FINDGATE: Fine-grained Defect Prediction Based on a Heterogeneous Discrete Code Graph-guided Attention Transformer

Jiaxi Xu[1,2], Ping Chen[1,2,*], Banghu Yin[3,*], Zhichang Huang[1,2] and Qiaochun Qiu[1,2]
[1] China Electronic Product Reliability and Environmental Testing Research Institute, Guangzhou, Guangdong, China
[2] The Ministry of Industry and Information Technology Key Laboratory of Performance and Reliability Testing and Evaluation for Basic Software and Hardware, Guangzhou, Guangdong, China
[3] College of System Engineering, National University of Defense Technology, Changsha, Hunan, China
xujiaxi@ceprei.com, chenping@ceprei.com, bhyin@nudt.edu.cn, huangzhichang@ceprei.com, qiuqiaochun@163.com
*corresponding author

Abstract—Recognizing defects in source code through deep learning methods has become an important research subject for improving software quality. Although Transformer-based models such as *CodeBERT* have demonstrated impressive performance improvement in defect prediction tasks, models relying on single-structured input data, such as sequences, have limited ability to capture the code's structural features. Treating code simply as text overlooks essential information such as control dependencies, data dependencies, and syntactic structures inherent in the code. This paper proposes the Heterogeneous Discrete Code Graph (*HDCG*), which assigns structural information to code tokens from multiple perspectives. We also introduce an improved transformer model *FINEGATE* that leverages *HDCG* to guide self-attention. The experiments demonstrate that *FINEGATE* can effectively predict source code defects and perform fine-grained defect localization.

Keywords- Software defect prediction; Code graph; Fine-grained; Transformer; Code structure

1. INTRODUCTION

With the rapid growth in software scale and complexity and the acceleration of software iteration, the difficulty of identifying and locating software defects through testing has increased significantly, leading to exponential growth in time and resource costs. In China, billions of dollars are spent annually to support software testing. **S**oftware **d**efect **p**rediction (SDP) has emerged as one of the fastest-growing and most widely recognized technologies in the field of software reliability in the past decade. Currently, machine learning, deep learning, and other artificial intelligence techniques have been extensively applied by researchers in software defect prediction. By learning from historical defect knowledge, preliminary analysis and diagnosis of software defect situations have been conducted, providing guidance for software testing, reducing costs, and improving cost-effectiveness.

1.1. Motivation

Early software defect prediction methods primarily involved manually defining and extracting defect features from software. These features were then used to train machine learning models, which were subsequently utilized for predicting defects in new versions of software. In recent years, transformer-based sequential deep learning models, which have been rapidly developed, have demonstrated impressive performance across various domains. In the field of software, a series of Transformer-based methods, led by *CodeBERT* as a representative, have emerged. Instead of manual feature engineering, these methods treat software code as textual data for learning and representation, resulting in improved performance in software defect prediction tasks.

Represented by *CodeBERT*, these methods typically discretize software code into token sequences, which are then input into a Transformer model with an embedding layer and self-attention modules. The fundamental idea behind these methods is to treat software code as a textual document, extract feature representations using a Transformer model, and then classify them with a multilayer perceptron. These end-to-end approaches effectively utilize a pretrained model to learn semantic information from code, eliminating the requirement of manual feature engineering, which shows improved performance in SDP tasks. However, unlike ordinary text, code possesses distinct structural characteristics. The sequential order of code lines alone cannot sufficiently represent those structures, which is often overlooked in these methods.

This paper proposes the **H**eterogeneous **D**iscrete **C**ode **G**raph (***HDCG***) to enhance the structural information of code tokens without introducing additional nodes, which addresses the abovementioned issues. Structural information is introduced through the improved **G**raph-guided **A**ttention **T**ransformer **E**nhanced (***GATE***) model. Moreover, to achieve fine-grained defect localization, we reference and enhance the method proposed by Liu et al., computing risk scores for code lines through multidimensional attention.

1.2. Contribution

Overall, this paper mainly includes the following contributions:

1. The paper introduces the *HDCG* to achieve structural information enhancement for code tokens, addressing structure information absent issues in current Transformer-based *SDP* methods while avoiding introducing redundant information to the input sequence.

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00017

2. The ***GATE*** model is proposed, which utilizes structural information and enhances SDP performance by guiding self-attention using *HDCG*.
3. ***FINDGATE***, an improved **Fin**e-grained **D**efect Localization method based on the **GATE** model, is proposed. This approach is capable of training the model with function-level data and achieves defect localization with a finer granularity.
4. A comprehensive set of experiments was conducted on a large-scale dataset to demonstrate the effectiveness of the proposed FINDGATE method. The comparative experimental results demonstrate that our proposed method achieves better or comparable performance to the current state-of-the-art methods in defect prediction and fine-grained defect localization. The results of the ablation experiments show that the multidimensional-attention-based localization method achieves a significant improvement compared to the single-dimensional-attention-based method.

1.3. Paper Organization

The remainder of the paper is organized as follows: Section 2 gives brief reviews of software defect prediction and transformer-based methods. Section 3 introduces our proposed method *FINDGATE*. Section 4 provides the experimental setting. The results are provided and analyzed in Section 5. Section 6 discloses the threats to validity. Section VII draws the conclusions.

2. Related Works

This section mainly discusses the current literature work in the field of software defect prediction based on machine learning, graph learning-based models, and fine-grained SDP.

Software defect prediction:

Machine learning-based SDP became mainstream in this domain after the first decade of the 21st century. According to Hata et al.[1], machine learning-based methods accounted for 66% of the total relevant research papers in 2009. Traditional ML-based SDP methods typically involve historical data collection and defect labeling, manually designed metric extraction, model training with feature data, and prediction. Later with the advancement of deep learning techniques, which gradually replaced feature engineering, sequence models such as RNNs were applied in SDP.

Graph-based SDP:

Based on architectures such as RNN, sequence models usually consider code as token sequences without taking the software structure into account. With the rise of graph-based learning, researchers began to focus on the graph representation of code. For instance, Li et al. proposed a series of methods such as SyseVR[2] and VulDeePecker[3], which incorporate code graph attributes such as data flow graphs into sequence models through traversal. To address the low adaptability of sequence models to graph structures, many works have introduced graph representation learning algorithms, such as graph convolutional neural networks (GCNs), into SDP tasks (e.g., GNN-DP[4], Reveal[5], Devign[6] and code2vul[7]). The methods above, which focus on defect prediction at the file-level or function-level, have not yet provided solutions for fine-grained SDP.

Transformer-based SDP:

Mainstream defect prediction methods cover a large area in their predictions, making it highly challenging for code reviewers to inspect the entire region within a limited time and effort. According to statistics[8], less than 3% of the lines of code in defective source files were actually defective. In the 2020s, to guide testing developers and code reviewers to focus their attention on smaller high-risk areas, some solutions have been proposed from the perspectives of training data and prediction results to narrow the scope of predictions.

Some researchers have approached narrowing the data range by using finer-grained samples. Liu et al.[9] slice the code to reduce the sample size and introduce intermediate code to construct code representations for vulnerability detection. Our previous work ACGDP[4] proposed a method of graph representation learning by extracting subgraphs through static analysis on Augmented-CPGs. Wartschinski et al.[10] introduced VUDENC, which utilizes a sliding window to capture token sequences as input for LSTM. With the rise of transformer-based models, Le et al.[11] combined program slicing methods with the CodeBERT model.

