# A Software Code Defect Detection Method Based on the Fusion of Transformer and Graph Neural Networks

**Zifu Wang[1], Aoxuan Ding[1], Chang Ma[1], Heng Chang[2], Jun Tang[1], Yujie Wu[1], Guangle Yao[1,*]**
[1]College of Computer Science and Cyber Security (Model Software College), Chengdu University of Technology, Chengdu, 610059, China
[2]College of Mechanical and Electrical Engineering, Beijing Information Science & Technology University, Beijing, 102206, China

Corresponding author: Guangle Yao (e-mail: guangle.yao@cdut.edu.cn).

**ABSTRACT** Software defect detection plays a key role in improving system reliability, security, and maintainability. This study proposes a hybrid Transformer–GNN framework to jointly model code semantics and program structure. Source files are normalized, segmented at the function level, and converted into token sequences, while multi-relation graphs are generated to describe syntax, control flow, and data dependencies. The Transformer branch captures long-range contextual information from code tokens, whereas the GNN branch learns structural interactions among statements, execution paths, and variables. Their representations are adaptively integrated through a gated fusion module for defect classification. Experiments show that the proposed method outperforms conventional classifiers, sequence-oriented neural networks, and standalone Transformer or GNN models. Ablation results also verify the importance of semantic encoding, structural learning, data-flow information, and gated fusion. Overall, combining lexical context with graph-based program relations provides a more accurate, robust, and interpretable solution for software defect identification.

**INDEX TERMS** Defect prediction, Code representation learning, Semantic–structural fusion, Program dependency analysis.

## I. INTRODUCTION

The reliability, security, and sustainable operation of increasingly sophisticated software applications are often compromised by latent defects in their source code. In application domains such as financial services, intelligent manufacturing, cloud computing, the Internet of Things, and embedded control systems, defects in source code may cause program failures, service outages, data breaches, or serious security risks. Therefore, identifying potential defects accurately and efficiently during software development and maintenance is of great significance for software engineering and cybersecurity. Existing approaches, including manual code inspection and static analysis tools, can detect certain types of defects, but they often depend on expert experience, predefined rules, and fixed program patterns. When faced with complex logic, diverse defect forms, and cross-project code differences, these methods may suffer from high false alarm rates, missed detections, and weak generalization capability [1,2]. Recent advances in neural representation learning have made it possible to identify defective code patterns directly from program data with limited reliance on manually engineered features. Compared with traditional algorithms, deep neural models reduce the reliance on handcrafted features and can extract latent representations from code data. Earlier studies mainly used convolutional neural networks, recurrent neural networks, and long short-term memory networks to learn sequential code features, and these methods achieved encouraging detection results. More recently, with the development of pre-trained code representation models, Transformer-based approaches have shown stronger advantages in modeling contextual semantic relationships and long-range dependencies among code tokens [3–5]. However, source code is not only a sequence of tokens. It also contains rich structural information, including syntactic organization, control flow, data flow, and variable dependencies. Purely sequence-based Transformer modeling may therefore be insufficient for representing program execution logic and structural dependency patterns. Graph neural networks provide an effective way to model programs through abstract syntax trees, control flow graphs, data flow graphs, or code property graphs. Nevertheless, graph-based models used alone may still be constrained by limited node-level semantic representation and insufficient global contextual modeling ability [6–9]. To address these issues, a hybrid neural architecture is introduced for identifying defective code. Function-level program units are converted into normalized token sequences and multi-relational graphs that encode syntax, execution paths, and variable interactions. A Transformer encoder is then adopted to learn contextual semantic representations from code sequences, while a graph neural network captures structural

dependency information from the corresponding program graphs. On this basis, a semantic–structural fusion mechanism is designed to combine the two types of representations for defect classification. The main purpose of this framework is to make full use of the global semantic modeling strength of Transformer and the structural representation ability of graph neural networks, thereby generating a more comprehensive and robust code representation. Comparative experiments with traditional machine learning methods, sequence-based deep learning models, standalone Transformer models, and standalone graph neural network models are conducted to verify the effectiveness of the proposed approach in software code defect detection.

## II. RESEARCH BACKGROUND AND METHODOLOGICAL ADVANCES

### A. CONVENTIONAL APPROACHES TO DEFECT IDENTIFICATION

Conventional defect detection includes code review, static analysis, dynamic testing, symbolic execution, and metric-based prediction. Although these techniques offer practical value, they are often constrained by expert dependence, incomplete test coverage, path explosion, high computational cost, and frequent false alarms. Traditional classifiers such as SVM, decision tree, random forest, and logistic regression further improve automated prediction by using software metrics, including code size, complexity, coupling, change frequency, and defect history. However, such statistical features rarely capture program semantics or dependency relationships, resulting in limited generalization and coarse-grained detection in complex codebases.

### B. DEEP LEARNING-BASED CODE DEFECT DETECTION

Deep neural models have shifted defect detection from manually designed metrics toward representation learning directly from source code. Early studies commonly treated programs as token or statement sequences and applied CNNs, RNNs, LSTMs, or BiLSTMs to identify semantic and contextual patterns associated with defects. Code slices and function-level fragments were frequently used as model inputs. However, sequence-based deep learning methods still have obvious limitations. Unlike natural language text, source code defects are not only reflected in local token patterns but are also closely related to program behaviors such as variable definition and use, conditional branches, loop structures, function calls, memory access, and resource release. Representing code merely as a linear sequence may cause the model to ignore critical information such as control flow, data flow, and syntactic hierarchy. In addition, traditional RNN-or LSTM-based models are limited in modeling long-range dependencies when processing long code fragments, making it difficult to effectively capture defect-triggering conditions distributed across different statements. Therefore, constructing code representations with stronger semantic expressiveness and structural awareness has become a key issue in deep learning-based code defect detection.

