

SGRL: A Hybrid Semantic-Graph Representation Learning Framework for Software Vulnerability Detection

Cong Bui Van

Posts and Telecommunications Institute of Technology (PTIT),
122 Hoang Quoc Viet, Nghia Do Ward, Hanoi City, Vietnam

Abstract—The increasing scale and complexity of software increase the risk of security vulnerabilities, creating a need for automated and effective detection methods. Transformer-based methods can learn semantic representations but do not fully exploit structural information, whereas graph neural network-based methods remain limited in representing token-level semantics. In addition, data imbalance affects models' learning and generalization capabilities. In this study, the author proposes SGRL (Semantic-Graph Representation Learning), a framework that combines CodeT5 and a Graph Convolutional Network (GCN) over a Code Property Graph (CPG) to jointly exploit the semantic and structural features of source code. The two feature sources are fused through concatenation and adjusted using BDC (Bernoulli Dropout in CNN) before classification by a Multi-Layer Perceptron (MLP). Experiments on the Verum dataset with the optimal configuration of a 512-token input length and a BDC probability of 0.2 achieve an Accuracy of 84.23%, Precision of 84.55%, Recall of 84.23%, and F1-score of 84.14%. Results on Verum and FFmpeg+Qume also show that SGRL outperforms REVEAL, Russell, VulDeePecker, SySeVR, and Devign under the corresponding experimental settings.

Keywords—Source code vulnerability detection; CodeT5; graph convolutional network; code property graph; representation learning; deep learning

I. INTRODUCTION

A. Problem Statement

In the context of digital transformation and the development of large-scale software systems, ensuring source code security is essential to minimizing the risk of vulnerability exploitation and system compromise. Programming errors such as buffer overflow, use-after-free, null pointer dereference, and out-of-bounds write can introduce serious vulnerabilities and may even lead to remote code execution [1]. Accordingly, source code vulnerability detection generally involves two main tasks: 1) extracting vulnerability-related indicators and features; and 2) analyzing these features to determine the presence of vulnerabilities [1–5]. On this basis, current source code vulnerability detection methods can be divided into two main categories: 1) rule- and pattern-based methods; and 2) machine learning- and deep learning-based methods.

1) *Rule-based detection*: Rule-based methods identify vulnerabilities using rules, patterns, or indicators of unsafe behavior derived from expert knowledge and known

vulnerabilities. Common techniques include static analysis, pattern matching, reverse engineering, and testing. These methods have the advantages of interpretability and the ability to detect predefined types of vulnerabilities effectively. However, constructing and updating rules requires considerable expert effort; their generalization capability is limited, they are prone to false alarms, and they have difficulty detecting new types of vulnerabilities.

2) *Machine learning- and deep learning-based detection*: Machine learning and deep learning methods transform source code into feature representations to learn the differences between vulnerable and secure source code. The development of Transformer models such as BERT [6], RoBERTa [7], CodeBERT [8], and CodeT5 [9] has improved the ability to learn semantic and contextual representations of source code through relationships among tokens. However, these models primarily represent source code as sequences and do not directly model structural relationships such as syntax, control flow, and data dependencies.

Conversely, Graph Neural Network (GNN)-based methods can exploit program structure through graphs such as the Abstract Syntax Tree (AST) [10], Control Flow Graph (CFG) [11], Program Dependence Graph (PDG) [12], and Code Property Graph (CPG) [13]. In particular, CPG combines syntactic, control-flow, and data-flow information, enabling the model to learn dependencies among program components more comprehensively. GCN is a common GNN architecture capable of aggregating information from neighboring vertices to learn structural representations of graphs.

Nevertheless, most existing studies focus separately on semantic representation using Transformers or structural representation using GNNs, and therefore do not fully exploit the interaction between these two types of information. Furthermore, vulnerability detection data often exhibit considerable class imbalance between secure and vulnerable samples [14], increasing bias toward the majority class and affecting the ability to detect vulnerable samples. Consequently, combining semantic representations from Transformers, structural representations from GNNs, and a mechanism for mitigating the effects of data imbalance is a necessary approach to improving the effectiveness and generalization capability of source code vulnerability detection systems.

B. Research Motivation and Objectives

1) *Research motivation:* In recent years, deep learning methods for source code vulnerability detection have primarily developed along two directions: 1) semantic representation learning using Pre-trained Language Models (PLMs), and 2) program structural representation learning using Graph Neural Networks (GNNs) [15–18]. Models such as CodeBERT, GraphCodeBERT, CodeT5, and UniXCoder have significantly improved the semantic representation of source code. In particular, CodeT5 can learn contextual representations of tokens and statements, but it primarily processes source code as sequences and therefore does not fully exploit structural relationships such as control flow and data dependencies.

Conversely, methods based on AST, CFG, PDG, and particularly CPG combined with GCN can effectively exploit the program's structural relationships. However, the effectiveness of GCN depends substantially on the quality of the input vertex features. Methods using One-Hot Encoding, Word2Vec, FastText, or GloVe do not fully capture the contextual semantics of source code. Moreover, data imbalance remains a considerable challenge because vulnerability detection datasets often contain substantially more secure than vulnerable samples. Techniques such as Random Oversampling, SMOTE, ADASYN, and GAN can reduce imbalance but still entail risks of generating redundant samples, distorting data distributions, or causing overfitting.

These limitations indicate a remaining research gap in jointly exploiting deep semantic representations, program structural representations, and mitigation of data imbalance during representation learning. Therefore, this study proposes an architecture combining a Transformer and GCN, in which CodeT5 learns semantic features, CPG–GCN learns structural features, and a feature-rebalancing mechanism based on BDC is applied to improve generalization and source code vulnerability detection performance.

2) *Proposed objectives and solution direction:* To address the limitations of existing methods, this study proposes the SGRL Framework, a representation learning architecture that combines a Transformer and a graph neural network to exploit both semantic and structural information in source code. CodeT5 learns semantic representations from normalized source code, whereas CPG combined with GCN learns program structural representations. The two feature sources are then fused to construct a unified representation space for vulnerability detection.

Specifically, source code is first normalized to reduce noise and remove unnecessary components such as comments, redundant whitespace, and symbols that have no value for the representation process. The normalized source code is fed into CodeT5 to obtain fixed-dimensional semantic representations. Simultaneously, the source code is converted into a Code Property Graph (CPG), in which vertices represent program components and edges describe syntactic, control-flow, and data-flow relationships. Graph features are propagated and aggregated through GCN layers to learn program structural representations.

C. Principle of Source Code Vulnerability Detection of the SGRL Framework

Recent studies indicate that exploiting semantic or structural features in isolation is insufficient for a comprehensive source code representation because programs simultaneously contain semantic, syntactic, control-flow, and data-flow information. Therefore, combining the two types of representations is necessary to improve vulnerability detection effectiveness.

On this basis, this study proposes the SGRL Framework, which combines CodeT5 and GCN to jointly learn semantic and structural features of source code. CodeT5 extracts semantic representations, whereas GCN learns structural representations from CPG. The two feature vectors are fused by concatenation to form a combined representation space used as input to the vulnerability classification stage.

D. Contributions of the Study

Based on the limitations of existing methods, this study proposes the SGRL Framework, which combines a pre-trained language model and a graph neural network to jointly exploit semantic and structural features while mitigating the effects of data imbalance. The main contributions are as follows:

First, an architecture combining CodeT5 and GCN is proposed to jointly learn semantic features from source code and structural features from CPG, thereby constructing a more comprehensive representation for vulnerability detection.

Second, a parallel representation learning process is developed in which CodeT5 learns context-rich semantic representations, whereas GCN learns structural relationships through CPG, including syntactic, control-flow, and data-flow information.

Third, a feature-fusion mechanism based on concatenation is proposed to preserve both representation sources before they are passed to the classifier, thereby enhancing the ability to exploit relationships between semantic and structural information.