Another group of researchers has focused on further explaining the prediction results to achieve fine-grained localization from coarse-grained predictions. IVDetect[12] uses the FA-GCN method to predict function-level vulnerabilities and employs GNNExplainer to pinpoint fine-grained locations. LineVD[13] utilizes CodeBERT and GAT to learn function-level and statement-level representations, respectively, and combines the results to achieve statement-level judgments. JITLINE[14] and LINEDP[8] employ the Local Interpretable Model-agnostic Explanations (LIME) technique for line-level computation. With the widespread use of attention mechanisms, works such as [15], DeeplineDP[16], and DPEA[17] utilize attention mechanisms for line-level scoring.

Balancing prediction accuracy and fine-grained localization data narrowing is challenging. Therefore, we are more optimistic about the approaches that focus on explaining the prediction results. In our work, we achieve fine-grained localization by interpreting the results of the graph-guided model.

3. Methodology

In this section, we introduce our novel **fin**e-grained software **d**efect prediction method based on Heterogeneous Discrete Code **G**raph-guided **A**ttention **T**ransformer **E**nhance model (*FINDGATE*), with three main components: Heterogeneous Discrete Code Graph (*HDCG*), graph-guided-attention transformer enhance model (*GATE*) and *GATE*-based fine-grained defect localization method (*FIND*).

3.1. Framework

This section introduces the framework of our method, which is divided into three stages, as shown in Figure 1: 1. Graph construction of *HDCG*; 2. *GATE* model-based defect prediction; 3. fine-grained defect localization.

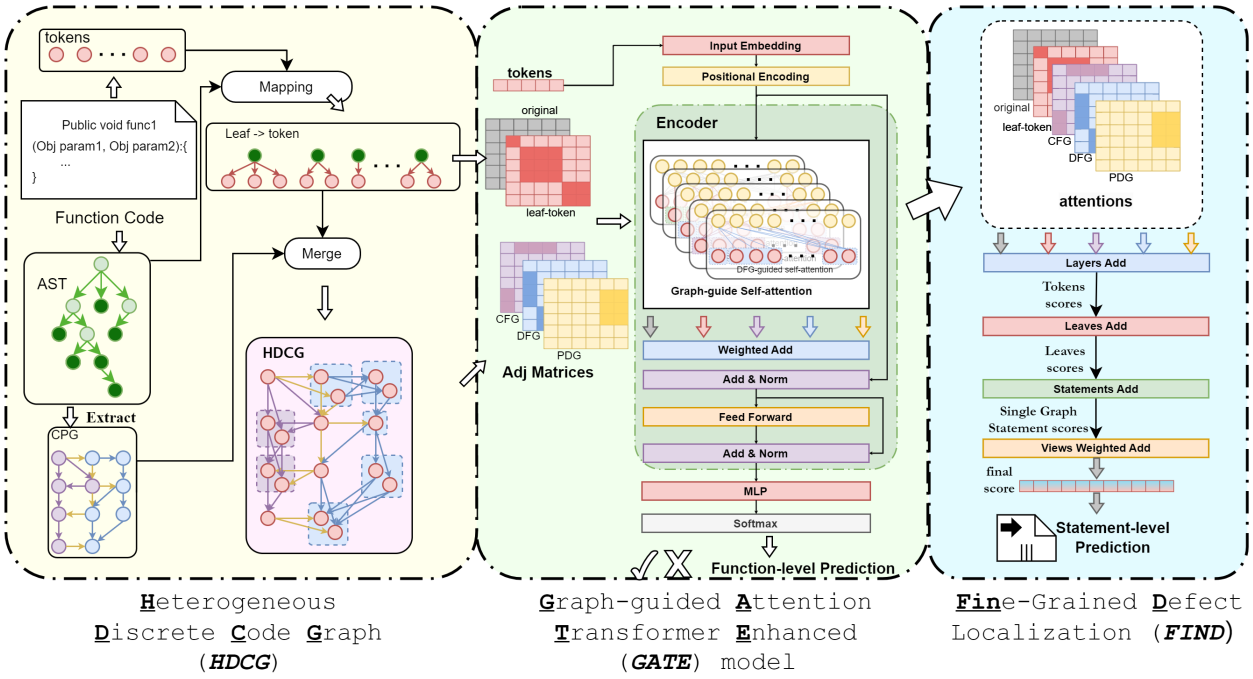

Figure 1. Overview of proposed approach *FINDGATE*

In the first stage, the source code is parsed into an abstract syntax tree (AST) and discretized into tokens. Tokens are mapped with AST leaf nodes through position. The Augmented-CPG is built as described in our previous work[4]. The Augmented-CPG is collapsed, keeping only the leaf nodes. Heterogeneous Discrete Code Graph was then yielded by expanding AST leaves into token nodes based on leaf-tokens map.

In the second stage, we propose an enhanced transformer-based SDP model by utilizing HDCG to guide self-attention. The HDCG captures different types of relationships and guides self-attention modules in different encoders to generate a comprehensive source code representation. A multilayer perceptron (MLP) is employed to classify the generated representation and perform the defect prediction task.

In the third stage, fine-grained defect localization was performed by utilizing the self-attention values from each graph-guided encoder based on the hypothesis "tokens that are most contributed to the predictions are likely to be vulnerable" proposed by Fu et al.[15] For each encoder, token-attention scores were calculated by heads-sum and layers-sum. Statement-attention scores were obtained by progressively aggregating the attention scores based on the leaf-token map and statement-leaf map. The statement scores from each encoder were summed up and ranked to indicate defect risks.

3.2. Graph construction

As mentioned earlier, a series of Transformer-based models, with CodeBERT[18] as a representative, have achieved remarkable performance in the SDP task. However, numerous studies have indicated that code structural information plays a significant role in code representation and SDP tasks. Non-Euclidean structured graphs are a suitable form for structural information reinforcement of code token sequences. As transformer-based models typically struggle to process graph data as input directly, methods such as GraphCodeBERT[19] have been attempted. These approaches often convert graph data nodes into sequences through traversal, inevitably leading to some structural information loss. On the other hand, appending the graph node sequence to the original token sequence increases the length of the input, limiting the length of the code token sequence in models of equal capacity. For example, CodeBERT-base can accept 512 code tokens in a sample at maximum, while GraphCodeBERT-base with the same model size can only accept 256 code tokens, half of CodeBERT-base's capacity, which significantly limits the applicability of the model.

We propose a graph representation called Heterogeneous Discrete Code Graph (HDCG), where code tokens serve as nodes and various structural relationships are attached. HDCG consists solely of code token nodes, and the input sequence fed into the model does not contain any redundant nodes, effectively addressing the issue of complex output sequences in methods such as GraphCodeBERT. The pure input sequence composed of code tokens also plays a better role in supporting the subsequent fine-grained defect localization based on graph-guided attention. The construction of HDCG involves the following steps: 1. Code tokenization, 2. Abstract Syntax Tree (AST) generation, 3. Leaf-token mapping, 4. Augmented-CPG generation, and 5. HDCG construction.

Code tokenization:

In the field of language processing, models require a vocabulary to represent sentences during both training and prediction. The total vocabulary size typically ranges from approximately 170 thousand to 1 million words in natural language processing (NLP). It is usually several orders of

magnitude higher in program language processing (PLP). In this paper, the Byte Pair Encoding (BPE) algorithm, originally developed for data compression, is used for code tokenization. BPE iteratively identifies the most frequent byte pairs in the data and merges them into a new "byte" token. By applying BPE tokenization, the size of the vocabulary is adequately reduced and "Out of Vocabulary" (OOV) words are represented with token combinations. In this paper, we employed a BPE tokenizer pretrained on CodeSearchNet[20] to generate a tokenizer specifically suitable for programming languages.