### C. TRANSFORMER-BASED CODE REPRESENTATION

The self-attention mechanism enables Transformer models to capture long-range relationships among code tokens more efficiently than recurrent architectures. Building on this advantage, pretrained models such as GraphCodeBERT, PLBART, CodeT5, and UniXcoder learn rich semantic representations from large code corpora, including associations among identifiers, statements, structural elements, and comments [3–5,10]. In software code defect detection, Transformer-based methods can effectively capture contextual code semantics, including variable usage patterns, function call relationships, conditional contexts, and exception-handling logic. Compared with traditional CNN, RNN, or LSTM models, Transformer has stronger representation capability for long code fragments and can alleviate the loss of long-range dependency information. Nevertheless, Transformer-based models usually take token sequences as the main input. Their modeling process mainly relies on self-attention to automatically learn correlations among tokens, without explicitly incorporating program structures such as control flow, data flow, and syntactic dependencies. Since many code defects are closely associated with execution paths, variable propagation relationships, and cross-statement dependencies, relying only on Transformer-based sequence encoding may fail to fully represent program structural semantics. Therefore, although Transformer has advantages in semantic code modeling, it still needs to be combined with structured program representations to further improve defect detection performance.

### D. GNN-BASED PROGRAM STRUCTURE MODELING

To better represent structural information in source code, graph neural networks have been widely used in program analysis and code defect detection. Source code can be transformed into various program graph structures. For example, abstract syntax trees describe the syntactic hierarchy of code, control flow graphs represent program execution paths, data flow graphs capture dependencies between variable definitions and uses, and program dependence graphs further integrate control and data dependencies. Code property graphs unify multiple program structures into a single graph representation, providing richer structural semantics for code defect detection. Based on these program graphs, graph neural networks such as GCN, GAT, GGNN, and GraphSAGE can learn dependency patterns through message passing among nodes, thereby capturing code relationships associated with defects [6-9]. Compared with sequence models, GNNs can more directly model nonlinear structural relationships in source code. For instance, defects such as null pointer dereference, buffer overflow, incomplete resource release, and missing input validation are often closely related to variable propagation paths, conditional constraints, and control branches. Graph-based modeling enables the model to focus on these critical program dependencies[10-13]. However,

relying solely on GNNs also has limitations. On the one hand, the initial semantic representations of graph nodes are often insufficient. If only simple token embeddings or handcrafted features are used, the model may fail to obtain rich contextual semantic information. On the other hand, the message-passing process of GNNs is usually constrained by neighborhood ranges, which limits their ability to model global code semantics and long-range contextual dependencies. Therefore, recent studies have begun to explore how to combine the contextual semantic modeling capability of Transformer with the program structure modeling capability of GNNs to construct more comprehensive software code representations [14,15]. In summary, existing software code defect detection methods have evolved from traditional rule-driven and handcrafted-feature-based approaches to deep learning models, Transformer-based pre-trained models. However, a single type of model still struggles to simultaneously capture contextual code semantics and program structural dependencies. Transformer can effectively capture global semantic information in code sequences but lacks explicit modeling of program structures. GNNs can represent control flow, data flow, and syntactic dependencies, but their node semantic representations and global contextual understanding remain relatively limited. Therefore, the fusion of Transformer and graph neural networks has become a promising direction for improving code defect detection performance, providing the theoretical basis for the method proposed in this study.

## III. METHODOLOGICAL FRAMEWORK

### A. TASK FORMULATION AND LEARNING OBJECTIVE

Software defect identification is treated as a binary prediction problem at the function level. For the i-th function, the lexical representation obtained after tokenization is defined as:

$$S_i = (w_{i1}, w_{i2}, \ldots, w_{iL_i}) \tag{1}$$

where $w_{ij}$ denotes the token at position $j$ and $L_i$ is the number of tokens in the function.

In parallel, structural information is organized into a heterogeneous program graph:

$$G_i = (V_i, \mathcal{E}_i, T_i) \tag{2}$$

where $V_i$ contains program entities, $\mathcal{E}_i$ describes their connections, and $T_i$ distinguishes relation categories associated with syntax, execution flow, and variable dependence. Each function is assigned a binary target $z_i \in \{0, 1\}$; $z_i$= 1 denotes defective code, whereas $z_i$= 0 indicates a non-defective sample. The learning objective is to estimate the defect likelihood by jointly exploiting lexical and graph-based representations:

$$p_i = \mathcal{F}_\theta(S_i, G_i) \tag{3}$$

where $\theta$ represents the trainable parameter set and $p_i$is the estimated probability that the i-th function contains a defect.

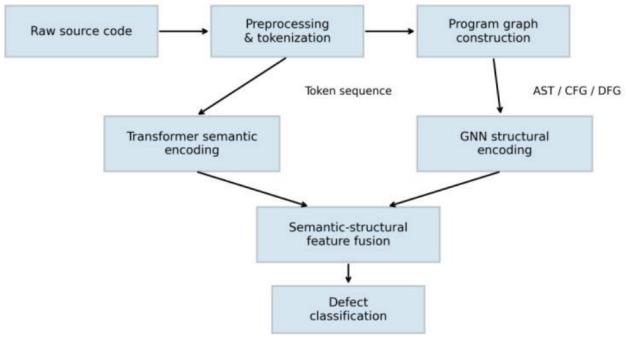


**FIGURE 1.** Architecture of the Transformer–GNN Framework for Software Defect Detection

### B. OVERALL FRAMEWORK

The proposed framework contains four stages: source-code preprocessing, program-graph generation, dual-branch representation learning, and defect classification. Function-level code fragments are converted into token sequences and graphs describing syntax, control flow, and data dependencies. The Transformer and GNN branches independently encode semantic context and structural relations, after which their outputs are fused to predict defective code, as illustrated in Figure 1.

