. [27]	90.08%	84.98%
	ASTBertX	95.07%	91.89%

As shown in Table 7, the proposed method outperforms the baseline in both the binary and five-class classification tasks. In the binary classification task, the proposed method achieves an accuracy of 99.38% and a macro F1-score of 99.29%, representing improvements of 0.94 and 0.52 percentage points over the method of Ladisa et al. [27]. This indicates that the model is nearly perfect in distinguishing malicious scripts from benign ones, demonstrating strong potential for real-world deployment. In the five-class task, the proposed method achieves an accuracy of 95.07% and a macro F1-score of 91.89%, improving upon the baseline by 4.99 and 6.91 percentage points, respectively—an even more substantial gain. This demonstrates that the dual-path fusion architecture combining GraphCodeBERT's sequence semantics and GATv2's structural semantics offers clear advantages for fine-grained exploit-type classification. The larger performance gain in the five-class task further confirms that the proposed fusion strategy effectively distinguishes fine-grained category boundaries, rather than merely improving binary discrimination. Notably, the method of Ladisa et al. [27] exhibits substantial variation in per-class performance in the five-class setting.

We compare the baseline algorithm (Ladisa et al. [27]) with our approach in Table 8. To make it more intuitive, the data in Table 5 are also presented here. As shown in this table, the baselint algorihtm method achieves an F1-score of only 77.89% on the DoS class and a recall of merely 63.55% on the File path class, reflecting considerable difficulty in distinguishing exploit types with similar semantic characteristics. In contrast, the proposed ASTBertX method attains F1-scores of 83.99% and 87.88% for these two classes, respectively, representing improvements of approximately 6 and 15 percentage points. Furthermore, the Ladisa's method also lags behind on the Injection class, where its F1-score of 86.11% falls noticeably short of ASTBertX's 93.46%. In contrast, the proposed dual-modality approach, integrating sequence semantics with structural semantics derived from abstract syntax trees, demonstrates clear and consistent advantages across all fine-grained vulnerability classes, highlighting the importance of incorporating both dimensions of code representation for accurate exploit-type classification.

Table 8: The comparisons with the baseline algorithm.

Model		Benign (%)	DoS (%)	File Path (%)	Injection (%)	Overflow (%)
ASTBertX	Precision	99.38	83.99	86.14	92.40	95.26
	Recall	99.20	83.99	89.69	94.55	94.43
	F1-score	99.29	83.99	87.88	93.46	94.84
Ladisa	Precision	97.16	78.12	85.43	83.60	90.37
	Recall	98.45	77.65	63.55	88.77	90.12
	F1-score	97.80	77.89	72.88	86.11	90.25

In summary, the proposed method maintains comparable binary-classification performance to the baseline, while demonstrating substantially stronger discriminative capability in fine-grained exploit-type

identification. This validates the effectiveness of the sequence–structure fusion strategy and the category-gated attention mechanism.

4.6 Robustness Evaluation

To assess the robustness of the model in complex adversarial environments, we conducted experiments using the Power-ASTNN dataset [10], which contains obfuscated malicious PowerShell scripts. This dataset comprises a substantial volume of obfuscated malicious scripts, effectively simulating real-world code obfuscation and variant attacks. In this study, we evaluated only the binary classification task (malicious vs. benign) and compared the results against those of the baseline methods: Power-ASTNN [10] and the model proposed by Ladisa et al. [27]. The experimental results are presented in Table 9.

Table 9: Comparison table of accuracy rates on confused datasets.

Method	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
ASTBertX	99.17	99.31	98.89	99.09
Power-ASTNN [10]	98.87	98.63	99.20	98.91
Ladisa et al. [27]	98.72	98.84	98.39	98.61

The experimental results demonstrate that the proposed method outperforms the baseline model, Zhang et al. [10], across all metrics, including accuracy, precision, and F1-score. It falls slightly short only in terms of recall, though the difference is negligible. Overall, the proposed method maintains stable classification performance, even under data obfuscation scenarios. Furthermore, when compared against the method proposed by Ladisa et al. [27], it surpasses the baseline across all evaluated metrics. This conclusively demonstrates its robust resilience and strong generalization capabilities in the face of code obfuscation attacks.