Fourth, this study proposes a new data-balancing method called BDC, based on generating data for the minority class through the Bernoulli distribution in the Dropout layer of a CNN. BDC balances the minority and majority classes in the experimental dataset. Applying BDC after feature fusion aims to enhance the model's generalization capability under imbalanced-data conditions. Unlike traditional data generation methods, BDC acts as an overfitting-reduction technique: this study exploits random feature removal according to a Bernoulli distribution to generate multiple representation combinations during training. This approach helps limit the model's dependence on a few dominant features while improving representation learning for minority-class samples without using synthetic sample generation techniques such as SMOTE [19] or GAN [20].

Fifth, experiments are conducted on the Verum dataset using different token-length and BDC-probability configurations. The best result is obtained with 512 tokens and BDC = 0.2, yielding an accuracy of 84.23%, Precision of 84.55%, Recall of 84.23%, and an F1-score of 84.14%, demonstrating the effectiveness of combining semantic–structural representations with a feature-adjustment mechanism for source code vulnerability detection.

II. RELATED WORKS

A. Rule-Based Source Code Vulnerability Detection

According to Chen et al. [21], rule-based vulnerability detection is a traditional approach commonly used in static source code analysis tools. This method constructs rules or source code patterns from expert knowledge and known vulnerability types, and then compares source code against the rule set to identify indicators of potential information security risks [22]. Representative tools such as Flawfinder, Cppcheck, Clang Static Analyzer, SonarQube, and Fortify SCA typically rely on secure coding rules and knowledge bases such as CWE [23] and CERT C/C++, enabling detection of several common vulnerability types, including buffer overflows, invalid pointer use, and memory leaks.

In addition, Liu et al. [3] proposed VulMatch to detect recurring vulnerabilities (1-day and N-day) by combining source code and binary information. Experiments on open-source projects showed that VulMatch performed better than some binary matching methods, such as Asm2vec [24] and Palmtree [25], while also providing explainable detection results. However, the method still encounters difficulties when processing large-scale, complex source code or incompletely labeled data.

In general, rule-based methods offer interpretability and effectiveness for known vulnerability types, but are limited in generalization and the detection of new vulnerability types. Detection effectiveness depends substantially on the completeness of the rule set and typically requires manual updates when new vulnerability types or programming techniques emerge. Furthermore, rules are primarily based on syntactic patterns and therefore do not fully exploit program semantics, control flow, and data flow, resulting in risks of false alarms and missed vulnerabilities, particularly for source code with complex structures. These limitations have encouraged the development of machine learning and deep learning methods capable of automatically learning representations and generalizing vulnerability features from data.

B. Source Code Vulnerability Detection Based on Natural Language Processing

Advances in machine learning and deep learning have introduced new approaches to source code vulnerability detection, enabling models to learn features automatically from data rather than relying on manually constructed rules. Early methods employed Bag-of-Words (BoW), TF-IDF, Word2Vec [26, 27], and FastText in combination with algorithms such as SVM, Random Forest, or XGBoost. However, these representations primarily reflect statistical relationships among tokens and do not fully exploit contextual information or dependencies within programs.

The emergence of source code language models such as CodeBERT, CodeGPT, and CodeT5 has substantially improved semantic representation learning. Ding et al. [28] showed that evaluation results for source code language models may be affected by duplication and data leakage, while these studies have not fully addressed complex structures and relationships beyond the scope of a single function. Kalouptoglou et al. [29] further investigated transfer learning with CodeGPT and

CodeBERT, showing that token-level contextual representations are effective; however, these methods still primarily exploit sequential representations and do not directly integrate structural information such as control flow and data dependencies.

Recently, Zhang et al. [30] proposed CodeSAGE, which combines Code Attribute Graph–GraphSAGE with BiLSTM and Attention to jointly exploit structural and semantic features. Although positive results were achieved, the method has not exploited source code-specific pre-trained language models such as CodeT5 or directly fused semantic representations from Transformers with structural representations from CPG/GCN. These limitations highlight the need to develop models capable of deeper integration between program semantics and structure, providing a basis for the approach proposed in this study.

C. Source Code Vulnerability Detection Based on Graph Structure

To overcome the limitations of methods that exploit only sequential representations, numerous studies have shifted toward graph-based source code representations, particularly AST, CFG, PDG, and CPG. AST describes syntactic structure; CFG represents control flow; and PDG models data and control dependencies. Yamaguchi et al. [13] proposed the Code Property Graph (CPG), which jointly models syntactic information, control flow, and data dependencies within a unified graph structure, thereby providing a more comprehensive representation of program structure and behavior.

On this basis, GNN architectures such as GCN, GGNN, GAT, and Graph Transformer have been applied to learn structural representations for vulnerability detection. Liu et al. [31] proposed Vul-LMGNNs, reporting positive results on the REVEAL and SARD datasets compared with Devign [32], LineVD [33], and FUNDED [34]. Subsequent studies, including iGnnVD [35], Adaptive GNN [36], and Heterogeneous GNN [37], further exploited diverse relationships and features in program graphs. In addition, VulMPFF [38] combines source code sequence information, lexical–syntactic relationships, and graph structure, achieving improved results over several existing methods. Studies by Cong et al. [39, 40] also indicate that combining Transformers and GNNs has the potential to improve representation learning by jointly exploiting semantic and structural source code information.

However, GNN-based methods still exhibit several limitations. First, representation learning effectiveness depends considerably on the quality of initial node features; methods using One-Hot Encoding, Word2Vec, or GloVe do not fully reflect the contextual semantics of source code. Second, many studies focus on graph-based information propagation without effectively exploiting pre-trained language models to enhance node-level semantic representations. Consequently, such models may capture structural relationships well but remain limited in distinguishing code segments with similar structures but different semantics and security properties. This indicates the need to combine pre-trained language models with GNNs to construct more comprehensive semantic–structural representations for source code vulnerability detection.

III. PROPOSED MODEL

A. Architecture of the SGRL Model

Overall, the SGRL Framework consists of five main stages: 1) source code normalization; 2) semantic representation learning; 3) structural representation learning; 4) feature fusion and rebalancing; and 5) vulnerability classification. The entire processing pipeline is illustrated in Fig. 1.

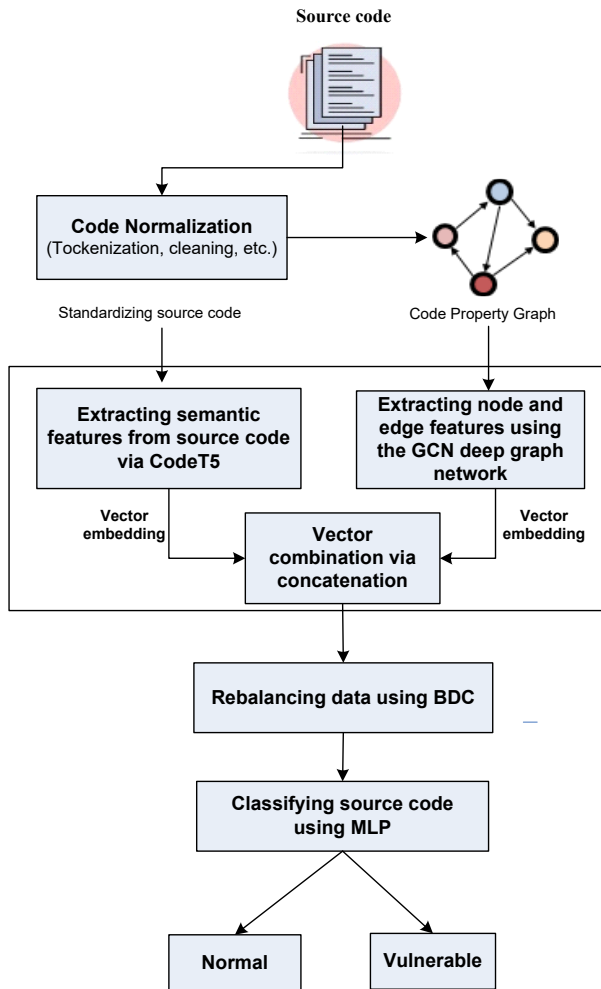


Fig. 1. Overall architecture of the SGRL framework.