AST generation:

To investigate the structural representation of code, researchers have used various code abstraction graphs in SDP, with abstract syntax trees (ASTs) being one of the most widely used. There are already some tools available for generating ASTs for different programming languages. To support different programming languages and enable language-agnostic analysis, we employed the open-source tool tree-sitter for AST generation and standardized the results into language-agnostic directed acyclic graphs (DAGs) $G_{ast} = (V_{ast}, E_{ast})$.

Node $v_{ast} = (src, pos, type_v) \in V_{ast}$ is represented as a triplet, where src denotes the source code, $pos = (s, e) = ((line_{start}, col_{start}), (line_{end}, col_{end}))$ represents the position in the source code, and $type_v$ denotes the node type. Edge $e_{ast} = (v_{source}, v_{target}, type_e) \in E_{ast}$ is defined as a triplet consisting of the source node v_{source}, the target node v_{target}, and the edge type$type_e$ = AST.

$L = \{ v_i^{leaf} | v_i^{leaf} \in V_{ast}, 0 < i < m \}$ represents the subset of leaf-nodes (leaves) in the AST, and $S = \{ s_i | 0 < i < n \}$ denotes the set of statements in the source code. The i-th statement $s_i = [v_j^{leaf}, v_{j+1}^{leaf}, \dots]$ is composed of leaf nodes $v_j^{leaf}, v_{j+1}^{leaf}, \dots$. Matrix $A^{s-l} \in R^{m \times n}$ represents the mapping relationship between statements and leaf nodes. Each element a_{ij}^{s-l} in the matrix is defined as follows:

$$a_{ij}^{s-l} = \begin{cases} 1, & \text{if } v_i^{leaf} \in s_j \\ 0, & \text{else} \end{cases} \quad (1)$$

Leaf-token mapping:

To incorporate the graph structure into tokens, it is necessary to establish a mapping between the leaves and tokens. Some studies have simply concatenated the tokenized results of the leaf nodes and tokens, treating them as a single token sequence. Some studies have simply linked leaves and tokens by tokenizing and concatenating leaf-nodes' source code instead of tokenizing the complete code. However, we found that this kind of approach results in differences with tokens generated directly from the complete source code. That might not affect the performance of tasks like code summarization but does make an impact on the SDP result.

We directly map the source code tokens to the leaf nodes based on positional information to address this issue. First, preprocessing steps such as comments and document removal, as well as rare symbol replacement, are applied to the source code. Then, the position $pos_i^t = (s_i, e_i)$, which indicates that the i-th token t_i is composed of characters from the s_i-th to the $(e_i - 1)$-th position in the original text, is obtained for each token by traversing the source code and token sequence. For each leaf node, the positional transformation is applied to convert its position into $pos_i^{ast} = (s_i, e_i)$ through the following formula:

$$s_i = \sum_j^{i-1} length_j^{line} + col_i^{start} \quad (2)$$

$$e_i = s_i + \left(col_i^{end} - col_i^{start}\right) \quad (3)$$

Matrix $A^{l-s} \in R^{m \times n}$ represents the mapping relationship between leaf-nodes and tokens. Each element a_{ij}^{l-t} in the matrix is defined as follows:

$$a_{ij}^{l-t} = \begin{cases} 1, & \text{if } pos_i^t \in pos_j^{ast} \\ 0, & \text{else} \end{cases} \quad (4)$$

Augmented-CPG generation:

In previous work ACGDP[4], we proposed Augmented-CPG, a code graph that fused Abstract Syntax Tree (AST) with additional relationships such as data dependencies, control dependencies, and function calls. An Augmented-CPG generator based on ANTLR has been developed. However, due to the limitations of the ANTLR tool, this generator is designed explicitly for only compilable Java language code.

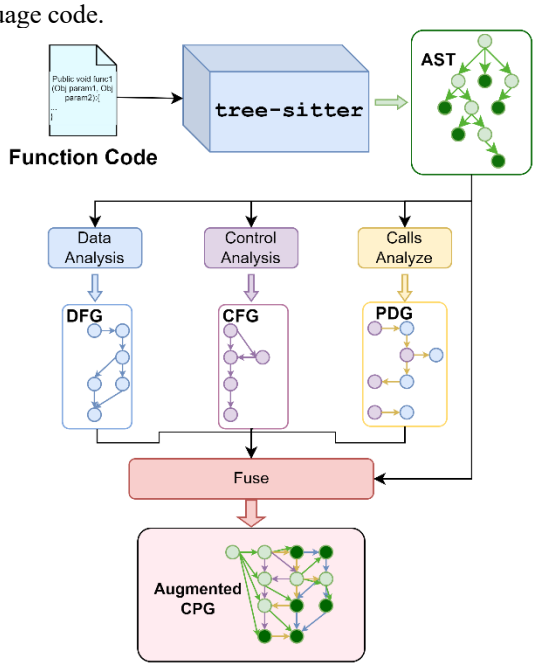

Figure 2 Augmented-CPG generation

1 https://github.com/tree-sitter/tree-sitter

After conducting research, we opted to utilize the tree-sitter[1] tool as the AST extraction tool, allowing us to redevelop an updated Augmented-CPG generator that can be applied to multiple programming languages, including Java, Python, C, C++, C#, Ruby, and more. Moreover, it does not require the code to be compilable. The extraction process of Augmented-CPG is shown in Figure 2.

$\text{Augmented} - \text{CPG} = (\text{V}, \text{E})$ is a multi-DiGraph consisting of AST nodes $\text{V} = \text{V}_{\text{ast}}$ and edges $\text{E} = \{\text{E}_{\text{ast}}, \text{E}_{\text{dfg}}, \text{E}_{\text{cfg}}, \text{E}_{\text{call}}\}$of various views.

HDCG construction:

Since Augmented-CPG is constructed based on the AST, which includes nodes of different abstraction levels, we first collapse the Augmented-CPG by removing non-leaf nodes and the AST edges E_{ast} in order to generate input consisting only of code tokens. During the collapsing process, other kinds of edges are connected to all the child nodes of the collapsed node through the outgoing AST edges. For example, in Figure 3, edge $e_{1,2}^{dfg}$ is a dataflow edge from AST node v_1^{ast} to v_2^{ast}. After collapsing, this edge is transformed into three data flow edges connecting the leaf nodes. Leaf-edges E_{leaf} between tokens from the same leaf-node are appended.

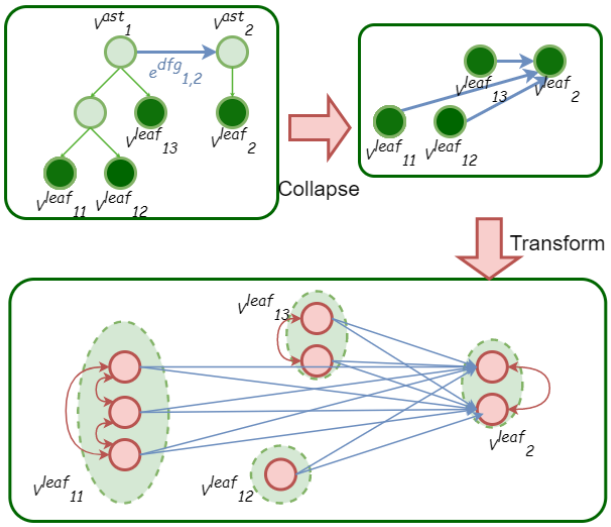

Figure 3 Augmented-CPG to HDCG

Then, as shown in Figure 3, all the leaf nodes are replaced with token nodes according to the leaf-token map, constructing an HDCG that consists only of discrete token nodes and edges.