The core idea of the proposed framework is to capture global contextual semantic information from code sequences using Transformer while explicitly modeling program structural dependencies using graph neural networks. Compared with a single sequence-based model, the proposed method can further perceive execution logic and variable propagation paths. Compared with a standalone graph-based model, it can obtain richer node semantic representations and stronger global contextual features. Therefore, the proposed framework constructs a more comprehensive code representation from both semantic and structural perspectives.

### C. CODE PREPROCESSING AND PROGRAM GRAPH CONSTRUCTION

In the code preprocessing stage, the source code is first standardized by removing comments, normalizing code format, eliminating irrelevant whitespace characters, and preserving key syntactic symbols. The code is then segmented into functions, which serve as the basic detection units, and each function is transformed into a token sequence. For excessively long code sequences, truncation or sliding-window strategies are used to control the input length. For shorter sequences, padding is applied to ensure consistency in batch processing. To further represent structural information in source code, this study constructs a multi-relation program graph. The nodes in the program graph can represent code tokens, statements, variables, or abstract syntax tree nodes, while the edges are used to describe program dependencies among nodes. Specifically, three types of structural relations are considered. The first type is syntactic relation, which describes the hierarchical syntactic composition of code. The second type is

control-flow relation, which represents execution paths among program statements. The third type is data-flow relation, which captures dependencies among variable definition, assignment, and use. By integrating these structural relations, the program graph can more comprehensively represent the static structure of code and potential defect propagation paths. For the i-th code sample, its program graph can be represented as:

$$G_i = (V_i, E_i^{\mathrm{AST}}, E_i^{\mathrm{CFG}}, E_i^{\mathrm{DFG}}) \tag{4}$$

where $E_i^{\mathrm{AST}}$, $E_i^{\mathrm{CFG}}$, and $E_i^{\mathrm{DFG}}$ denote abstract syntax tree edges, control-flow edges, and data-flow edges, respectively. Compared with using a single graph structure, the multi-relation program graph preserves syntactic hierarchy, execution order, and variable dependencies simultaneously, providing richer input for subsequent GNN-based structural encoding.

### D. TRANSFORMER-BASED SEMANTIC REPRESENTATION LEARNING

The semantic branch converts each tokenized function into a context-aware representation. Token identities are projected into a continuous space and combined with position-dependent vectors so that lexical meaning and token order are retained:

$$Z_i = \mathrm{Embed}(S_i) + P_i \tag{5}$$

where the embedding operator maps the lexical sequence into dense vectors, and the positional term distinguishes token locations within the function. The resulting matrix is processed by stacked Transformer blocks. Within each block, multi-head attention establishes direct interactions between distant program elements. A single attention head is expressed as:

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^{\mathsf{T}}}{\sqrt{d_h}}\right)V \tag{6}$$

where the query, key, and value matrices are learned projections, and the scaling factor is determined by the dimensionality of one attention head. After the final encoding layer, the function is represented by a sequence of context-sensitive token vectors:

$$H_i^{\mathrm{sem}} = (u_{i1}, u_{i2}, \ldots, u_{iL_i}) \tag{7}$$

Each vector summarizes the local token together with information propagated from other positions in the function. A sequence aggregation operator then compresses these token-level states into one semantic descriptor:

$$s_i = \mathrm{Aggregate}\left(H_i^{\mathrm{sem}}\right) \tag{8}$$

The aggregation may be implemented using a classification token, mean pooling, or learned attention. The resulting descriptor encodes usage context, invocation behavior, branching logic, and long-distance lexical interactions that are informative for defect discrimination.

### E. GRAPH-BASED STRUCTURAL REPRESENTATION LEARNING

The graph branch is introduced to characterize program relationships that cannot be fully preserved in a linear token stream. Context-aware token vectors generated by the semantic encoder are aligned with their associated program entities and used to initialize the graph nodes. For a node v, the initial state is expressed as

$$h_v^{(0)} = \mathrm{Align}(H_{T,v}) \tag{9}$$

where the alignment operator associates contextual token vectors with their corresponding graph entities. Relation-aware message propagation is then performed over the program graph. At each propagation layer, a node state is renewed by combining its previous representation with information received from relation-specific neighbors:

$$h_v^{(l+1)} = \phi\left(W_s^{(l)} h_v^{(l)} + \sum_{r\in\mathcal{R}} \sum_{u\in N_r(v)} a_{uv}^{r,l} W_r^{(l)} h_u^{(l)}\right) \tag{10}$$

The relation-specific neighborhood contains nodes connected through a given edge category. Separate trainable matrices transform the central state and incoming messages, while a learned attention coefficient controls the contribution of each neighbor before nonlinear activation. This design enables the network to assign unequal importance to syntactic, control-flow, and data-flow links. After the stacked propagation layers, the node-level structural embeddings are collected as:

$$H_G = \{h_1^{(L)}, h_2^{(L)}, \ldots, h_m^{(L)}\} \tag{11}$$

where m is the number of entities contained in the graph. A permutation-invariant aggregation operator subsequently converts the node set into a fixed-dimensional representation of the complete function:

$$h_G = \rho(H_G) \tag{12}$$

The readout stage may use mean, maximum, or attention-based pooling. Its output summarizes hierarchical syntax, feasible execution routes, data transfer, and variable propagation, thereby supplying relational evidence that complements the contextual information learned from the token sequence.