The functions, tasks, and operating process of the blocks in the SGRL model are as follows:

- “Source code”: The source code dataset used is Verum [41], written in C/C++.
- “Code Normalization” block: Removes redundant data and comments, searches for and checks erroneous source code segments, and removes them from the source code.
- “Code Property Graph” block: Analyzes and transforms the source code into CPG form using the Joern tool [42]. Joern analyzes source code in terms of syntax and semantics to represent it as a CPG graph.

- “Extracting semantic features from source code via CodeT5” block: Extracts semantic features of the source code, including features of individual nodes or of the entire code segment.
- “Extracting node and edge features using the GCN deep graph network” block: Extracts source code features from the CPG graph.
- “Vector combination via concatenation” block: Fuses two feature vectors—semantic source code features extracted using CodeT5 and structural features extracted from the graph using GCN—through concatenation.
- “Rebalancing data using BDC” block: After the source code feature profiles are obtained, the authors observe a very large difference between the numbers of normal source code features and vulnerable source code features. Immediate classification may result in inaccurate predictions or, in some cases, model overfitting. Therefore, this study proposes generating data from the minority class to balance the experimental dataset using BDC. The result is a new dataset containing source code feature profiles balanced between the vulnerable and non-vulnerable labeled classes.
- “Classifying source code using MLP” block: Detects and classifies source code vulnerabilities based on the features constructed by the “Vector combination via concatenation” block. The author proposes using MLP for source code vulnerability classification.

The above describes the operation of the blocks in Fig. 1. The following sections describe the model's source code vulnerability detection task in detail.

B. Data Preprocessing

1) *Source code normalization*: In the first stage, source code is preprocessed and normalized to remove redundant components such as comments, whitespace, and unnecessary characters; at the same time, variable names, function names, and parameters are standardized according to a unified rule. This process helps reduce vocabulary diversity, limit sparse data, and improve the model's generalization capability. Preprocessing is necessary because raw source code contains many symbols, complex syntax, and differences in programming style, making the representation learning process difficult.

2) *Representing source code as a CPG graph*:

a) *Introduction to CPG*: CPG, introduced by Fabian Yamaguchi et al. in 2014 [13], combines AST, CFG, and PDG into a common data structure consisting of nodes and node types representing source code components such as classes and methods; labeled directed edges representing relationships among components; and key-value pairs containing node attributes, where valid keys depend on the node type. In this study, the author proposes normalizing and converting source code into CPG form to facilitate feature extraction.

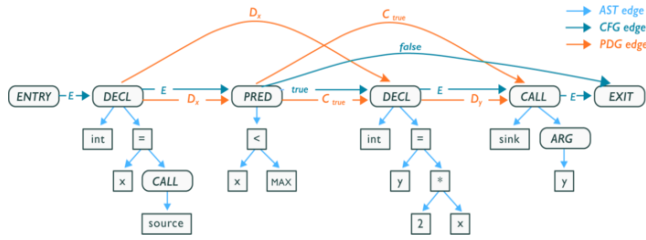


Fig. 2. Representation of source code according to CPG.

Fig. 2 shows that CPG is the combination of AST, CFG, and PDG, helping provide more important information for source code feature extraction because these graphs describe both static features (variable names, functions, etc.) and their relationships.

b) *Introduction to the Joern platform:* Joern [43] is a platform commonly used today in static analysis for source code, including C/C++ source code. It enables vulnerability detection by converting source code into a CPG through nodes (functions, statement blocks, etc.) to describe source code features. This study applies the Joern tool to represent C/C++ source code as a CPG graph consisting of nodes containing syntactic code snippets and syntactic edges (AST), control edges (CFG, PDG), thereby expressing all properties of the source code.

C. Source Code Feature Extraction

From Fig. 1, combining representation learning using a Transformer model and a graph neural network to jointly exploit semantic and structural source code information is an approach intended to maximize the use of different source code feature components. After normalization, the source code is processed in parallel along two directions: 1) the first branch passes it through CodeT5 to learn semantic representations at the token and statement levels; 2) the second branch represents the normalized source code as a CPG graph and then learns graph features using GCN. Regarding the second branch, studies [13, 40, 41, 43] have listed the vertex and edge features of CPG. CPG is a particularly useful representation of original code because it provides a compact, integrated representation comprising elements of control-flow and data-flow graphs, in addition to AST and the Program Dependence Graph (PDG). Each of these elements provides additional context about the overall semantic structure of the code [13]. Formally, CPG is denoted by $G = (V, E)$, where V represents graph vertices (or nodes) and E represents edges. Each vertex in CPG comprises a vertex type (e.g., ArithmeticExpression or CallStatement) and a segment of original code. Each edge contains information about its edge type (e.g., IS_AST_PARENT or FLOWS_TO) [42].

The semantic and structural features extracted by CodeT5 and GCN, respectively, are fused through concatenation to form a combined feature vector for the subsequent classification stage. This is an entirely new proposal for source code feature extraction that uses a model combining semantic and structural representation learning, rather than focusing on a single feature type as is currently common. With this new proposal, important source code features can be extracted more comprehensively. The following sections describe the source code feature extraction process based on CodeT5 and GCN in detail.

1) Extracting semantic features of source code using CodeT5

a) *Overview of CodeT5:* In 2021, Wang et al. proposed CodeT5 [9], an Encoder-Decoder Transformer model pre-trained specifically for source code. Unlike previous models such as CodeBERT, which use only an Encoder architecture, CodeT5 is designed to support both code understanding and code generation tasks through a unified architecture. In particular, CodeT5 introduces an Identifier-aware Pre-training mechanism, allowing the model to learn the semantic roles of identifiers in programs, thereby significantly improving the quality of source code representation across many tasks such as bug detection, vulnerability detection, clone detection, and code generation.

After that, CodeT5 uses the Multi-Head Self-Attention mechanism to model relationships among all tokens in the code segment. For each attention head, the three matrices Query, Key, and Value are computed as follows:

$$Q = XW_Q, K = XW_K, V = XW_V, \quad (1)$$

where, W_Q , W_K , and W_V are weight matrices to be learned. The attention weight is determined according to Eq. (2) below:

$$Attention(Q, K, V) = Soft \max \left(\frac{QK^T}{\sqrt{d_k}} \right) V, \quad (2)$$

where, d_k is the dimension of the key vector.

b) *Method for extracting semantic features of source code using CodeT5:* After normalization, the source code is simultaneously fed into two independent representation learning branches. The first branch uses CodeT5 to exploit the semantic features of the program. In this branch, CodeT5, a Transformer model pre-trained on a large volume of multilingual source code, can learn contextual representations that reflect the meaning of each token as well as the relationships among statements in the program. The output of CodeT5 is a fixed-dimensional semantic representation vector that reflects the content and context of the entire source code segment.

It can be said that the process of extracting semantic features of source code using CodeT5 is described in the following figure:

From Fig. 3, it can be seen that the semantic feature extraction process is performed through the following steps:

Step 1: After normalization, each code segment is represented as a token sequence according to Eq. (3) below:

$$S = \{t_1, t_2, \dots, t_L\}, \quad (3)$$

Step 2: Here, L is the number of tokens after normalization. The token sequence is fed into the encoder of CodeT5. Each token is first mapped into an embedding vector in a d -dimensional feature space, as shown in Eq. (4):

$$E = \{e_1, e_2, \dots, e_L\}, e_i \in \mathbb{R}^d. \quad (4)$$

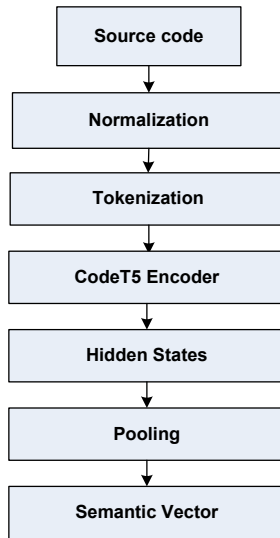


Fig. 3. The process of extracting semantic features of source code using CodeT5.