Indicate edges $\text{E}_{\text{leaf}}, \text{E}_{\text{dfg}}, \text{E}_{\text{cfg}}, \text{E}_{\text{call}}$ of Augmented-CPG as adjacency matrices $M^{leaf}, \text{M}^{\text{dfg}}, \text{M}^{\text{cfg}}, \text{M}^{\text{call}}$. The relationship of AST nodes and AST leaf nodes can be represented as a matrix $\text{A}^{\text{ast}-\text{l}}$. The adjacency matrices of HDCG edges can be computed as follows:

$$\begin{cases} \text{M}_{\text{h}}^{\text{leaf}} = \text{M}^{\text{leaf}} \\ \text{M}_{\text{h}}^{\text{dfg}} = \text{M}^{\text{dfg}} \times \text{A}^{\text{ast}-\text{l}} \times \text{A}^{\text{l}-\text{t}} \\ \text{M}_{\text{h}}^{\text{cfg}} = \text{M}^{\text{cfg}} \times \text{A}^{\text{ast}-\text{l}} \times \text{A}^{\text{l}-\text{t}} \\ \text{M}_{\text{h}}^{\text{call}} = \text{M}^{\text{call}} \times \text{A}^{\text{ast}-\text{l}} \times \text{A}^{\text{l}-\text{t}} \end{cases} \quad (5)$$

3.3. Graph-guided Attention Transformer Encoder

The **G**raph-guided **A**ttention **T**ransformer **E**ncoder (***GATE***) is an improved sequence-to-sequence encoder based on the Transformer encoder. It is designed to adapt to the input of *HDCG* and learn the structural information of the code. Compared to recent Transformer-based approaches that attempt to learn code structures, *GATE* does not introduce additional sequences in the input, ensuring its ability to handle longer code.

The *HDCG* generated from a given code S is represented using the token sequence$T = [t_1, t_2, \dots, t_n]$ and the adjacency matrices $M_h^{leaf} \in R^{n\times n}, M_h^{dfg} \in R^{n\times n}, M_h^{cfg} \in R^{n\times n}$, and $M_h^{call} \in R^{n\times n}$. Here,$\text{t}_\text{i}$ represents the i-th token in the code S.

Input embedding:

In terms of input embedding, we follow an approach similar to widely adopted transformer-based code learning models like CodeBERT and CodeT5. First, we convert the token sequence T into a numerical sequence X by looking up the vocabulary. Then, we trim or pad the sequence to a specified parameter l to ensure its length. Finally, the sequence is embedded into a high-dimensional space $\mathbb{R}^d$ using an embedding layer.

Sequence X is trimmed or padded:

$$\grave{\text{X}} = \begin{cases} [< \text{C} >, x_1, x_2, \dots, x_{l-2}, < \text{S} >], & |\text{X}| > \text{l} - 2 \\ [< \text{C} >, x_1, x_2, \dots, x_{|X|}, < \text{S} >, 0, \dots, 0], & \text{else} \end{cases} \quad (6)$$

Additionally, the adjacency matrix M_h is trimmed if $|X| > l - 2$:

$$\grave{M}_h = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & m_{0,0} & \dots & m_{0,l-2} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & m_{l-2,0} & \dots & m_{l-2,l-2} & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \quad (7)$$

If $|X| \le l - 2$, the adjacency matrix M_h is padded:

$$\grave{M}_h = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \dots 0 \\ 0 & m_{0,0} & \dots & m_{|X|} & 0 \dots 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & m_{|X|,0} & \dots & m_{|X|,|X|} & 0 \dots 0 \\ 0 & 0 & \dots & 0 & 0 \dots 0 \\ & & \vdots & & \\ 0 & 0 & \dots & 0 & 0 \dots 0 \end{bmatrix} \quad (8)$$

Graph guided Attention

Traditional Transformer encoders typically use positional embeddings to capture the sequential order of tokens for attention computation. However, in *HDCG*, tokens have more complex structural relationships. To address this, we propose utilizing *HDCG* for structure-aware self-attention computation, resulting in graph-guide attention (GA):

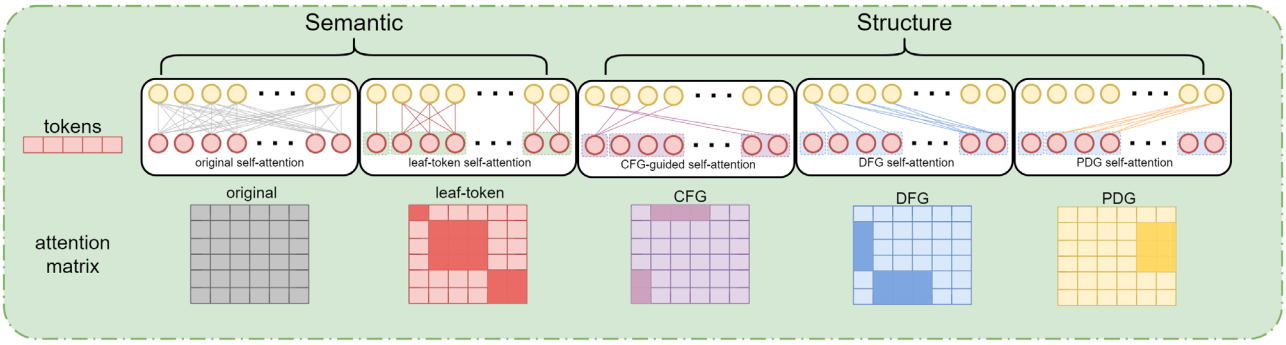

Figure 4. Graph guided self-attention

$$\begin{cases} \mathrm{A}^{\mathrm{origin}}(\mathrm{X}) = \mathrm{A}(\grave{\mathrm{X}}, 1) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{dk}}}\right)\mathrm{X} \\ \mathrm{A}^{\mathrm{leaf}}(\grave{\mathrm{X}}) = \mathrm{A}\left(\grave{\mathrm{X}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{leaf}}}\right) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{dk}}}\mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{leaf}}}\right)\grave{\mathrm{X}} \\ \mathrm{A}^{\mathrm{dfg}}(\grave{\mathrm{X}}) = \mathrm{A}\left(\grave{\mathrm{X}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{dfg}}}\right) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{dk}}}\mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{dfg}}}\right)\grave{\mathrm{X}} \\ \mathrm{A}^{\mathrm{cfg}}(\grave{\mathrm{X}}) = \mathrm{A}\left(\grave{\mathrm{X}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{cfg}}}\right) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{dk}}}\mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{cfg}}}\right)\grave{\mathrm{X}} \\ \mathrm{A}^{\mathrm{call}}(\grave{\mathrm{X}}) = \mathrm{A}\left(\grave{\mathrm{X}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{call}}}\right) = \mathrm{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{dk}}}\mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{call}}}\right)\grave{\mathrm{X}} \end{cases} \quad (9)$$

Unlike methods like *GraphCodeBERT* and *StructCoder* that attempt to incorporate data flow by modifying the input sequence, *GATE* does not alter the input sequence. Instead, it appropriately integrates structural information by changing the attention calculation. Importantly, since this approach does not modify the input sequence, it allows for the addition of structural information from various perspectives without being limited by the sequence length by parallelizing multiple self-attention modules.