5 Conclusion

This paper proposes a vulnerability-exploit script classification method based on multi-modal fusion. By integrating sequence-semantic features extracted by GraphCodeBERT with structural-semantic features derived from ASTs via GATv2, and further applying category-gated and category-attention mechanisms for adaptive fusion, the method achieves multilingual and fine-grained exploit-type identification. Experiments conducted on a multilingual dataset constructed from Exploit-DB demonstrate that the proposed method achieves 95.07% accuracy in the five-class task and 99% accuracy in the binary task, significantly outperforming baseline approaches.

Beyond the core detection task, ASTBertX demonstrates practical potential for integration into real-world software development workflows. The XGBoost classifier operates at millisecond-level inference latency, making it feasible to integrate into AI code copilot tools (e.g., GitHub Copilot, Cursor) as a human-in-the-loop security screening layer, where the model issues real-time vulnerability type warnings while the developer retains final authority. At the pipeline level, ASTBertX supports a three-layer defense strategy: semantic-enhanced AST extraction during static analysis, a CI/CD pre-commit hook to intercept malicious commits before they enter the repository, and periodic batch scanning of legacy codebases and third-party libraries. To support real-world deployment, we have released a production-ready toolchain and an Agent Skill providing a step-by-step operational guide, publicly available at: https://github.com/cxl1314520/ASTBertX. Despite the encouraging results, it is essential to acknowledge the inherent limitations of static analysis: it cannot detect runtime behaviors triggered only under specific conditions, such as multi-stage

payloads or environment-aware obfuscation techniques. To overcome this limitation, we propose a hybrid defense scheme that combines ASTBertX's static pre-screening with dynamic execution in a sandbox for high-risk predictions. Suspicious code snippets flagged by ASTBertX with high confidence can undergo dynamic testing in an isolated sandbox environment before being integrated into the production codebase, thereby establishing a more robust defense chain.

Despite the promising results, several limitations remain that warrant further investigation. First, the dataset does not achieve full language coverage: Perl lacks benign counterparts, leading to relatively lower detection performance for underrepresented languages, and emerging languages such as Go and Rust are not yet included—an issue rooted in imbalanced pre-training corpora and limitations in Tree-sitter's parsing capability. Second, the model's generalization ability to zero-day exploit scenarios has not been fully validated, as the training data consist primarily of historical public samples from Exploit-DB, potentially limiting adaptability to novel exploitation patterns in dynamic real-world environments. Third, robustness against adversarial manipulation remains insufficient: while binary classification was evaluated on an obfuscated PowerShell dataset, systematic cross-language adversarial evaluation and defense design have not yet been conducted. Finally, the dataset size and class distribution impose further constraints, as some exploit types contain limited samples and broader attack scenarios such as logic flaws and privilege escalation are not yet covered. To address these limitations, future research will proceed along four directions: (1) enhancing low-resource language modeling through continual pre-training, next-generation code models, and improved parsing tools; (2) developing incremental and few-shot learning frameworks for rapid adaptation to zero-day vulnerabilities; (3) improving adversarial robustness via diverse data-augmentation strategies and contrastive-learning methods; and (4) enriching program-structure representations by incorporating data-flow graphs and program-dependency graphs, with discriminative loss functions to mitigate inter-class confusion.

Acknowledgement: Not applicable.

Funding Statement: We gratefully acknowledge the support from the Science and Technology Projects of the Ministry of Public Security of China (No. 2022JSM04); Sichuan Science and Technology Program (No. 2024NSFSC2045, 2025NSFSC2086); the Science and Technology Plan Project of Luzhou City (No. 2023JYJ020) and the Project of Police Integration Computing Key Laboratory of Sichuan Province (No. JWRH202504003).

Author Contributions: Xinglong Cao: Designed the methodology, performed coding and software implementation, and wrote the initial draft of the manuscript. Cong Wang: Reviewed the methodology, supervised the research, and managed project administration. Jie Yan: Performed data cleaning and data curation. Songcan Yu: Conducted software testing, debugging, and validation of results. Mingze He: Assisted in methodology design, validated results, participated in manuscript review and revision, and assisted in coding. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The benchmark datasets used in this study are publicly available: Exploit-DB-EX (https://huggingface.co/datasets/wwe123/Exploit-DB-EX), Power-ASTNN (https://github.com/Juncheng-Lu/PowerASTNN/tree/main/dataset_ASTNN), CodeParrot (https://huggingface.co/datasets/codeparrot/github-code). The experimental code has been published at (https://github.com/cxl1314520/ASTBertX).