Step 3: To preserve information about the order of token occurrence, the positional vector is added to the input embedding vector:

$$X = E + P, \quad (5)$$

where, P is the positional embedding matrix.

Thanks to this mechanism, the representation of each token is updated based on the context of the entire code segment instead of depending only on neighboring tokens. This is especially useful in vulnerability detection, where the cause of a vulnerability may arise from interactions among statements located far apart. The output of the encoder is the semantic representation matrix shown in Eq. (6) below:

$$H_s = \{h_1, h_2, \dots, h_L\} \in \mathbb{R}^{L \times d}, \quad (6)$$

where, each vector h_i contains the contextual semantic information of the i -th token.

Step 4: To obtain a fixed representation for the entire code segment, pooling can be applied, for example, averaging over all tokens according to Eq. (7):

$$Z_s = \frac{1}{L} \sum_{i=1}^L h_i, \quad (7)$$

where, $Z_s \in \mathbb{R}^d$ is the semantic feature vector of the code segment.

In the SGRL model, the vector Z_s is not used directly for classification but serves as one representation learning branch. In parallel with semantic representation learning using CodeT5, the other branch, also from the normalized source code segments, is represented through a CPG graph and then fed into the GCN network to extract the structural features of the program. The two feature vectors obtained from these two representation learning branches are fused through Concatenation, forming a multi-source representation that integrates both semantic information and structural information. This fused representation serves as the input for the subsequent

stages of the model to improve the ability to detect vulnerabilities in source code.

2) Extracting source code features using GCN

a) Overview of GCN: GCN, developed by Thomas Kipf and Max Welling [32], is a variant of Graph Neural Networks (GNNs). A characteristic of GCN is that it uses local spectral filters on graphs to extract graph substructures and thereby reveal graph structure. In terms of its operating principle, the convolution operation in a GCN layer extracts features from neighboring components, similarly to a CNN layer [32, 44]. However, because CNNs do not work well on non-Euclidean datasets, GCN architectures seek to extract attributes from adjacent vertices. GCN is also applied in several fields, such as node classification [45], link prediction [46], graph classification [47], and graph embedding [44]. Eq. (8) below presents the propagation of representative attributes in a GCN layer.

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) + H^{(l)} \quad (8)$$

Where:

A : The adjacency matrix of the graph,

X : The input attribute matrix of the node,

$\tilde{D}$: the degree matrix corresponding to $\tilde{A}$,

I : The identity matrix of size $N \times N$,

$$\tilde{A} = A + I,$$

$\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$: The symmetrically normalized combination of the adjacency matrix,

$W^{(l)}$: the learned weight matrix of layer l ,

σ : A nonlinear function applied element-wise,

$H^{(l)}$: The output of layer l ,

$$H^{(0)} = X.$$

b) Method for extracting features of the CPG graph based on GCN: In this study, the author selects three main components that describe the structure and information of the CPG graph and also serve as the three inputs to the proposed training model: the adjacency matrix, the node attribute matrix, and the edge attribute matrix. This information is then fed into the GCN model for processing, whose general architecture is shown in Fig. 4.

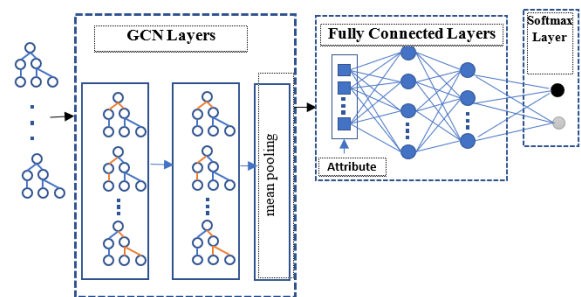


Fig. 4. Architecture of the GCN model.

From Fig. 4, it can be seen that the GCN network architecture performs feature extraction as follows:

- GCN layer: includes GCN layers and a pooling layer to extract graph features. Thus, with the behavior profile as input, the GCN network extracts the characteristics of the graph linking the processes. The output of the GCN layer is hidden representative attributes that show the connections among vertices in the graph through GCN layers. For example, in the case of using two GCN layers, the first GCN layer has the task of considering neighboring vertices of each vertex to update the attributes of that vertex. Next, the second GCN layer recursively updates the attributes of the vertices in the graph. In practice, more than two GCN layers can be used; however, study [44] indicated that only two GCN layers may still ensure a balance between effectiveness and analysis and processing time.
- Fully connected layer: this layer is similar to an MLP network and has the task of learning the attributes processed through the GCN layers.
- Softmax layer: this layer has the task of computing the probabilities of the output labels. The file being predicted is assigned to the class with the highest probability.

3) *Building the source code feature vector*: After the representation learning stage, the model obtains two complementary feature vectors, including semantic features from CodeT5 and structural features from GCN on CPG. The semantic features reflect the context and relationships among tokens, while the structural features represent syntactic relations, control flow, and data flow. Combining these two feature sources creates a more comprehensive source code representation, effectively supporting the vulnerability detection process.

Assume that $Z_s \in \mathbb{R}^{d_s}$ is the semantic feature vector created by CodeT5 and Z_g is the structural feature vector generated by GCN. The source code feature vector is built by concatenation along the feature dimension as follows:

$$Z = \text{Concat}(Z_s, Z_g) = [Z_s; Z_g], \quad (9)$$

Where:

- d_s is the dimension of the semantic feature vector;
- d_g is the dimension of the structural feature vector;
- $Z \in \mathbb{R}^{d_s+d_g}$ is the fused feature vector.

Thus, Concatenation is used to fuse both semantic and structural features of source code. Compared with addition or averaging, Concatenation preserves complete information from the two representation spaces without losing their distinctive features. The fused vector is then fed into subsequent learning layers so that the model can automatically determine the contribution level of each feature type during optimization.

D. Data Rebalancing

After constructing the feature profiles, direct classification may be affected by data imbalance because the numbers of samples in the vulnerable and non-vulnerable classes differ substantially. Traditional methods such as SMOTE and GAN can be used to balance data, but sample generation based on neighborhood relationships in feature space is not entirely suitable for the characteristics of source code data. Therefore, this study proposes a rebalancing mechanism based on BDC to increase the diversity of minority-class representations and reduce the impact of data imbalance during training. Accordingly, for the vector representation, each component becomes: $x = (x_1, x_2, \dots, x_m)x_h (h = 1, 2, \dots, m)$

$$\hat{x}_h = a_h \cdot x_h \quad (10)$$

where, $a_h \sim P$ is the variable is a random variable with a Bernoulli distribution [48]:

$$P(a_h) = \begin{cases} 1-p, & a_h = 0 \\ p, & a_h = 1 \end{cases} \quad (11)$$

It can be seen that, in the problem of generating data for the minority class (i.e., the vulnerable class), the Bernoulli distribution will:

- Generate a random value according to the Bernoulli distribution
- Return a result of 0 (non-vulnerable) or 1 (vulnerable)
 - With probability p , it takes the value 1
 - With probability $1-p$, it takes the value 0

The algorithm below describes the data generation process using BDC.

Algorithm 1: Data balancing using BDC

Input: Feature -*feature*

Label -*label*

Rate - α

Number of generated samples based on minority class - n

$D = \{\}$

Output: $D_{balance}$

Function: $BDC(feature, label, \alpha = 0.1, n = 10)$:

for $f_i, l_i \in feature, label$ do:

if l_i of minority class do:

for $t = 1$ to n do:

$f'_i \leftarrow generate_minority_class_with_rate(\alpha)$

Add f'_i in D

end

end

end

return $D_{balance}$

end

From Algorithm 1, it can be seen that:

Step 1: Search for the features of the minority class

Step 2: Iterate through the features of the minority class and apply the Bernoulli generation function to generate data for the minority class.