Graph guided Encoder

Through the graph-guide attention module, attention scores from different perspectives can be calculated. There are two methods for combining multi-perspective attention: summation and concatenation. In the concatenation method, the hidden representation is obtained by concatenating the outputs of each attention module:

$$\mathrm{H} = \mathrm{Concat}\left(\mathrm{A}^{\mathrm{origin}}, \mathrm{A}^{\mathrm{dfg}}, \mathrm{A}^{\mathrm{cfg}}, \mathrm{A}^{\mathrm{call}}\right) \quad (10)$$

In the summation method, the hidden representation is obtained by element-wise summation of the outputs of each attention module:

$$\mathrm{H} = \mathrm{Sum}\left(\mathrm{A}^{\mathrm{origin}}, \mathrm{A}^{\mathrm{dfg}}, \mathrm{A}^{\mathrm{cfg}}, \mathrm{A}^{\mathrm{call}}\right) \quad (11)$$

The summation method keeps the exact representation dimensions with different numbers of input perspectives.

We used the weighted summation for computing the hidden representation:

$$H = \begin{array}{c} W_1 \cdot A^{o}rigin + W_1 \cdot A^{l}eaf + W_2 \cdot A^{d}fg \\ +W_3 \cdot A^{c}fg + W_4 \cdot A^{c}all \end{array} \quad (12)$$

$\mathrm{W_i}$ is a learnable weight matrix that can adaptively adjust the importance of various structural information.

The expression of GATE for the trimmed or padded input $\grave{\mathrm{X}}$ and $\grave{\mathrm{M}}_{\mathrm{h}} = \left[1, \mathrm{M}_{\mathrm{h}}^{\mathrm{leaf}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{dfg}}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{cfg}}}, \mathrm{M}_{\mathrm{h}}^{\grave{\mathrm{call}}}\right]$ can be represented as follows:

$$\mathrm{GATE}\left(\grave{\mathrm{X}}, \grave{\mathrm{M}}_{\mathrm{h}}\right) = \sum \mathrm{W_i} \cdot \mathrm{A}\left(\grave{\mathrm{X}}, \grave{\mathrm{M}}_{\mathrm{h}}[\mathrm{i}]\right) \quad (12)$$

Classifier

We utilized a two-layer fully connected neural network followed by a softmax function as the defect classifier. To mitigate overfitting, we also incorporated dropout regularization.

3.4. Fine-Grained Defect Localization

Indeed, existing software defect datasets often provide defect labels at the file or function level due to various challenges, such as data collection difficulty and model complexity, which lead to the same prediction model granularity. However, not all code segments within a file or function labeled as defective are necessarily faulty. Therefore, even after defect prediction, code reviewers still need to invest considerable effort in reviewing the entire suspected module, although most of the code segments within the predicted defective module may not pose a high risk.

In some previous work[2], [4], [9], [11], [21], researchers attempted to refine samples by combining code-slicing methods to achieve more fine-grained results. However, these methods tend to focus on specific types of vulnerabilities, and excessively fine-grained code slicing can significantly degrade prediction performance. Recently, some researchers[15], [16] proposed the hypothesis that "tokens contributing the most to predictions are likely to be vulnerable" and conducted validation studies. Building upon this hypothesis, we calculate statement defect risk scores by leveraging the self-attention values from each graph-guided encoder.

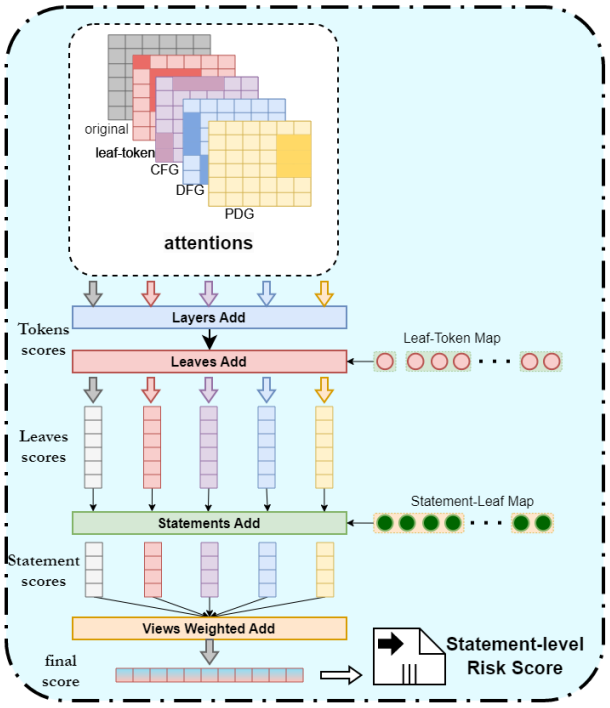

Figure 5 GATE-based Fine-Grained Defect Localization

As shown in Figure 5, in the final layer of the model, the attention matrix $M_A \in R^{n \times n}$ of each GATE module is row-summed to obtain the attention scores for each token:

$$Score^t = M_A \times [1,1,\dots,1]^T = \left[s_1^t, s_2^t, \dots, s_{|T|}^t\right] \quad (13)$$

In which, s_i^t represents the score of token t_i in that GATE module. Then, the scores of all tokens belonging to the same leaf are summed to obtain the score of the leaves:

$$Score^l = (A^{l-t} \times Score^t)^T = \left[s_1^l, s_2^l, \dots, s_{|L|}^l\right] \quad (14)$$

Similarly, the scores of all leaves belonging to the same statement are summed to obtain the score of the statement:

$$Score^s = (A^{s-l} \times Score^l)^T = \left[s_1^s, s_2^s, \dots, s_{|S|}^s\right] \quad (15)$$

Finally, the scores of statements calculated by each GATE module are weighted summed element-wise to obtain the final score for each statement:

$$Score = \sum w_j^s \cdot Score_j^s \in R^{\mathbb{1} \times |S|}$$

This score serves as the defect risk score for the statement, where a higher score indicates a higher likelihood of a defect.

3.5. Training and prediction

HDCGs are generated for each sample of the function level software defects dataset using the method described in Section III.A. The *FINDGATE* model is trained and hyperparameter tuned for function-level SDP tasks using the training set and validation set. The function-level and statement-level SDP performance of the trained model is evaluated on the test set. This approach significantly reduces the difficulty of obtaining training data because it does not require data with statement-level defect labels during training and tuning.

4. Experiment

4.1. Research Questions

To validate our proposed *FINDGATE*, we presented and conducted the following research questions (RQs). We addressed the research problems and validated our proposed strategy by analyzing the experimental observations.

RQ1: How accurate is our *FINDGATE* for function-level defect predictions?

RQ2: How accurate is our *FINDGATE* for line-level defect localization?

RQ3: What is the cost-effectiveness of our *FINDGATE* for line-level defect localization?

RQ1 and RQ2 aim to validate the performance of the proposed method on coarse-grained and fine-grained tasks. RQ3 focuses explicitly on the cost-effectiveness validation of the proposed method. RQ2 and RQ3 also conducted experiments using the graph-only module to conduct ablation studies, intending to verify the contribution of the graph structure to task performance.