### F. SEMANTIC-STRUCTURAL FEATURE FUSION

To effectively integrate the semantic features extracted by Transformer and the structural features learned by GNN, this study designs a semantic-structural feature fusion module. Given the semantic representation $h_T$ from the Transformer branch and the structural representation $h_G$ from the GNN branch, the two representations are first concatenated, and a gating mechanism is used to adaptively learn the importance of the two types of features:

$$z = \sigma(W_z[h_T; h_G] + b_z) \tag{13}$$

where $[h_T; h_G]$ denotes the concatenation operation, $W_z$ and $b_z$ are learnable parameters, and z is the fusion gate. The final fused code representation is then computed as:

$$h_F = z \odot h_T + (1 - z) \odot h_G \tag{14}$$

where $\odot$ denotes element-wise multiplication and $h_F$ denotes the final semantic-structural representation. Through this mechanism, the model can automatically adjust the contribution of semantic and structural information according to different code samples. When a defect is mainly associated with contextual semantics or function call patterns, the model can assign higher weights to the Transformer branch. When a defect is more dependent on control flow, data flow, or variable propagation paths, the model can rely more on the structural representation produced by the GNN branch. Compared with simple feature concatenation, the gated fusion mechanism can more flexibly model the complementary relationship between semantic and structural features, preventing either type of feature from being weakened or disturbed by noise during the fusion process. This module is an important component of the proposed method and plays a key role in improving code defect detection performance.

### G. CLASSIFICATION OBJECTIVE AND TRAINING STRATEGY

The integrated feature vector is converted into a defect confidence score through a linear prediction head followed by logistic activation:

$$\widehat{p} = \text{sigmoid}\left(w_o^{\mathsf{T}} h_F + b_o\right) \tag{15}$$

Here, the weight vector and bias are trainable parameters of the output layer, and the resulting value represents the estimated likelihood of a defective function. A sample is assigned to the defective class when this score reaches a predefined decision threshold; otherwise, it is regarded as non-defective. Model parameters are learned by minimizing the average binary cross-entropy over a training batch:

$$\mathcal{L}_{\text{cls}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\widehat{p}_i) + (1 - y_i)\log(1 - \widehat{p}_i)\right] \tag{16}$$

In this expression, N is the batch size, while the target label and estimated defect probability correspond to the two sample-specific terms in the loss. Dropout, parameter regularization, and validation-based early stopping are applied to limit overfitting. When defective samples are underrepresented, weighted loss or balanced sampling may be used to prevent the majority class from dominating optimization. Training is performed jointly across the semantic encoder, graph propagation network, fusion mechanism, and output head. This end-to-end optimization allows the two representation branches to adapt to the classification objective and improves sensitivity to defect-related semantic and structural evidence.

## IV. EXPERIMENTS

### A. DATASET

To evaluate the effectiveness of the proposed Transformer-GNN fusion model for software code defect detection, this study adopts function-level code defect detection datasets as the experimental objects. Function-level detection is a commonly used task setting in current code defect detection research, aiming to determine whether a given function contains potential defects. Compared with project-level or file-level defect prediction, function-level detection provides a finer detection granularity and can more directly reflect the model's ability to understand code semantics and program structures. The experimental data were obtained from public software defect detection datasets, as summarized in Table I. Each sample contains a source-code function together with a binary label, where defective functions are marked as 1 and defect-free functions as 0. To ensure reliable evaluation, the data were divided into training, validation, and test subsets. The training subset was used to learn model parameters, while the validation subset supported hyperparameter tuning and early stopping. The test subset was kept separate and used only for final evaluation, thereby reducing the risk of information leakage during model selection.

In this study, the CodeXGLUE Defect Detection dataset is used as the main experimental dataset, while the Devign or Big-Vul dataset can be further adopted for extended validation. Since different datasets vary in code sources, defect types, and sample distributions, experiments on multiple datasets can further evaluate the generalization ability of the proposed model under different coding scenarios.

### B. BASELINE METHODS

Several representative baseline models were selected to provide a comprehensive evaluation of the proposed approach. These baselines cover conventional machine learning methods, sequence-based deep learning models, Transformer-oriented models, and GNN-based models. Through comparisons with different model categories, the advantages of the fusion framework in capturing code semantics and program structural dependencies can be more clearly examined. As shown in Table II. Among these baselines, CodeBERT and GraphCodeBERT are used to verify the effectiveness of Transformer-based semantic encoding, while GCN, GAT, and GGNN are used to examine the effectiveness of program graph structural modeling. By comparing these models, it can be determined whether the proposed semantic-structural fusion mechanism can further improve code defect detection performance.