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Greenberg A. CyberAv3ngers: the Iranian saboteurs hacking water and gas systems worldwide [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://www.wired.com/story/cyberav3ngers-iran-hacking-water-and-gas-industrial-systems/.
2. HackRead. UnitedHealth Group's data breach impacts 190 million Americans [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://hackread.com/unitedhealth-groups-data-breach-impacts-americans/.
3. IARM Information Security. The top IoT security incidents of 2024: insights & lessons [Internet]. 2024 [cited 2026 Apr 27]. Available from: https://iarminfo.com/the-top-iot-security-incidents-of-2024/.
4. Infosecurity Magazine Staff. Unpatched CCTV cameras exploited to spread Mirai variant [Internet]. 2024 [cited 2026 Apr 27]. Available from: https://www.infosecurity-magazine.com/news/unpatched-cctv-cameras-exploited/.
5. Arghire I. Google sues operators of 10-million-device Badbox 2.0 botnet [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://www.securityweek.com/google-sues-operators-of-10-million-device-badbox-2-0-botnet/.
6. Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 2019); 2019 Dec 8–14; Vancouver, BC, Canada. Red Hook, NY, USA: Curran Associates Inc.; 2019. p. 10197–207.
7. Sejfia A, Das S, Shafiq S, Medvidović N. Toward improved deep learning-based vulnerability detection. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering; 2024 Apr 14–20; Lisbon Portugal. New York, NY, USA: ACM; 2024. p. 1–12. doi:10.1145/3597503.3608141.
8. Song Z, Xu J, Li K, Shan Z. HCRVD: a vulnerability detection system based on CST-PDG hierarchical code representation learning. Comput Mater Contin. 2024;79(3):4573–601. doi:10.32604/cmc.2024.049310.
9. Ciavatta JAS, Higuera JRB, Higuera JB, Montalvo JAS, Riera TS, Melero JP. Integration of large language models (LLMs) and static analysis for improving the efficacy of security vulnerability detection in source code. Comput Mater Contin. 2026;86(3):11. doi:10.32604/cmc.2025.074566.
10. Zhang S, Li S, Lu J, Yang W. Power-ASTNN: a deobfuscation and AST neural network enabled effective detection method for malicious PowerShell scripts. Comput Secur. 2025;154(2):104441. doi:10.1016/j.cose.2025.104441.
11. Alahmadi A, Alkhraan N, BinSaeedan W. MPSAutodetect: a malicious powershell script detection model based on stacked denoising auto-encoder. Comput Secur. 2022;116(11):102658. doi:10.1016/j.cose.2022.102658.
12. Hannousse A, Nait-Hamoud MC, Yahiouche S. A deep learner model for multi-language webshell detection. Int J Inf Secur. 2023;22(1):47–61. doi:10.1007/s10207-022-00615-5.
13. Feng P, Wei D, Li Q, Wang Q, Hu Y, Xi N, et al. GlareShell: graph learning-based PHP webshell detection for web server of industrial internet. Comput Netw. 2024;245(1):110406. doi:10.1016/j.comnet.2024.110406.
14. Pereira P, Gonçalves J, Vitorino J, Maia E, Praça I. Enhancing JavaScript malware detection through weighted behavioral DFAs. In: Cybersecurity. Cham, Switzerland: Springer Nature; 2025. p. 201–14. doi:10.1007/978-3-031-94855-8_13.
15. Chenet CP, Savino A, Di Carlo S. Zero-day hardware-supported malware detection of stack buffer overflow attacks: an application exploiting the CV32e40p RISC-V core. In: Proceedings of the 2025 IEEE 26th Latin American Test Symposium (LATS); 2025 Mar 11–14; San Andres Islas, Colombia. Piscataway, NJ, USA: IEEE; 2025. p. 1–6. doi:10.1109/lats65346.2025.10963939.
16. Youssef A, Abdelrazek M, Karmakar C. Use of ensemble learning to detect buffer overflow exploitation. IEEE Access. 2023;11:52009–25. doi:10.21227/q4zc-td75.
17. Lo RT, Hwang WJ, Tai TM. SQL injection detection based on lightweight multi-head self-attention. Appl Sci. 2025;15(2):571. doi:10.3390/app15020571.
18. Gowtham M, Pramod HB. Semantic query-featured ensemble learning model for SQL-injection attack detection in IoT-ecosystems. IEEE Trans Reliab. 2022;71(2):1057–74. doi:10.1109/TR.2021.3124331.
19. Santhosh S, Sambath M, Thangakumar J. Detection of DDOS attack using machine learning models. In: Proceedings of the 2023 International Conference on Networking and Communications (ICNWC); 2023 Apr 5–6; Chennai, India. Piscataway, NJ, USA: IEEE; 2023. p. 1–6. doi:10.1109/ICNWC57852.2023.10127537.