4.2. Datasets

In recent studies[6], [8], [13], [15], [16], [22], datasets such as *Devign*[6], *Big-Vul*[23], and *LineDP*-dataset[8] are commonly used for performance evaluation of these methods. To ensure experimental reproducibility and facilitate benchmark comparisons, we conducted an investigation on these datasets. *Devign* and *Big-Vul* are datasets that include real-world open-source C/C++ projects, while the *LineDP* dataset consists of Java projects. Due to the data collection process, *Devign* is a balanced dataset, whereas *Big-Vul* and *LineDP*-dataset reflect the imbalanced distribution of real-world code. *Big-Vul* has function-level granularity with approximately three times the sample size of the LineDP-dataset.

We used the Big-Vul dataset, which was split into an 8:1:1 ratio by *LineVul*[15]. HDCGs for 188,636 functions in the dataset were generated using the Augmented-CPG generator, and no cases of generation failure were observed (7% of samples encountered parsing failures in *DeepDFA*[22]).

4.3. Experiment Design

For RQ1, to verify the performance of the proposed *FINDGATE* on the function-level defect prediction task, it is compared with the following methods: Transformer-based methods, GNN-based methods, sequence-based methods (RNN, BiLSTM), and traditional machine learning (ML) based methods.

Table 1 compares methods for RQ1

Type	Method	Year
Transformer-based	LineVul[15]	2022
GNN+Transformer	DeepDFA+LineVul[22]	2022
GNN-based	DeepDFA[22]	2022
	DeepVD[24]	2022
	Devign[6]	2019
	IVDetect[12]	2021
	Reveal[5]	2020
BiLSTM-based	SySeVR[2]	2018
	VulDeePecker[3]	2020
RNN-based	DeepLineDP[16]	2022
	Russell et al.[25]	2018
ML-based	LineDP[8]	2020

Following the conventions in defect prediction research, we evaluated the performance of our method using precision, recall, and F1 score.

For the baselines in RQ2 and RQ3, as shown in Table 2, we selected representative methods for line-level defect (vulnerability) localization: *LineVul*[15], *DeepLineDP*[16], and *JITLine*[14]. These methods are based on Transformer self-attention, RNN-attention, and LIME (Local Interpretable Model-Agnostic Explanations). Additionally, to conduct ablation experiments and investigate the effect of *GATE*, we trained a model that uses only graph-guided attention without the original attention (graph-only).

Table 2 compares methods for RQ2&3

Method	Localization way	Year
LineVul	Self-attention	2022
DeepLineDP	Attention	2022
JITLine	LIME	2021
Graph-Only(ablation)	Self-attention	\

In accordance with the experiments of the baseline methods, for RQ2, we used the following metrics to evaluate the accuracy of line-level localization:

1. Top-10 Accuracy: The probability that at least one real defective line is among the top ten lines ranked by risk in the defect samples.
2. IFA (Initial False Alarm): The average number of lines that are needed to find the first defective line when inspecting according to risk rank.
3. Total Effort: The average number of lines that need to be inspected to discover all the defective lines.

For RQ3, we used the following metrics to evaluate the cost-effectiveness:

1. Effort@20%Recall: The ratio of the number of lines inspected to discover 20% of the defective lines. It measures the efficiency of defect discovery.
2. Recall@1%loc: The percentage of defective lines that can be discovered by inspecting only the top 1% of the lines ranked by risk. It indicates the effectiveness of the ranking strategy in identifying defects.

5. Results And Analysis

The experimental results and analysis are presented in this section.

5.1. RQ1: How accurate is our *FINDGATE* for function-level defect predictions?

The experimental results for RQ1 are presented in Table 3, where the best performance is highlighted in bold. It can be observed from Table 3 that *FINDGATE* achieved the best performance in terms of precision, recall, and F1 score, surpassing *LineVul*, which also uses Transformer as the backbone model, by 1%, 5%, and 3%, respectively.

Table 3 RQ1 results

Type	Method	Precision	Recall	F1
Transformer-based	*FINDGATE	**0.98**	**0.91**	**0.95**
	LineVul[15]	0.97	0.86	0.92
GNN+ Transformer	DeepDFA+ LineVul[22]	**0.98**	0.90	0.94
GNN-based	DeepDFA[22]	0.54	0.90	0.67
	DeepVD[24]	0.70	0.78	0.74
	Devign[6]	0.26	0.18	0.21
	IVDetect[12]	0.23	0.72	0.35
	Reveal[5]	0.19	0.74	0.30
RNN-based	DeepLineDP[16]	0.42	0.83	0.56
	Russell et al.[25]	0.24	0.16	0.19
BiLSTM-based	SySeVR[2]	0.15	0.74	0.25
	VulDeePecker[3]	0.12	0.49	0.19
ML-based	LineDP[8]	0.48	0.17	0.25

The closest performing method is *DeepDFA+LineVul* proposed by Steenhoek et al., which utilized GNN to learn structure and *CodeBERT* to learn semantics, achieving similar precision and slightly lower recall. Steenhoek et al. and we employed different approaches in incorporating code graph structure knowledge into the Transformer-based method, both resulting in improved performance, highlighting the positive contribution of graph structure knowledge to SDP tasks.

Table 4 Average performance of different model types

Type	Precision$^{(+/-)}$	Recall$^{(+/-)}$	F1$^{(+/-)}$
*FINDGATE	0.98	0.91	0.95
Transformer+Graph	0.98	0.91	0.94
Transformer-based	$0.97^{\uparrow 0.01}$	$0.86^{\uparrow 0.05}$	$0.92^{\uparrow 0.03}$
GNN-based	$0.54^{\uparrow 0.60}$	$0.90^{\uparrow 0.25}$	$0.67^{\uparrow 0.49}$
Sequence-based	$0.42^{\uparrow 0.75}$	$0.83^{\uparrow 0.36}$	$0.56^{\uparrow 0.65}$
ML-based	$0.48^{\uparrow 0.50}$	$0.17^{\uparrow 0.74}$	$0.25^{\uparrow 0.69}$

Table 4 provides the average performance of different model types. Overall, the combination of graph knowledge and Transformer (*FINDGATE* and *DeepDFA+LineVul*) achieved the best performance, closely followed by the Transformer-only method (*LineVul*). The GNN-based methods, which leverage code graph knowledge, outperformed the sequence-based methods. Traditional ML methods showed relatively poor performance on our dataset.

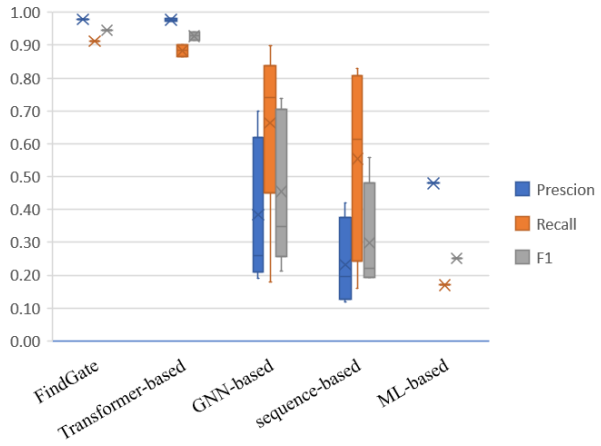

Figure 6 SDP performance of different types of models

5.2. RQ2: How accurate is our *FINDGATE* for line-level defect localization?

Table 5 illustrates the performance of different methods on line-level defect localization, with the best score highlighted in bold for each metric. The differences between *FINDGATE* and other methods are provided in the upper right corner. It can be observed that *FINDGATE* achieved the best scores in top-10 accuracy, IFA, and total effort, which validated our proposed approach.

