Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.049310

ARTICLE

HCRVD: A Vulnerability Detection System Based on CST-PDG Hierarchical Code Representation Learning

Zhihui Song, Jinchen Xu, Kewei Li and Zheng Shan*

School of Cyberspace Security, Information Engineering University, Zhengzhou, 450001, China

*Corresponding Author: Zheng Shan. Email: shanzhengzz@163.com

Received: 03 January 2024 Accepted: 12 April 2024 Published: 20 June 2024

ABSTRACT

Prior studies have demonstrated that deep learning-based approaches can enhance the performance of source code vulnerability detection by training neural networks to learn vulnerability patterns in code representations. However, due to limitations in code representation and neural network design, the validity and practicality of the model still need to be improved. Additionally, due to differences in programming languages, most methods lack cross-language detection generality. To address these issues, in this paper, we analyze the shortcomings of previous code representations and neural networks. We propose a novel hierarchical code representation that combines Concrete Syntax Trees (CST) with Program Dependence Graphs (PDG). Furthermore, we introduce a Tree-Graph-Gated-Attention (TGGA) network based on gated recurrent units and attention mechanisms to build a Hierarchical Code Representation learning-based Vulnerability Detection (HCRVD) system. This system enables cross-language vulnerability detection at the function-level. The experiments show that HCRVD surpasses many competitors in vulnerability detection capabilities. It benefits from the hierarchical code representation learning method, and outperforms baseline in cross-language vulnerability detection by 9.772% and 11.819% in the C/C++ and Java datasets, respectively. Moreover, HCRVD has certain ability to detect vulnerabilities in unknown programming languages and is useful in real open-source projects. HCRVD shows good validity, generality and practicality.

KEYWORDS

Vulnerability detection; deep learning; CST-PDG code representation; tree-graph-gated-attention network; cross-language

1 Introduction

Computer software plays an important role in today's information society, but the security risks it faces are becoming increasingly prominent. The 2023 Open-Source Security and Risk Analysis (OSSRA) report audited more than 1,700 codebases, and found that 84% of them contained at least one known software vulnerability [1]. So-called software vulnerabilities are security flaws or weaknesses that exist in computer systems or applications. They arise due to oversights in the development process or inadequate security measures. Once discovered and exploited by attackers, these vulnerabilities can lead to the acquisition of unauthorized privileges or the execution of

malicious operations, posing significant risks to the entire software ecosystem. Detecting and fixing vulnerabilities during the software development stage can significantly reduce costs and mitigate security risks, so the importance of source code vulnerability detection is self-evident. Additionally, effective source code vulnerability detection can promptly discover potential vulnerabilities from open-source code repositories, preventing the further propagation of vulnerabilities due to code cloning.

Therefore, research on source code vulnerability detection has attracted much attention for decades. Typically, source code vulnerability detection methods rely on static analysis techniques, such as rule matching or code similarity comparison, to identify potential vulnerabilities. The methods based on rule matching generate vulnerability rules by manually analyzing various vulnerabilities, and then these rules are used to match the source code to detect vulnerabilities. This approach is used by many open-source tools (e.g., Flawfinder [2], Clang [3]) and commercial products (e.g., Checkmarx [4], Fortify [5]). They are capable of accurately locating vulnerabilities and generating alerts, thus frequently used in software development and code review. The methods based on code similarity comparison [6–8] identify identical vulnerabilities by analyzing the similarity between the code under inspection and the code containing known vulnerabilities. It only requires a small number of vulnerable code instances to detect the same vulnerability in the target program, making it highly effective in detecting vulnerabilities caused by code cloning.

Despite their proven effectiveness, these methods and tools have numerous disadvantages. The methods based on rule matching often suffer from imperfect rules, leading to high false positive and false negative rates. The methods based on code similarity struggle to detect vulnerabilities that are not caused by code cloning, resulting in a significant false negative rate. In addition, they share some common shortcomings, they (1) have difficulties in detecting complex contextual vulnerabilities, and (2) have poor generality and are difficult to migrate to code written in other programming languages.

1.1 Recent Efforts

To overcome these limitations, deep learning (DL) has been introduced into the field of vulnerability detection research in recent years. DL-based methods for source code vulnerability detection rely on combinations of different code representations and neural networks. Prior research has used neural network models such as convolutional (CNN) [9,10], recurrent (RNN) [11–14], and graph neural network (GNN) [15–19], and explored text [9], sequence [12–14,20], and graph [21–25] representations of source code. Code representation is a form that is transformed from source code to facilitate processing by neural networks. Text representation treats code as textual data, enabling the utilization of natural language processing (NLP) or text mining techniques to analyze patterns and structures within the code. Sequence representation divides code into discrete tokens or statements, which are selected according to certain rules and combined into a sequence, such as program slice. Graph representation converts code into graphs according to some logical structure of the program (e.g., Abstract Syntax Tree (AST) [26], Program Dependency Graph (PDG)). These representations rely on GNN to learn vulnerability patterns. Due to the use of more structure information of the code, some graph-based methods [15,19,24,25] have become the current state-of-the-art techniques.

DL-based methods have addressed the issues of prior approaches to a certain extent. They eliminate the need for manually defined rules, reduce false positive and false negative rates, and can detect complex contextual vulnerabilities. Some studies have also categorized vulnerabilities or improved the granularity of detection to a more precise line-level. These advancements have facilitated the transition of such methods from academic research to practical applications.