**TABLE 1.** Description of the Experimental Dataset

| Dataset | Programming language | Detection level | Label type | Main purpose |
|---|---|---|---|---|
| CodeXGLUE Defect Detection | C/C++ | Function-level | Binary label | Main benchmark dataset |
| Devign Dataset | C/C++ | Function-level | Binary label | Supplementary comparison |
| Big-Vul Dataset | C/C++ | Function-level | Binary label | Generalization evaluation |

**TABLE 2.** Baseline Methods Used for Comparison

| Category | Baseline method | Description |
|---|---|---|
| Traditional machine learning | SVM | Uses manually extracted or statistical code features for classification |
| Traditional machine learning | Random Forest | Ensemble learning method based on decision trees |
| Traditional machine learning | XGBoost | Gradient boosting model for defect classification |
| Deep sequential model | CNN | Extracts local token-level patterns from code sequences |
| Deep sequential model | BiLSTM | Models bidirectional contextual dependencies in code sequences |
| Transformer-based model | CodeBERT | Uses pre-trained Transformer representations for code defect detection |
| Transformer-based model | GraphCodeBERT | Incorporates code semantic representation with data-flow awareness |
| GNN-based model | GCN | Learns structural information from program graphs |
| GNN-based model | GAT | Introduces attention weights into graph-based dependency modeling |
| GNN-based model | GGNN | Performs gated message passing over program graph structures |
| Proposed method | Transformer–GNN Fusion | Integrates semantic features and structural dependency features |

### C. EVALUATION METRICS

Software code defect detection is usually formulated as a binary classification task. However, since defective samples are often fewer than defect-free samples in real-world software systems, class imbalance may exist in the data distribution. Therefore, this study uses not only Accuracy to evaluate the overall classification correctness of the model, but also Precision, Recall, F1-score, AUC, and MCC for comprehensive evaluation. The evaluation metrics used in this study are summarized in Table III. Accuracy reflects the overall proportion of correctly classified samples, while Precision focuses on the reliability of defect predictions by measuring how many samples identified as defective are truly defective. Recall evaluates the model's ability to capture actual defects, indicating the coverage of defective samples. F1-score combines Precision and Recall through their harmonic mean, making it more suitable for imbalanced defect detection tasks. AUC assesses the discrimination capability of the model under different decision thresholds. In addition, MCC takes true positives, true negatives, false positives, and false negatives into account simultaneously, providing a more stable evaluation for binary classification performance.

Among these metrics, F1-score and AUC are the main evaluation indicators in this study. This is because, in code defect detection, the model is expected not only to accurately determine whether code contains defects, but also to maintain a good balance between false positives and false negatives. If Recall is low, many real defects may be missed. If Precision is low, excessive false alarms may increase the review burden for developers. Therefore, evaluating model performance with multiple metrics is more reasonable.

### D. IMPLEMENTATION DETAILS

The proposed model is optimized in an end-to-end training manner. The Transformer branch is used to extract semantic representations from code sequences, the GNN branch is used to extract structural representations from program graphs, and the fusion module integrates the two types of features for defect classification. In the implementation, source code is first preprocessed and tokenized, and then the corresponding program graph is constructed. The initial representations of graph nodes are obtained by mapping the token-level representations generated by the Transformer branch, so as to enhance the semantic expressiveness of GNN node features. As shown in table IV.

For a fair comparison, all models were evaluated using the same data partition and performance metrics. Baseline methods involving pre-trained models followed a consistent maximum input length and fine-tuning setting, while GNN-based methods used the same program graph construction strategy. For conventional machine learning models, statistical code features or embedding-derived representations were used as model inputs. The proposed Transformer–GNN fusion method was then systematically assessed from four aspects: overall detection performance, ablation analysis, graph structure comparison, and visualization-based case interpretation.

**TABLE 3.** Evaluation Metrics for Defect Detection

| Metric | Definition | Evaluation focus |
|---|---|---|
| Accuracy | Ratio of correctly classified samples | Overall classification performance |
| Precision | Ratio of true defective samples among predicted defective samples | False-positive control |
| Recall | Ratio of correctly detected defective samples among all defective samples | Defect discovery ability |
| F1-score | Harmonic mean of Precision and Recall | Balanced detection performance |
| AUC | Area under the ROC curve | Classification robustness under different thresholds |
| MCC | Correlation coefficient between prediction and ground truth | Robust performance under imbalanced data |

**TABLE 4.** Main Implementation Settings

| Parameter | Setting |
|---|---|
| Programming language | C/C++ |
| Detection granularity | Function-level |
| Maximum sequence length | 512 tokens |
| Transformer encoder | CodeBERT or Transformer encoder |
| GNN encoder | GAT/GGNN |
| Graph relations | AST edges, CFG edges, DFG edges |
| Hidden dimension | 256 |
| Batch size | 16 |
| Dropout rate | 0.3 |
| Optimizer | AdamW |
| Learning rate | 2e-5 for Transformer, 1e-3 for GNN and classifier |
| Loss function | Binary cross-entropy |
| Early stopping | Based on validation F1-score |
| Evaluation metrics | Accuracy, Precision, Recall, F1-score, AUC, MCC |

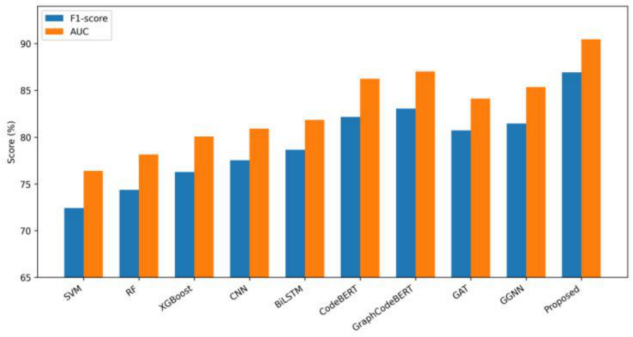


**FIGURE 2.** Overall Performance Comparison of Different Models

## V. RESULTS

### A. PERFORMANCE COMPARISON

The overall effectiveness of the proposed Transformer–GNN fusion model was assessed through comparisons with multiple categories of baseline methods, including conventional machine learning approaches, sequence-based deep learning models, Transformer-oriented methods, and GNN-based models. The detailed performance results are summarized in Table V, while Figure 2 provides a visual comparison of F1-score and AUC across different models.