Table 5 Performance on line-level defect localization

Method	Top-10 Accuracy	IFA	Total Effort
*FINDGATE	**0.8**	**5.48**	**0.49**
LineVul	0.65 $^{\uparrow 0.15}$	5.77 $^{\downarrow 0.29}$	0.52 $^{\downarrow 0.03}$
DeepLineDP	0.59 $^{\uparrow 0.21}$	10.71 $^{\downarrow 5.23}$	0.56 $^{\downarrow 0.07}$
JITLine	0.10 $^{\uparrow 0.70}$	24.20 $^{\downarrow 18.72}$	0.54 $^{\downarrow 0.27}$
Graph-Only	0.48 $^{\uparrow 0.32}$	6.00 $^{\downarrow 0.52}$	0.55 $^{\downarrow 0.06}$

In terms of top-10 accuracy, *FINDGATE* achieves a score of 0.80, while other baseline methods range from 0.10 to 0.65. *FINDGATE's* accuracy is 19%-88% higher than that of the other baseline methods. For IFA, *FINDGATE* achieves a score of 5.48, while other baseline methods range from 5.77 to 10.8. *FINDGATE* requires checking an average of 5%-77% fewer lines to discover the first defective line compared to other baseline methods. Regarding total_effort, *FINDGATE* achieves a score of 0.49, while other baseline methods range from 0.52 to 0.56. *FINDGATE* requires 6%-36% less effort compared to other baseline methods.

LineVul, which also uses Transformer as the backbone model, achieved slightly lower performance, indicating that utilizing the built-in self-attention mechanism in Transformer for line-level defect localization is superior to the other two methods.

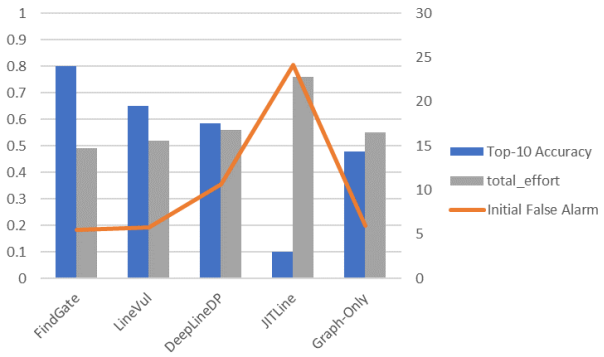

Figure 7 Line-level defect localization performance

In addition, we also attempted to perform line-level defect localization using only graph-guided self-attention. The results showed that it achieved accuracy only higher than *JITLine*, and achieved better IFA compared to both *JITLine* and *DeepLineDP*. It might be safe to conclude that semantic knowledge contributes the most to line-level defect localization, followed by graph structure knowledge.

5.3. RQ3: What is the cost-effectiveness of our *FINDGATE* for line-level defect localization?

Table 6 presents the cost-effectiveness of each method in line-level defect localization, with the best score highlighted in bold for each metric. The differences between each method and *FINDGATE* are shown in the upper right corner. It can be observed that *FINDGATE* achieves the best results in both Effort@20%Recall and Recall@1%LOC.

Table 6 Cost-effectiveness on line-level defect localization

Method	Effort@20%Recall	Recall@1%LOC
*FINDGATE	**0.0088**	**0.22**
LineVul	0.0107 $^{\downarrow 0.002}$	0.19 $^{\uparrow 0.03}$
DeepLineDP	0.0208 $^{\downarrow 0.012}$	0.03 $^{\uparrow 0.19}$
JITLine	0.0150 $^{\downarrow 0.006}$	0.09 $^{\uparrow 0.13}$
Graph-Only	0.0148 $^{\downarrow 0.006}$	0.13 $^{\uparrow 0.09}$

In terms of Effort@20%Recall, *FINDGATE* achieves a score of 0.0088, while other baseline methods range from 0.0107 to 0.0208. *FINDGATE* requires 18%-58% less effort than other baseline methods, meaning *FINDGATE* needs less effort to identify the same number of defective lines.

In terms of Recall@1%LOC, *FINDGATE* achieves a score of 0.22, while other baseline methods range from 0.02 to 0.19. *FINDGATE* discovers 16%-87% more defective lines compared to other baseline methods, which means that *FINDGATE* can identify more defective lines with the same effort.

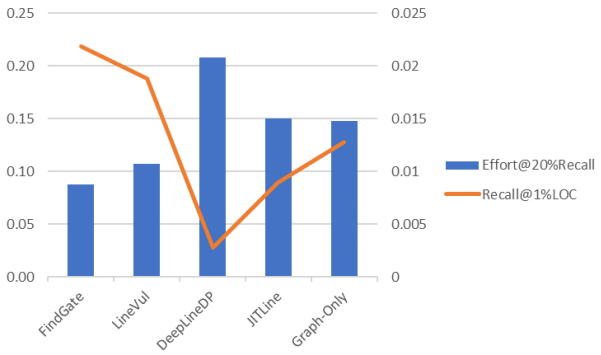

Figure 8 Cost-effectiveness of line-level defect localization

6. Threats to Validity

6.1. Construct Validity

The threat to structural validity is related to the dataset selection. We used a version of the *Big-Vul* dataset[23] split by Fu et al.[15] We did not use some commonly used public defect prediction datasets[26] because they only provide file-level or function-level labels, which are not suitable for line-level research. Pornprasit's dataset[16] has line-level labels, but the samples are at the file granularity rather than the function granularity, and the dataset size is relatively small. To ensure a fair comparison, we selected the same dataset, *Big-Vul*, as *LineVul*[15], *IVDetect*[12], and *DeepDFA*[22] et al.

Additionally, *JITLine*[14] was not included in the comparison in RQ1 because its prediction task focuses on code changes under just-in-time scenarios, which makes it difficult to compare directly. As a substitute, we compared our approach with *Line-DP*[8] in RQ1, which also utilizes BOW+RF.

6.2. External Validity

The external validity threats are related to the generalizability of our *FINEGATE* method. We conducted experiments on the large-scale line-level defect dataset, *Big-Vul*, to ensure a fair comparison with other methods. However, in future work, exploring other line-level defect datasets would be valuable to further validate our approach's effectiveness.

6.3. Internal validity

The internal validity threats are related to the hyperparameter settings during the fine-tuning of the *FINEGATE* model. For the backbone model, we used the default hyperparameter settings of the encoder in *codeT5* to avoid the computational cost of hyperparameter tuning for a transformer model with millions of parameters, which is beyond our resource constraints. Additionally, due to time and resource limitations, the *FINDGATE* model used in this study was only trained for 10 epochs. Increasing the number of training epochs may improve the performance of the model.

In our experiments, we directly used the *LineVul* model trained by Fu et al[15]. We also used open-source reimplementation packages to experiment with models like *DeepLineDP*[2], *DeepVD*[3] and *DeepDFA*[4]. For models such as *IVDetect* that were not reproducible (which is also a concern in other studies), we reused the results reported in Fu and Steenhoek's papers[12], [15], [22], ensuring strict consistency in the data partitioning method.

To mitigate these threats, we will provide detailed experimental data publicly, and we expect to complete the organization and release of our replication package within half a year after the publication of our paper, which will enhance the transparency of our work.