Step 3: Add the generated data to the original data

E. Source Code Vulnerability Classification

After the representation learning stage, the model obtains a fused feature vector integrating semantic information from CodeT5 and structural information from GCN. This vector is fed into an MLP to classify the source code into two classes: vulnerable and non-vulnerable. MLP learns a nonlinear mapping function from the feature space to the label space, thereby exploiting complex relationships among features and determining the class of the input sample.

Assume that the fused representation vector after the representation learning stage is denoted as:

$$h^* \in \mathbb{R}^d \quad (12)$$

where, d is the feature-space dimension. This input is fed through the hidden layers of the MLP. At the l -th hidden layer, the output is determined according to the equation:

$$h^{(l+1)} = \Phi(W^{(l)}h^{(l)} + b^{(l)}), \quad (13)$$

Where:

- $W^{(l)}$ is the weight matrix of the l -th layer;
- $b^{(l)}$ is the bias vector;
- $\Phi(\cdot)$ is the nonlinear activation function; in this study, the ReLU function is used and defined as:

$$\Phi(x) = \max(0, x) \quad (14)$$

Using multiple hidden layers helps the model learn nonlinear relations between semantic features and structural features, thereby increasing the ability to distinguish vulnerable source code samples from normal source code samples.

After the last hidden layer, the feature vector is mapped to the output layer, whose number of nodes equals the number of classes to be classified. In the source code vulnerability detection problem, the number of output classes is two, corresponding to two labels:

$$y = \{0, 1\} \quad (15)$$

Where:

- 0: the code segment does not contain a vulnerability;
- 1: the code segment contains a vulnerability.

The output logits of the model are computed according to the equation:

$$o = W_o h^{(L)} + b_0 \quad (16)$$

where, W_o are b_0 are the weight and bias of the output layer, respectively.

To convert the logits into predicted probabilities for each class, the model uses the softmax function:

$$P(y = i|x) = \frac{\exp(o_i)}{\sum_{j=1}^2 \exp(o_j)}, i \in \{0, 1\}, \quad (17)$$

Where:

- o_i : is the logit of the i -th class;
- $P(y = i|x)$: is the probability that the code segment belongs to class i .

The final predicted label is determined according to the maximum-probability criterion:

$$\hat{y} = \arg \max_i P(y = i|x) \quad (18)$$

During training, the parameters of the entire model are optimized by minimizing the Cross-Entropy loss function, defined as follows:

$$\mathcal{L} = - \sum_{i=1}^2 y_i \cdot \log P(y = i|x) \quad (19)$$

Where:

- y_i : is the actual label of the i -th class;
- $P(y = i|x)$: is the probability predicted by the model.

The optimization process is performed by the Adam algorithm, in which the model parameters are updated in the direction of decreasing the value of the loss function on the training dataset. Thanks to its effective optimization capability for deep learning models, Adam helps speed up convergence and improve stability during training.

Through the combination of semantic representation from CodeT5, structural representation from GCN, and the MLP classifier, the model can learn distinguishing features of vulnerable samples even when they appear in many different forms in source code. At the same time, using an MLP allows the model to effectively exploit nonlinear interactions among the fused features, contributing to improving the accuracy of the source code vulnerability detection problem.

IV. EXPERIMENTS AND EVALUATION

A. Experimental Dataset

Table I presents statistics on the experimental datasets proposed for use in the study.

TABLE I. EXPERIMENTAL DATA STATISTICS

	Total number of samples	Number of vulnerabilities	Number of normal samples
Verum Dataset [41]	18169	1658	16511
FFmpeg+Qume Dataset [49]	25905	11804	14101

The Verum dataset [41] is built from open-source projects with CVE-coded vulnerabilities [50], in which code regions

containing vulnerabilities are labeled as vulnerable and safe regions are labeled as non-vulnerable. Verum supports research on vulnerability detection in C/C++ source code. Study [41] shows that this dataset has a large imbalance between security vulnerability data and normal data, with a small code duplication rate of about 0.6%, and is therefore suitable for the code vulnerability classification task.

To demonstrate the effectiveness of the proposed model, the study evaluates it using the two main datasets summarized in Table I.

B. Experimental Scenarios and Evaluation Criteria

1) *Experimental scenarios*: To demonstrate the effectiveness of the proposed model, the author conducts the following experimental scenarios:

a) *Scenario 1*: How effective is the author's proposed method in classifying source code vulnerabilities?

b) *Scenario 2*: Evaluation of the role of the individual components in the author's proposed model. Through this, it is confirmed whether the author's proposed method is more effective than the separate approach of each component in the proposed model on the same experimental dataset.

c) *Scenario 3*: Comparing the SGRL model with several other approaches on the same experimental dataset.

2) *Evaluation criteria*: To evaluate effectiveness, the study uses several metrics:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100\% \quad (20)$$

$$precision = \frac{TP}{TP + FP} \times 100\% \quad (21)$$

$$Recall = \frac{TP}{TP + FN} \times 100\% \quad (22)$$

$$F1 - score = \frac{2 \times precision \times Recall}{precision + Recall} \times 100\% \quad (23)$$

Where:

- Accuracy: The proportion of correctly classified samples over the total number of samples. Accuracy is calculated according to (20).
- Precision: The proportion of TP over (TP + FP), reflecting precision when predicting positive (21).
- Recall: The proportion of TP over (TP + FN), indicating the ability to correctly detect positive samples (22).
- F1-score: The harmonic mean of Precision and Recall; the higher it is, the better the classifier (23).

C. Experimental Result on the Proposed Model

1) *Experimental results of Scenario 1*: In this experimental scenario, the author conducts experiments on the Verum dataset. Through the dataset description in Section IV, Part A, study [41] also confirmed that this dataset is very suitable for application to security vulnerability classification tasks. Table II below presents statistics on the effectiveness of the proposed method in security vulnerability detection based on several criteria defined in Section IV-B-2.

TABLE II. EXPERIMENTAL RESULTS OF SCENARIO 1

BDC	GCN_CodeT5											
	128				256				512			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
0.1	81.85	81.95	81.85	81.79	82.59	82.67	82.59	82.55	83.48	83.81	83.48	83.39
0.2	82.59	82.74	82.59	82.53	83.04	83.18	83.04	82.98	84.23	84.55	84.23	84.14
0.3	82.29	82.41	82.29	82.24	82.89	83.04	82.89	82.83	84.08	84.34	84.08	84.0

From the experimental results of Scenario 1 in Table II, the model combining GCN and CodeT5 with an input length of 512 tokens and a BDC probability of 0.2 is the most suitable configuration among the parameters investigated. This configuration achieves an accuracy of 84.23%, Precision of 84.55%, Recall of 84.23%, and F1-score of 84.14%, attaining the highest values for all four evaluation metrics. The results indicate that increasing the token sequence length to 512 enables the model to exploit the semantic context of source code more effectively, while a BDC probability of 0.2 provides an appropriate degree of regularization for representation learning. Increasing the BDC probability further to 0.3 leads to a slight decline in performance, indicating that excessive feature removal may discard useful information. Therefore, combining a token length of 512 with BDC = 0.2 is the optimal configuration, yielding the best and most consistent results across all the evaluation metrics of the proposed model. This is also scientifically reasonable because, according to the survey

presented in this study and the study by Chakraborty et al. [41], as many as 70% of the source code files in the Verum dataset have lengths greater than 256 and less than 512. Accordingly, selecting a token length of 512 yields the best model performance.

Fig. 5 presents the confusion matrix calculated for the proposed model with a token length of 512 and a BDC probability of 0.2.