The comparative results in Table V reveal that traditional machine learning methods provide only moderate detection performance. For example, SVM reaches an F1-score of 72.41% and an AUC of 76.38%. Random Forest and XGBoost improve the results compared with SVM, but their performance is still clearly behind that of deep learning models. This suggests that manually designed statistical features or shallow code representations are not sufficient to capture complex semantic patterns and dependency relationships in source code. Sequence-based neural models, including CNN and BiLSTM, achieve better results than conventional machine learning approaches, indicating that deep networks can automatically extract defect-related information from code sequences. However, these models mainly process code as linear token sequences and do not explicitly model program structures such as control flow, data flow, and variable dependencies. As a result, their detection capability remains weaker than that of Transformer-based methods, including CodeBERT and GraphCodeBERT. Among all compared models, the proposed Transformer–GNN fusion framework obtains the strongest overall performance. Its Accuracy, F1-score, and AUC reach 87.35%, 86.92%, and 90.47%, respectively. Compared with GraphCodeBERT, the proposed model increases the F1-score by 3.87 percentage points and the AUC by 3.45 percentage points. These findings indicate that combining Transformer-based contextual semantic learning with GNN-based program structure modeling can more effectively enhance software defect detection performance.

### B. ABLATION STUDY

An ablation analysis was carried out to examine the contribution of different components in the proposed framework. Four model variants were constructed by individually excluding the Transformer branch, the GNN branch, the gated fusion module, and the data-flow edges. Their performance was then compared with that of the complete model, as reported in Table VI and Figure 3.

The ablation results show that removing the Transformer

**TABLE 5.** Overall Performance Comparison of Different Defect Detection Models

| Model | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) | AUC (%) | MCC |
|---|---|---|---|---|---|---|
| SVM | 73.85 | 70.96 | 73.92 | 72.41 | 76.38 | 0.478 |
| Random Forest | 75.42 | 72.83 | 75.95 | 74.36 | 78.15 | 0.511 |
| XGBoost | 77.31 | 74.92 | 77.69 | 76.28 | 80.06 | 0.546 |
| CNN | 78.16 | 75.84 | 79.27 | 77.52 | 80.91 | 0.563 |
| BiLSTM | 79.08 | 77.16 | 80.19 | 78.64 | 81.83 | 0.586 |
| CodeBERT | 83.26 | 81.35 | 83.00 | 82.17 | 86.24 | 0.661 |
| GraphCodeBERT | 84.02 | 82.04 | 84.08 | 83.05 | 87.02 | 0.674 |
| GAT | 81.15 | 79.82 | 81.67 | 80.73 | 84.11 | 0.625 |
| GGNN | 82.04 | 80.64 | 82.30 | 81.46 | 85.36 | 0.643 |
| Proposed Model | 87.35 | 86.28 | 87.56 | 86.92 | 90.47 | 0.741 |

**TABLE 6.** Ablation Study of the Proposed Model

| Model Variant | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) | AUC (%) |
|---|---|---|---|---|---|
| w/o Transformer | 82.04 | 80.64 | 82.30 | 81.46 | 85.36 |
| w/o GNN | 84.02 | 82.04 | 84.08 | 83.05 | 87.02 |
| w/o Fusion Gate | 85.01 | 83.92 | 84.83 | 84.37 | 88.12 |
| w/o DFG Edges | 85.63 | 84.51 | 85.52 | 85.01 | 88.79 |
| Full Model | 87.35 | 86.28 | 87.56 | 86.92 | 90.47 |

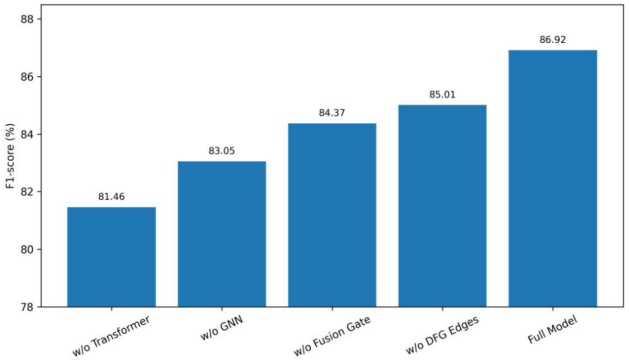


**FIGURE 3.** Ablation Study of the Proposed Model

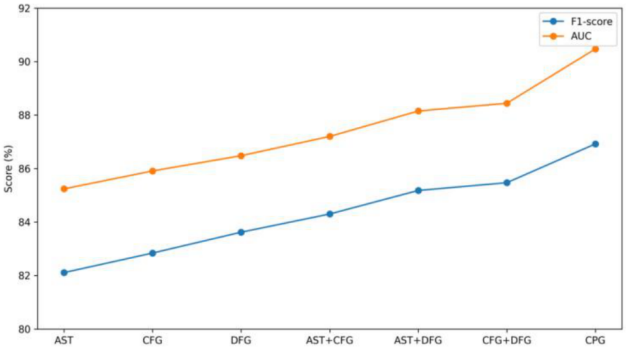


**FIGURE 4.** Effect of Different Program Graph Structures

branch leads to the most significant performance degradation, with the F1-score dropping to 81.46%. This indicates that the Transformer branch plays a critical role in capturing contextual code semantics and long-range token dependencies. In defective code, triggering conditions are often distributed across different statements, such as variable definitions, conditional checks, and dangerous function calls. Therefore, global semantic modeling is essential for effective defect detection. When the GNN branch is removed, the model degenerates into a sequence-based semantic model mainly relying on Transformer representations. The F1-score decreases to 83.05%, which is 3.87 percentage points lower than that of the full model. This confirms that program structural information is also important for defect detection. Many defects are closely related to control paths, variable propagation, and data dependencies, which cannot be fully represented by code token sequences alone. After removing the gated fusion module, the F1-score drops to 84.37%. This result indicates that simply concatenating semantic and structural features is not sufficient to fully exploit the complementarity between the two types of information. The gated fusion mechanism can adaptively adjust the contribution of Transformer-based features and GNN-based features according to different code samples, enabling the model to select more appropriate information sources under different defect patterns.