7. Conclusion

In the field of defect prediction and vulnerability detection, fine-grained localization at the line level has gained significant attention. In this paper, we propose *FINDGATE*, a method for defect prediction and fine-grained localization that leverages transformer-based models to learn code structure and semantic knowledge simultaneously. Using the innovative *GATE* (Graph-guided Attention Transformer Encoder) to learn *HDCG*, we address the limitations of other methods that attempt to learn graph knowledge, such as loss of structural information, input sequence redundancy, and over-complexity.

By conducting empirical evaluations on large-scale real-world datasets and comparing *FINDGATE* with state-of-the-art methods such as *LineVul* and *DeepDFA*, we demonstrate that *FINDGATE* achieves the following:

1) In function-level prediction tasks, *FINDGATE* performs slightly better than the state-of-the-art method *DeepDFA+LineVul* (which used a more complicated model) and shows 3%-392% F1 score improvements compared to other baseline methods.
2) In line-level localization tasks, *FINDGATE* achieves a 19%-88% improvement in top-10 accuracy and reduces effort by 18%-58% in terms of cost-effectiveness.
3) Through the ablation experiments of a graph-only model in RQ2&RQ3, we further validate the positive contribution of graph structure knowledge to model performance.

Therefore, by integrating code graph structure knowledge with semantic knowledge, the *FINDGATE* model can assist software testing analysts in defect prediction and localization more accurately and efficiently.

Acknowledgment

This work was supported by Research on FPGA Software Code Rule Set and Code Defect Pattern Library (JWVY227200010) and the Key-Area Research and Development Program of Guangdong Province (No. 2020B0909030005).

2 https://github.com/awsm-research/DeepLineDP

3 https://github.com/deepvd2022/deepvd2022

4 https://github.com/whitemech/DeepDFA

REFERENCES

[1] H. Hata, T. Kikuno, and O. Mizuno, “A systematic review of software fault prediction studies and related techniques,” *Comput. Softw.*, vol. 29, no. 1, pp. 106–117, 2012, doi: 10.11309/jssst.29.1_106.

[2] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities,” pp. 1–13, 2018, [Online]. Available: http://arxiv.org/abs/1807.06756

[3] Z. Li *et al.*, “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection,” *arXiv*, no. February, 2018, doi: 10.14722/ndss.2018.23158.

[4] J. Xu, J. Ai, J. Liu, and T. Shi, “ACGDP: An Augmented Code Graph-Based System for Software Defect Prediction,” *IEEE Trans. Reliab.*, vol. 71, no. 2, pp. 850–864, 2022, doi: 10.1109/TR.2022.3161581.

[5] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep Learning based Vulnerability Detection: Are We There Yet?,” *Ieee Trans. Softw. Eng.*, vol. TBD, p. 1, 2020, [Online]. Available: https://git.io/Jf6IA.

[6] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Adv. Neural Inf. Process. Syst.*, vol. 32, pp. 1–11, 2019.

[7] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, “Learning to map source code to software vulnerability using code-as-a-graph,” *Icst*, pp. 1–8, 2019.

[8] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting Defective Lines Using a Model-Agnostic Technique,” *IEEE Trans. Softw. Eng.*, no. 1, pp. 1–23, 2020, doi: 10.1109/TSE.2020.3023177.

[9] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, “VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector,” *IEEE Trans. Dependable Secur. Comput.*, no. ii, pp. 1–15, 2021, doi: 10.1109/TDSC.2021.3076142.

[10] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, “VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python,” *Inf. Softw. Technol.*, vol. 144, Apr. 2022, doi: 10.1016/j.infsof.2021.106809.

[11] T. H. M. Le and M. A. Babar, *On the Use of Fine-grained Vulnerable Code Statements for Software Vulnerability Assessment Models*, vol. 1, no. 1. Association for Computing Machinery, 2022. [Online]. Available: http://arxiv.org/abs/2203.08417

[12] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability detection with fine-grained interpretations,” *ESEC/FSE 2021 - Proc. 29th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, pp. 292–303, 2021, doi: 10.1145/3468264.3468597.

[13] D. Hin, A. Kan, H. Chen, and M. A. Babar, “LineVD: Statement-level Vulnerability Detection using Graph Neural Networks,” *MSR ’22 Proc. 19th Int. Conf. Min. Softw. Repos. May 23-24, 2022, Pittsburgh, PA, USA*, vol. 1, no. 1, 2022, [Online]. Available: http://arxiv.org/abs/2203.05181

[14] C. Pornprasit and C. K. Tantithamthavorn, “JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction,” *Proc. - 2021 IEEE/ACM 18th Int. Conf. Min. Softw. Repos. MSR 2021*, pp. 369–379, 2021, doi: 10.1109/MSR52588.2021.00049.

[15] M. Fu and C. Tantithamthavorn, “LineVul: A Transformer-based Line-Level Vulnerability Prediction,” in *19th International Conference on Mining Software Repositories (MSR ’22), May 23â•fi24, 2022, Pittsburgh, PA, USA*, 2022, vol. 1, no. 1. doi: 10.1145/3524842.3528452.

[16] C. Pornprasit and C. Tantithamthavorn, “DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction,” *IEEE Trans. Softw. Eng.*, 2022, doi: 10.1109/TSE.2022.3144348.

[17] T. Zhang, Q. Du, J. Xu, J. Li, and X. Li, “Software defect prediction and localization with attention-based models and ensemble learning,” *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, vol. 2020–Decem, pp. 81–90, 2020, doi: 10.1109/APSEC51365.2020.00016.

[18] Z. Feng *et al.*, “CodeBERT: A pre-trained model for programming and natural languages,” *Find. Assoc. Comput. Linguist. Find. ACL EMNLP 2020*, pp. 1536–1547, 2020, doi: 10.18653/v1/2020.findings-emnlp.139.

[19] D. Guo *et al.*, “GraphCodeBERT: Pre-training Code Representations with Data Flow,” pp. 1–18, 2020, [Online]. Available: http://arxiv.org/abs/2009.08366

[20] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” 2019, [Online]. Available: http://arxiv.org/abs/1909.09436

[21] W. Zheng, Y. Jiang, and X. Su, “VulSPG: Vulnerability detection based on slice property graph representation learning,” *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, vol. 2021–Octob, pp. 457–467, 2021, doi: 10.1109/ISSRE52982.2021.00054.

[22] B. Steenhoek, W. Le, and H. Gao, “DeepDFA: Dataflow Analysis-Guided Efficient Graph Learning for Vulnerability Detection,” pp. 1–15, 2022, [Online]. Available: http://arxiv.org/abs/2212.08108

[23] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries,” *Proc. - 2020 IEEE/ACM 17th Int. Conf. Min. Softw. Repos. MSR 2020*, pp. 508–512, 2020, doi: 10.1145/3379597.3387501.

[24] W. Wang, “DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection.” https://github.com/deepvd2022/deepvd2022

[25] R. Russell *et al.*, “Automated Vulnerability Detection in Source Code Using Deep Representation Learning,” *Proc. - 17th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2018*, no. Ml, pp. 757–762, 2019, doi: 10.1109/ICMLA.2018.00120.

[26] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, “Mining Software Defects: Should We Consider Affected Releases?,” *Proc. - Int. Conf. Softw. Eng.*, vol. 2019–May, pp. 654–665, 2019, doi: 10.1109/ICSE.2019.00075.