Thus, although the Verum dataset exhibits severe class imbalance, with secure source code accounting for approximately 90.88%, the SGRL model under the 512/0.2 configuration demonstrates superior identification capability by correctly detecting 1,397 of the 1,658 actual vulnerable samples (Recall = 84.23%). Meanwhile, the number of false positives (FP) remains very low (255 samples out of a total of 16,511 secure samples), maintaining Precision at 84.55% and minimizing the cost of manual code inspection in practice.

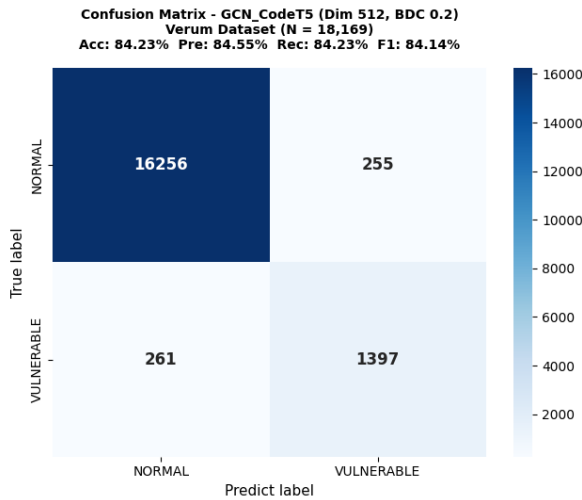


Fig. 5. Confusion matrix values of the proposed SGRL model.

2) *Experimental results of Scenario 2:* In this scenario, to demonstrate the effectiveness of the proposed model, the author

conducts experiments on each component of the proposed model to prove that the combination of the GCN deep graph network with CodeT5 gives the best result.

First, the author conducts an experiment using only CodeT5 to extract source code features, combined with BDC to rebalance the vulnerable and non-vulnerable labels in the experimental dataset. The resulting features are then passed through a Multi-Layer Perceptron (MLP) feedforward neural network to classify source code as vulnerable or non-vulnerable.

Table III shows that, when the model is evaluated using individual components—in this case, using only CodeT5 for source code feature extraction—the resulting architecture performs considerably worse than the model proposed in this study. Specifically, at the same configuration of a 512-token input length and BDC probability of 0.2, which yields the best results among the investigated token lengths of 128, 256, and 512 and their corresponding BDC probabilities, the Accuracy, Precision, Recall, and F1-score are lower than those of the proposed model by 0.9%, 1.11%, 0.9%, and 0.85%, respectively.

TABLE III. EXPERIMENTAL RESULTS WHEN USING ONLY THE CODET5 MODEL FOR SOURCE CODE FEATURE EXTRACTION

BDC	CodeT5											
	128				256				512			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
0.1	81.55	81.56	81.55	81.53	82.14	82.15	82.14	82.15	82.44	82.49	82.44	82.41
0.2	81.55	81.56	81.55	81.53	82.14	82.15	82.14	82.15	83.33	83.44	83.33	83.29
0.3	81.55	81.56	81.55	81.53	82.14	82.15	82.14	82.15	82.44	82.49	82.44	82.41

In the second case, the same configuration is retained, but the source code feature extraction method is changed by replacing the combined GCN and CodeT5 components with the

CodeBERT model. Experiments using token lengths of 128, 256, and 512 and corresponding BDC probabilities of 0.1, 0.2, and 0.3 yield the results shown in Table IV below:

TABLE IV. EXPERIMENTAL RESULTS WHEN USING ONLY THE CODEBERT MODEL FOR SOURCE CODE FEATURE EXTRACTION

BDC	CodeBERT											
	128				256				512			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
0.1	82.59	83.04	82.59	82.47	83.93	84.25	83.93	83.84	82.44	82.91	82.44	82.31
0.2	82.74	83.37	82.74	82.58	82.44	82.82	82.44	82.33	92.59	82.91	82.95	82.49
0.3	82.44	82.78	82.44	82.34	82.74	82.93	82.74	82.67	82.59	82.6	82.59	82.59

Table IV indicates that the best result is obtained with a token length of 512 and a corresponding BDC probability of 0.2. However, when GCN combined with CodeT5 is used for source code feature extraction, the results are considerably better in terms of Precision, Recall, and F1-score, by 1.64%, 1.28%, and 1.65%, respectively. Only the Accuracy of CodeBERT when used independently is approximately 8.36% higher than that of the author's proposed GCN–CodeT5 model. However, higher Accuracy alone is insufficient to establish that a model has better vulnerability detection capability. For vulnerability detection, the ability to correctly identify vulnerable samples, as reflected by Recall, is particularly important. The proposed model combining GCN and CodeT5 achieves Recall of 84.23% and

F1-score of 84.14%, which are higher than those of CodeBERT by 1.28% and 1.65%, respectively, indicating a better balance between detecting vulnerable samples and limiting incorrect predictions. Thus, although CodeBERT achieves higher Accuracy, the model combining GCN and CodeT5 demonstrates a more balanced performance and is better suited to the objective of source code vulnerability detection.

In the third case, only the GraphCodeBERT model is used to replace the GCN and CodeT5 components in the author's proposed model. With experimental token lengths of 128, 256, and 512 and corresponding BDC probabilities of 0.1, 0.2, and 0.3, the results shown in Table V below are obtained:

TABLE V. EXPERIMENTAL RESULTS WHEN USING ONLY THE GRAPHCodeBERT MODEL FOR SOURCE CODE FEATURE EXTRACTION

BDC	GraphCodeBERT											
	128				256				512			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
0.1	83.78	84.16	83.78	83.68	82.14	82.76	82.14	81.98	83.63	83.81	83.63	83.57
0.2	83.78	84.16	83.78	83.68	82.14	82.76	82.14	81.98	83.48	83.67	83.48	83.42
0.3	83.78	84.16	83.78	83.68	82.14	82.76	82.14	81.98	83.48	83.7	83.48	83.41

From Table V, it can be clearly seen that the replacement model GraphCodeBERT gives poorer classification results than the model proposed in the study. Specifically, the accuracy, precision, recall, and F1-score are lower than those of the proposed model by 0.75%, 0.88%, 0.75%, and 0.72%, respectively. This result demonstrates the superiority of the combined CodeT5 and GCN model for the source code feature extraction task, proving the correctness and reasonableness of the authors' proposed model.

3) *Experimental results of Scenario 3*: Comparison of the SGRL model with several other approaches.

This scenario aims to evaluate and compare the SGRL model with several other studies on different experimental datasets. The two datasets used are Verum and FFmpeg+Qume, representative datasets currently widely employed in source code vulnerability detection. The Verum dataset is used in Scenarios 1 and 2 of this study, whereas FFmpeg+Qume is used in Scenario 3 to demonstrate that the SGRL model performs well not only on one dataset but also on different datasets. The experimental results for Scenario 3 are presented in Table VI.

The experimental results in Table VI show that the author's proposed method achieves substantially better results on most metrics. In this study, the author reuses the experimental results reported in study [41] to compare the proposed approach with

REVEAL [41], Russell [51], VulDeePecker [52], SySeVR [55], and Devign [56]. Comparing the experimental results in Table VI with those in Table II for the Verum dataset also indicates that the proposed method performs substantially better than the other approaches. Specifically, compared with REVEAL [41], the author's proposal achieves higher Precision, Recall, and F1-score by 50.52%, 19.99%, and 39.65%, respectively; only its Accuracy is 2.71% lower than that of REVEAL. Similarly, the results of Russell [51], VulDeePecker [52], SySeVR [55], and Devign [56] are lower than those of the author's method across all metrics, ranging from a minimum difference of 66.87% in Precision to a maximum difference of 70.36% in Recall. On the FFmpeg+Qume dataset [49], the author's proposal achieves substantially better results on all metrics than the other methods, including Russell [51], VulDeePecker [52], SySeVR [55], and Devign [56], evaluated on the same FFmpeg+Qume experimental dataset. Specifically, the maximum relative improvements of the author's proposal over the other studies in Accuracy, Precision, Recall, and F1-score are 38.3%, 42.29%, 63.14%, and 55.53%, respectively. This is a very good result, indicating that the model proposed in this study is effective not only on one experimental dataset but also on multiple datasets. Its effectiveness is reflected in the experimental results, in which the proposed model achieves good results that exceed those of all other approaches on most metrics.