### C. EFFECT OF DIFFERENT GRAPH STRUCTURES

To analyze the influence of different program graph structures on model performance, this study further compares AST, CFG, DFG, and their combined graph structures. The results are shown in Table VII and Figure 4.

As shown in Table VII, although single graph structures can provide certain program structural information, their representation capability is limited. The AST-only model mainly describes the syntactic hierarchy of code and can capture syntactic structural patterns, but it is insufficient for representing execution paths and variable propagation relationships. Therefore, its F1-score is only 82.11%. The CFG-only model represents control paths and is more ef-

**TABLE 7.** Effect of Different Graph Structures on Detection Performance

| Graph Structure | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) | AUC (%) |
|---|---|---|---|---|---|
| AST only | 83.12 | 81.47 | 82.76 | 82.11 | 85.24 |
| CFG only | 83.65 | 82.10 | 83.59 | 82.84 | 85.91 |
| DFG only | 84.23 | 82.89 | 84.36 | 83.62 | 86.48 |
| AST + CFG | 84.91 | 83.52 | 85.09 | 84.30 | 87.20 |
| AST + DFG | 85.79 | 84.61 | 85.76 | 85.18 | 88.15 |
| CFG + DFG | 86.02 | 84.95 | 86.00 | 85.47 | 88.44 |
| CPG / AST + CFG + DFG | 87.35 | 86.28 | 87.56 | 86.92 | 90.47 |

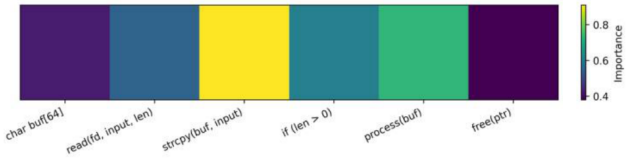


**FIGURE 5.** Visualization of Statement-Level Importance in a Defective Function

fective in modeling branch conditions and execution order, improving the F1-score to 82.84%. The DFG-only model further focuses on dependencies between variable definitions and uses, achieving an F1-score of 83.62%, which indicates that data dependency information is more sensitive to defect detection. When two types of graph structures are combined, the model performance further improves. Among them, AST + DFG and CFG + DFG outperform AST + CFG, suggesting that data-flow relations have stronger discriminative value for identifying defects. Especially for defects caused by variable propagation, missing input validation, and memory access errors, DFG provides more direct clues than purely syntactic structures. Finally, the CPG structure integrating AST, CFG, and DFG achieves the best performance, with F1-score and AUC reaching 86.92% and 90.47%, respectively.

### *D. VISUALIZATION AND CASE STUDY*

To further analyze the interpretability of the proposed model, this study selects a representative defective function for case analysis. The function contains a potential buffer overflow risk, mainly because input data are copied into a fixed-size buffer without sufficient length checking. Table VIII presents the importance scores assigned by the model to different statements, and Figure 5 visualizes the attention regions of the model in the defective function.

According to the results in Table VIII, the statement strcpy(buf, input) receives the largest contribution weight, suggesting that the model mainly relies on this unsafe memory-copy operation when judging the function as defective. Since strcpy may cause buffer overflow when boundary checking is insufficient, this statement has a strong association with the defect label. Meanwhile, the model also pays attention to process(buf) and if (len > 0), suggesting that it not only identifies the dangerous function call itself, but also considers the subsequent use of the buffer and whether the condition check is sufficient. The observed improvement can be attributed to the complementary representation of lexical context and graph-level relations, allowing the network to jointly model semantic interactions, data-flow behavior, and connectivity patterns within a program. For example, there is an evident data propagation path among read(fd, input, len), strcpy(buf, input), and process(buf). The GNN can model this dependency through data-flow and control-flow edges, while Transformer can capture semantic associations among dangerous function calls, buffer variables, and surrounding statements from the token sequence. Integrating the two representation streams yields more reliable defect identification and offers additional insight into the factors underlying each prediction. Overall, the evaluation confirms that jointly exploiting sequential semantics and graph-structured program information is a viable and effective strategy for identifying defective source code. Benchmark evaluations show that the dual-path framework identifies defective code more effectively than feature-engineered classifiers, recurrent or convolutional sequence learners, and models relying solely on semantic attention or graph propagation. The ablation analysis further reveals that semantic feature extraction, graph-based structural learning, data-flow modeling, and gated representation fusion each play an important role in improving model performance. In addition, the graph structure comparison suggests that multi-relation program graphs are better able to describe program semantics from multiple structural perspectives. The case analysis also illustrates that the model can identify code regions closely associated with defect formation. These results suggest that the proposed framework has advantages in detection accuracy, structural representation learning, and interpretability.