20. Li S, Cao Y, Liu S, Lai Y, Zhu Y, Ahmad N. HDA-IDS: a hybrid DoS attacks intrusion detection system for IoT by using semi-supervised CL-GAN. Expert Syst Appl. 2024;238(15):122198. doi:10.1016/j.eswa.2023.122198.
21. Wichmann P, Groddeck A, Federrath H. FileUploadChecker: detecting and sanitizing malicious file uploads in web applications at the request level. In: Proceedings of the 17th International Conference on Availability, Reliability and Security; 2022 Aug 23–26; Vienna, Austria. New York, NY, USA: ACM; 2022. p. 1–10. doi:10.1145/3538969.3538999.
22. Han T, Zhan X, Tao J, Cao K, Xiong Y. Anomaly upload behavior detection based on fuzzy inference. In: Proceedings of the 2021 IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC/PiCom/CBDCom/CyberSciTech); 2021 Oct 25–28; Online. Piscataway, NJ, USA: IEEE; 2021. p. 923–9. doi:10.1109/dasc-picom-cbdcom-cyberscitech52372.2021.00154.
23. Bui QC, Iannone E, Camporese M, Hinrichs T, Tony C, Tóth L, et al. A systematic literature review on automated exploit and security test generation. arXiv:2502.04953. 2025.
24. National Institute of Standards and Technology. CVE-2025-4255: buffer overflow vulnerability in RMD command handler of PCMan FTP server 2.0.7 [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2025-4255.
25. Offensive Security Limited. Exploit database—exploits for penetration testers, researchers and ethical hackers [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://www.exploit-db.com/.
26. Vu DL, Newman Z, Meyers JS. Bad snakes: understanding and improving Python package index malware scanning. In: Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE); 2023 May 14–20; Melbourne, VIC, Australia. Piscataway, NJ, USA: IEEE; 2023. p. 499–511. doi:10.1109/ICSE48619.2023.00052.
27. Ladisa P, Ponta SE, Ronzoni N, Martinez M, Barais O. On the feasibility of cross-language detection of malicious packages in npm and PyPI. In: Proceedings of the 39th Annual Computer Security Applications Conference; 2023 Dec 4–8; Austin, TX, USA. New York, NY, USA: ACM; 2023. p. 71–82. doi:10.1145/3627106.3627138.
28. Erdemir E, Park K, Morais MJ, Gao VR, Marschalek M, Fan Y. SCORE: syntactic code representations for static script malware detection. arXiv:2411.08182. 2024.
29. Ohm M, Plate H, Sykosch A, Meier M. Backstabber's knife collection: a review of open source software supply chain attacks. In: Detection of intrusions and malware, and vulnerability assessment. Cham, Switzerland: Springer International Publishing; 2020. p. 23–43. doi:10.1007/978-3-030-52683-2_2.
30. Kaspersky ICS CERT. APT and financial attacks on industrial organizations in Q4 2024 [Internet]. 2025 [cited 2026 Apr 27]. Available from: https://ics-cert.kaspersky.com/publications/reports/2025/03/25/apt-and-financial-attacks-on-industrial-organizations-in-q4-2024.
31. IBM Security. X-Force 2026 threat intelligence index. Armonk (NY): IBM Corporation; 2026 [cited 2026 Jul 27]. Available from: https://www.ibm.com/reports/threat-intelligence.
32. Luo Z, Du Q, Wang Y, Roychoudhury A, Jiang Y. Enhancing protocol fuzzing via diverse seed corpus generation. IEEE Trans Softw Eng. 2025;51(9):2693–709. doi:10.1109/TSE.2025.3595396.
33. Exploit-DB-EX dataset [Internet]. 2024 [cited 2026 Apr 27]. Available from: https://huggingface.co/datasets/wwe123/Exploit-DB-EX.
34. Sun W, Fang C, Miao Y, You Y, Yuan M, Chen Y, et al. Abstract syntax tree for programming language understanding and representation: how far are we? arXiv:2312.00413. 2023.
35. CodeParrot GitHub code dataset [Internet]. 2021 [cited 2026 Apr 27]. Available from: https://huggingface.co/datasets/Makoveli89/codeparrot-github-code.