TABLE VI. COMPARISON RESULTS FOR THE SGRL MODEL AND OTHER APPROACHES ON THE VERUM AND FFMPEG+QUME DATASETS

Experimental Dataset	Approach	Accuracy	Precision	Recall	F1-score
Verum	Article proposal	84.23	84.55	84.23	84.14
	REVEAL [41]	86.94	34.03	64.24	44.49
	Russell [51]	90.98	24.63	10.91	15.24
	VulDeePecker [52]	89.05	17.68	13.87	15.7
	SySeVR [53]	84.22	24.46	40.11	30.25
	Devign [54]	88.41	34.61	26.67	29.87
FFmpeg+Qume	Article proposal	90.82	89.65	91.84	90.73
	REVEAL [41]	62.51	56.85	74.61	64.42
	Russell [51]	58.13	54.04	39.50	45.62
	VulDeePecker [52]	53.58	47.36	28.70	35.20
	SySeVR [53]	52.52	48.34	65.96	56.03
	Devign [54]	58.57	53.60	62.73	57.18

Fig. 6 presents the confusion matrix calculated for the proposed model with a token length of 512 and a BDC probability of 0.2.

For the FFmpeg+Qume dataset, which has a more balanced label distribution, the SGRL model demonstrates stability and strong generalization capability by correctly predicting 9,943 vulnerable samples and 12,285 normal samples. The false-

negative (FN) and false-positive (FP) counts remain symmetrically balanced across the main diagonal, demonstrating that combining graph structure (GCN) with the CodeT5 source code semantic representation model enables the system to process source code functions of substantial length and complexity effectively.

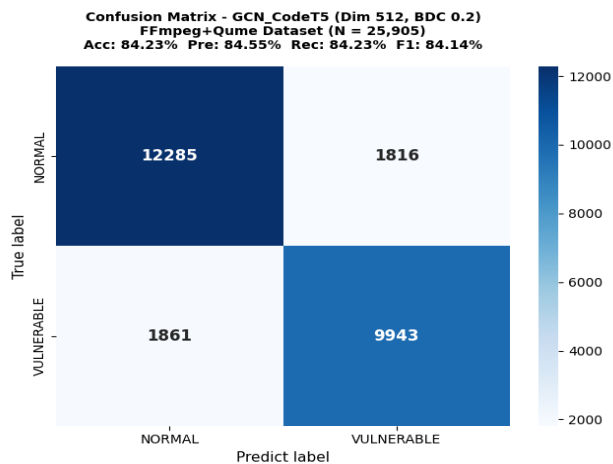


Fig. 6. Confusion matrix values of the SGRL model on the FFmpeg+Qume dataset.

V. CONCLUSION

Finally, from an academic perspective, the study contributes to expanding the research direction of multi-source representation learning for source code vulnerability detection by integrating a pre-trained language model with a graph neural network in a unified architecture. The research results not only affirm the complementary role between semantic representation and structural representation, but also provide an experimental basis for developing next-generation vulnerability detection models capable of simultaneously exploiting multiple sources of information from source code. This is also a premise for extending the SGRL Framework toward combining Graph Transformer, Graph Attention Network, or adaptive feature fusion mechanisms in subsequent studies. Although the proposed method has achieved promising results, the study still has several limitations. First, the method depends on the ability of Joern to analyze source code and construct the Code Property Graph (CPG). For source code with complex structures, specific syntactic characteristics, or features that are not fully supported by the tool, the quality of the generated CPG may be affected, thereby influencing the effectiveness of structural feature representation. Second, the proposed architecture integrates both Transformer and GCN branches to extract semantic and structural features, respectively, and may therefore require greater computational and memory resources than methods employing a single representation branch. These limitations should be considered when deploying the proposed method in real-world software development environments. In future research, the authors will focus on improving the ability to process complex source code and optimizing the architecture to reduce computational costs while maintaining the capability to simultaneously exploit semantic and structural information from source code. These efforts will also provide a foundation for extending the SGRL framework by integrating Graph

Transformer models, Graph Attention Networks, or adaptive feature fusion mechanisms in subsequent studies.

REFERENCES

- [1] G. Lin, S. Wen, Q.-L. Han, J. Zhang and Y. Xiang, "Software Vulnerability Detection Using Deep Neural Networks: A Survey," in *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825-1848, Oct. 2020, doi: 10.1109/JPROC.2020.2993293.
- [2] "The National Institute of Standards and Technology (NIST)," 2024. [Online]. Available: <https://www.nist.gov/>. [Accessed 16 9 2026].
- [3] Z. Liu, L. Pan and C. Chen et al., "VulMatch: Binary-level Vulnerability Detection Through Signature," in *Network and System Security: 18th International Conference, NSS 2024, Abu Dhabi, United Arab Emirates, November 20-22, 2024, Proceedings*, 2024, pp. 409-427, https://doi.org/10.1007/978-981-96-3531-3_20.
- [4] X. Li, L. Wang, Y. Xin, Y. Yang, Q. Tang and Y. Chen, "Automated Software Vulnerability Detection Based on Hybrid Neural Network," *Applied Sciences*, vol. 11, no. 7, p. 3201, 2021, <https://doi.org/10.3390/app11073201>.
- [5] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, Melbourne, Australia, August 2017. DOI: 10.24963/ijcai.2017/423.
- [6] J. DevlinMing, M.-W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of NAACL-HLT 2019*, p. 4171-4186, October 2018, DOI: 10.48550/arXiv.1810.04805.
- [7] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," in *International Conference on Learning Representations*, 2019, <https://arxiv.org/abs/1907.11692>.
- [8] Z. Feng, D. Guo and D. Tang et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Conference: Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, DOI: 10.18653/v1/2020.findings-emnlp.139.
- [9] Y. Wang, W. Wang, and S. Joty et al., "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP 2021)*, 2021, pp. 8696-8708, DOI: 10.18653/v1/2021.emnlp-main.685.
- [10] F. Yamaguchi, M. Lottmann and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *ACSAC '12: Proceedings of the 28th Annual Computer Security Applications Conference*, vol. 28, pp. 359-368, 2012, DOI: 10.1145/2420950.2421003.
- [11] H. Gascon et al., "Structural detection of android malware using embedded call graphs," in *ACM workshop on Artificial intelligence and security*, New York, USA, pp. 45-54, 2013, <https://doi.org/10.1145/2517312.2517315>.
- [12] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, 1987.
- [13] F. Yamaguchi, N. Golde, D. Arp and Konrad Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 18-21 May 2014, doi: 10.1109/SP.2014.44.
- [14] A. Abiri, A. Tajary and M. Fateh, "Impact of Oversampling Methods on Imbalanced Dataset for Software Fault Prediction," in *2025 15th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2025, DOI: 10.1109/ICCKE68588.2025.11273874.
- [15] Y. Guo, Q. Hu and Q. Tang et al., "An Empirical Study of the Imbalance Issue in Software Vulnerability Detection," in *Computer Security – ESORICS 2023: 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25–29, 2023, Proceedings, Part IV*, 2024, pp. 371-390, vol. 14347, https://doi.org/10.1007/978-3-031-51482-1_19.
- [16] S. S. Daneshvar, D. Tan and S. Wang et al., "A Study on Mixup-Inspired Augmentation Methods for Software Vulnerability Detection," in *EASE '25: Proceedings of the 29th International Conference on Evaluation and*