**TABLE 8.** Statement-Level Importance in a Defective Code Sample

| Statement | Structural/Semantic role | Importance score |
|---|---|---|
| `char buf[64]` | Fixed-size buffer declaration | 0.42 |
| `read(fd, input, len)` | External input acquisition | 0.55 |
| `strcpy(buf, input)` | Unsafe data copy operation | 0.91 |
| `if (len > 0)` | Incomplete condition check | 0.61 |
| `process(buf)` | Subsequent use of buffer | 0.73 |
| `free(ptr)` | Resource release operation | 0.38 |

## VI. CONCLUSION

A hybrid neural architecture is introduced to identify defective code by integrating token-level representations learned from program text with dependency information extracted from graph-structured program data. In this framework, the source code is first divided and processed at the function level. A multi-relation program graph is then built to represent syntactic organization, control logic, and data dependency patterns. The program is subsequently encoded through two complementary pathways: long-range contextual relationships between lexical units are modeled by the attention-based branch, whereas graph message passing is employed to characterize the connections among program entities. The semantic and structural representations are subsequently integrated through an adaptive fusion module and used for final defect classification. Evaluation across multiple baselines confirms the superiority of the proposed framework over conventional classifiers, sequence-oriented neural architectures, and standalone semantic or graph-based networks. The ablation analysis further shows that semantic representation learning, graph-based structural modeling, data-flow information, and the gated fusion strategy all make positive contributions to detection performance. In addition, the comparison of graph construction strategies suggests that integrating AST, CFG, and DFG enables the program graph to describe code structure more comprehensively, thereby helping the model identify potential defects more effectively. By jointly modeling semantic dependencies within source-code tokens and relational patterns embedded in program graphs, the developed framework delivers more reliable defect identification while providing clearer insight into its prediction process. Future work may proceed in several directions. First, the current function-level detection task can be extended to line-level or statement-level localization, which would improve its applicability in practical code review. Second, large-scale pre-trained code models or large language models may be introduced to strengthen the modeling of complex code contexts and cross-function dependencies. Future work should examine the model under cross-domain conditions involving heterogeneous codebases, multiple programming paradigms, and practical engineering environments, thereby determining its transferability and deployment potential throughout the software lifecycle.

## REFERENCES

[1] N. S. Harzevili, A. B. Belle, J. Wang, et al., "A systematic literature review on automated software vulnerability detection using machine learning," ACM Comput. Surv., vol. 57, no. 3, pp. 55:1–55:36, 2024.
[2] Y. Zheng, S. Pujar, B. Lewis, et al., "D2A: A dataset built for AI-based vulnerability detection methods using differential analysis," in Proc. 43rd IEEE/ACM Int. Conf. Softw. Eng.: Softw. Eng. Pract., Piscataway, NJ, USA: IEEE, 2021, pp. 111–120.
[3] D. Guo, S. Ren, S. Lu, et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learn. Representations (ICLR), 2021.
[4] W. U. Ahmad, S. Chakraborty, B. Ray, et al., "Unified pre-training for program understanding and generation," in Proc. 2021 Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol., Stroudsburg, PA, USA: Association for Computational Linguistics, 2021, pp. 2655–2668.
[5] Y. Wang, W. Wang, S. Joty, et al., "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in Proc. 2021 Conf. Empirical Methods in Natural Language Processing (EMNLP), Stroudsburg, PA, USA: Association for Computational Linguistics, 2021, pp. 8696–8708.
[6] V. A. Nguyen, D. Q. Nguyen, V. Nguyen, et al., "ReGVD: Revisiting graph neural networks for vulnerability detection," in Proc. 44th Int. Conf. Softw. Eng. Companion, New York, NY, USA: ACM, 2022, pp. 51–55.
[7] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in Proc. 19th Int. Conf. Mining Softw. Repositories, New York, NY, USA: ACM, 2022, pp. 608–620.
[8] H. Hin, A. Kan, H. Chen, et al., "LineVD: Statement-level vulnerability detection using graph neural networks," in Proc. 19th Int. Conf. Mining Softw. Repositories, New York, NY, USA: ACM, 2022, pp. 596–607.
[9] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in Proc. 2022 Int. Joint Conf. Neural Networks (IJCNN), Piscataway, NJ, USA: IEEE, 2022, pp. 1–8.
[10] D. Guo, S. Lu, N. Duan, et al., "UniXcoder: Unified cross-modal pre-training for code representation," in Proc. 60th Annu. Meeting Assoc. Comput. Linguistics, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022, pp. 7212–7225.
[11] N. T. Islam, G. De La Torre Parra, D. Manuel, et al., "An unbiased transformer source code learning with semantic vulnerability graph," in Proc. 2023 IEEE 8th Eur. Symp. Security Privacy, Piscataway, NJ, USA: IEEE, 2023, pp. 144–159.
[12] Y. Chen, Z. Ding, L. Alowain, et al., "DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection," in Proc. 26th Int. Symp. Research Attacks, Intrusions Defense, New York, NY, USA: ACM, 2023, pp. 654–668.
[13] F. Qiu, Z. Liu, X. Hu, et al., "Vulnerability detection via multiple-graph-based code representation," IEEE Trans. Softw. Eng., vol. 50, no. 8, pp. 2178–2199, 2024.
[14] R. Liu, Y. Wang, H. Xu, et al., "Source code vulnerability detection: Combining code language models and code property graphs," arXiv preprint arXiv:2404.14719, 2024.
[15] A. Lekssays, H. Mouhcine, K. Tran, et al., "LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models," in Proc. 34th USENIX Security Symp., Berkeley, CA, USA: USENIX Association, 2025.