- Assessment in Software Engineering*, 2025, pp. 616-621, <https://doi.org/10.1145/3756681.3757017>.
- [17] P. Haridas, G. Chennupati and N. Santhi et al., "Code Characterization With Graph Convolutions and Capsule Networks," in *IEEE Access*, vol. 8, pp. 136307-136315, 2020, doi:10.1109/ACCESS.2020.3011909..
- [18] B. Wu, F. Zou and P. Yi et al., "SlicedLocator: Code vulnerability locator based on sliced dependence graph," *Computers & Security*, vol. 134, no. C, p. 103469, 2023, <https://doi.org/10.1016/j.cose.2023.103469>.
- [19] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, no. 1, p. 321-357, 2002.
- [20] I. J. Goodfellow, J. Pouget-Abadie and M. Mirza et al., "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 27, p. 2672-2680, 2014.
- [21] D. Chen, Y.-d. Zhang and . W. Wei et al., "Efficient vulnerability detection based on an optimized rule-checking static analysis technique," *Frontiers of Information Technology & Electronic Engineering*, vol. 18, pp. pp. 332-345, 2017.
- [22] R. Croft, D. Newlands and Z. Chen et al., "An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing," in *ESEM '21: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1-12, No. 8, 2021, <https://doi.org/10.1145/3475716.3475781>.
- [23] "CWE TOP25," 2024. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. [Accessed 16 9 2026].
- [24] S. H. Ding, B. C. Fung and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 19-23 May 2019, IEEE, pp. 472-489, DOI: 10.1109/SP.2019.00003.
- [25] X. Li, Y. Qu and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236-3251, <https://doi.org/10.1145/3460120.3484587>.
- [26] T. Mikolov, K. Chen and G. Corrado et al., "Efficient Estimation of Word Representations in Vector Space," 2013, arXiv:1301.3781.
- [27] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *Proceedings of the 31st International Conference on Machine Learning, PMLR*, vol. 32, no. 2, pp. 1188-1196, 2014.
- [28] Y. Ding, Y. Fu and O. Ibrahim et al., "Vulnerability Detection with Code Language Models: How Far Are We?," in *ICSE '25: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pp.1729-1741, 2025, <https://doi.org/10.1109/ICSE55347.2025.00038>.
- [29] I. Kaloutsoglou, M. Siavvas and A. Ampatzoglou et al., "Transfer learning for software vulnerability prediction using Transformer models," *Journal of Systems and Software*, vol. 227, p. 112448, 2025, <https://doi.org/10.1016/j.jss.2025.112448>.
- [30] G. Zhang, T. Yao and J. Qin et al., "CodeSAGE: A multi-feature fusion vulnerability detection approach using code attribute graphs and attention mechanisms," *Journal of Information Security and Applications*, vol. 89, p. 103973, 2025, <https://doi.org/10.1016/j.jisa.2025.103973>.
- [31] R. Liu, Y. Wang and H. Xu et al., "Vul-LMGNNs: Fusing Language Models and Online-Distilled Graph Neural Networks for Code Vulnerability Detection," *Information Fusion*, vol. 115, p. 102748, March 2025. <https://doi.org/10.1016/j.inffus.2024.102748>.
- [32] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations 2017 (ICLR 2017)*, 2017.
- [33] D. Hin, A. Kan and H. Chen et al., "Linevd: Statement-level vulnerability detection using graph neural networks," in *In Proceedings of the 19th international conference on mining software repositories*, pp. 596-607, 2022.
- [34] H. Wang et al., "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection," in *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943-1958, 2021, doi: 10.1109/TIFS.2020.3044773..
- [35] J. Chen, Y. Yin and S. Cai et al., "iGnnVD: A novel software vulnerability detection model based on integrated graph neural networks," *Science of Computer Programming*, vol. 238, no. C, p. 103156, 2024, <https://doi.org/10.1016/j.scico.2024.103156>.
- [36] C. Liang, Q. Wei and Z. Jiang et al., "A Source Code Vulnerability Detection Method Based on Adaptive Graph Neural Networks," in *ASEW '24: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2024, pp. 187-196, <https://doi.org/10.1145/3691621.3694950>.
- [37] Y. Luo, W. Xu and D. Xu , "Detecting code vulnerabilities with heterogeneous GNN training," *International Journal of Information Security*, vol. 24, no. 213, 2025, <https://doi.org/10.1007/s10207-025-01132-x>.
- [38] X. Cao, J. Wang and P. Wu et al., "VulMPFF: A Vulnerability Detection Method for Fusing Code Features in Multiple Perspectives," *IET Information Security*, vol. 1, pp. pp. 1-15, 2024, <https://doi.org/10.1049/2024/4313185>.
- [39] B. V. Cong and D. X. Cho, "Detecting software vulnerabilities based on source code analysis using GCN Transformer," in in *2023 RIVF Int. Conf. Comput.*, Hanoi, Vietnam, 2023.
- [40] C. V. Bui, C. X. Do and T. T. Do, "Detection of source code vulnerabilities using Nature language processing and deep graph network," *Journal of Science and Technology on Information security*, vol. 23, no. 3, pp. 27-42, 2024, DOI: 10.54654/isj.v3i23.1057.
- [41] S. Chakraborty, R. Krishna and Y. Ding et al., "Deep Learning based Vulnerability Detection: Are We There Yet?," *IEEE Transactions on Software Engineering*, vol. 48, pp. 3280-3296, Sept. 2022, DOI: 10.1109/TSE.2021.3087402.
- [42] "JOERN, The Bug Hunter's Workbench," 2024. [Online]. Available: <https://joern.io/>. [Accessed 16 9 2026].
- [43] B. V. Cong and C. D. Xuan, "A New Framework for Software Vulnerability Detection Based on an Advanced Computing," *Computers, Materials & Continua*, vol. 79, no. 3, pp. 3700-3723, 2024, DOI: 10.32604/cmc.2024.050019.
- [44] J. Busch et al., "NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification," in *SSDBM '21: Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, pp. 121 - 132, 2021.
- [45] M. Chen, Z. Wei and Z. Huang et al., "Simple and Deep Graph Convolutional Networks," in *Proceedings of the 37th International Conference on Machine Learning, PMLR*, Vol. 119, pp. 1725-1735, 2020.
- [46] K. KC, R. Li, F. Cui and A. Haake, "Predicting Biomedical Interactions with Higher-Order Graph Convolutional Networks," *Journal of latex class files*, vol. 14, no. 8, p. 99, 2020.
- [47] M. Balcilar and G. Renton et al., "Bridging the Gap Between Spectral and Spatial Domains in Graph Neural Networks," 2020.
- [48] J. K. Blitzstein et al., *Introduction to Probability*, Boca Raton, FL 33487-2742, USA: Taylor & Francis Group, LLC, ISBN-13: 978-1-1383-6991-7, 2019.
- [49] "Download FFmpeg," 2026. [Online]. Available: <https://ffmpeg.org/download.html>. [Accessed 16 9 2026].
- [50] "CVE," 2024. [Online]. Available: <http://cve.mitre.org>. [Accessed 16 9 2026].
- [51] G. Siewruk and W. Mazurczyk, "Context-Aware Software Vulnerability Classification Using Machine Learning," in *IEEE Access*, vol. 9, pp. 88852-88867, 2021, doi: 10.1109/ACCESS.2021.3075385.
- [52] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Network and Distributed Systems Security (NDSS) Symposium 2018, 18-21 February 2018.*, San Diego, CA, USA, 2018, <http://dx.doi.org/10.14722/ndss.2018.23158..>
- [53] Z. Li et al., "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," in *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 2244-2258, doi: 10.1109/TDSC.2021.3051525., 2022.
- [54] Y. Zhou, S. Liu and J. Siow et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," In: *Advances in Neural Information Processing Systems*, no. 915, p. 10197-10207, 2019.

- [55] Z. Li et al., "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," in *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 2244-2258, 2022, doi: 10.1109/TDSC.2021.3051525..
- [56] Y. Zhou, S. Liu and J. Siow et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," In: *Advances in Neural Information Processing Systems*, no. 915, p. 10197-10207, 2019, <https://doi.org/10.48550/arXiv.1909.03496>.