1.2 Limitations

Although DL-based methods have made significant progress, they still have some limitations in the design of neural networks and code representation, and there are still unresolved issues in their applications.

Firstly, most studies have directly applied existing neural networks without designing new neural network architectures specifically tailored for code representations to enhance model accuracy.

Secondly, code representation significantly impacts the effectiveness of the methods, but existing code representations have certain limitations. The detection capabilities of text- and sequence-based methods are constrained because they fail to adequately leverage the structured semantic information inherent in code [24], and some methods rely on manually defined features. Although graph-based methods demonstrate stronger detection capabilities, they often exhibit poorer robustness. These methods require complete and compilable code for analysis, and the accuracy of the models heavily depends on the design of the graph representation. Currently used graph representations suffer from the following drawbacks: Information from AST or PDG is used in a manner that is incomplete, harms code semantics, or inhibits graph-based learning. For example, reference [26] completely parsed the source code into an AST but at the same time ignores the control and data dependency information; references [18,19] combined PDG, control dependence graph and text but do not consider the AST information. Some studies [23,24,27,28] used more complex graph structures, but the simple combination of graphs with different meanings may harms original code semantics. Moreover, it is not suitable to use homogeneous graph neural network (GNN) to process such heterogeneous graphs. The large number of connections in the combinatorial graph causes different nodes to have a number of similar neighbors, which may cause their features to tend to be the same. In other words, this will result in the over-smoothing problem in GNN. These problems inhibit graph-based learning.

Lastly, most research has primarily concentrated on vulnerability detection in the source code of the C programming language, while studies on cross-language vulnerability detection and enhancing the generality of models remain relatively scarce.

1.3 The Proposed Solution

In this paper we propose HCRVD: A vulnerability detection system based on hierarchical code representation learning. The goal of HCRVD is to improve the code representation and neural network of DL-based vulnerability detection techniques, thereby boosting the validity and practicality of function-level vulnerability detection methods. Additionally, it also aims to enhance its generalizability for cross-language vulnerability detection tasks.

To accomplish this, we first propose a novel Concrete syntax trees-Program dependence graph (CP) hierarchical code representation, which unfolds the code hierarchically to fully extract the deep features of code syntax and semantics. The CP hierarchical code representation takes into full account the structural and semantic information at both the function and statement levels. At the function level, each function in the code is parsed into a PDG, which encapsulates the control dependencies and data dependencies of that function. Each node in the function PDG represents a statement. At the statement level, each statement is further parsed into a Concrete Syntax Tree (CST). The leaf nodes of each CST contain the original lexical information of the code, while the non-leaf nodes represent the syntactic structure of the code.

Subsequently, to enable the model to effectively learn program details, we designed a Tree-Graph-Gated-Attention (TGGA) neural network tailored to the structure of the CP representation. At the

statement level, TGGA employs a Tree neural network with Gating and Attention mechanisms (TGA) to learn the embedding vectors of statements and propagate them to the higher function level. At the function level, TGGA utilizes a Graph neural network with Gating and Attention mechanisms (GGA) to learn the embedding vector of the entire hierarchical code representation, which is then used for learning of vulnerability patterns and classification.

Finally, to enable HCRVD to handle code data from multiple languages, we employ various methods to reduce the heterogeneity across different languages. First, the code is normalized to remove information irrelevant to program semantics and identifiers specific to different programming languages, preventing the model from associating vulnerabilities with specific patterns. Second, the same code analysis tools are used for function PDG extraction and statement AST parsing across different languages, which helps reduce data discrepancies arising from variations in code analysis methods. Third, the introduction of hierarchical code representation learning enables the model to establish connections between vulnerability patterns and code structures. The inclusion of numerous syntax structure information nodes reduces the proportion of differentiated identifiers, and the hierarchical information transfer allows the model to learn the internal structural information of the code, adapting to tasks across different languages. These measures enhance the cross-language vulnerability detection capabilities of HCRVD.

We evaluated HCRVD on a C/C++ & Java cross-language dataset containing 64,722 functions extracted from SARD [29] and NVD [30], of which 31,362 functions are from the Java language and 33,360 functions are from the C/C++ language. The dataset includes 27,911 vulnerability functions and 36811 non-vulnerability functions. Evaluation results show that HCRVD surpasses many competitors in vulnerability detection capabilities on the C/C++ single language code dataset, and the accuracy is improved by at least 3.81% compared to existing methods. It benefits from the CST-PDG hierarchical code representation learning method, and outperforms baseline in cross-language vulnerability detection by 9.772% and 11.819% in the C/C++ and Java datasets, respectively. HCRVD has certain ability to detect vulnerabilities in unknown programming languages without any exposure to this language at all, and its accuracy rate exceeds baseline by 34.775%. HCRVD plays a more effective level of detection in real open-source projects in different languages. HCRVD shows strong validity, generality and practicality.

1.4 Contributions

In general, the contributions of this work include:

1. We propose a novel hierarchical code representation, CP structure, which combines CST and PDG to maximize the retention of code syntax and semantic information related to vulnerabilities in functions.

2. We design a TGGA neural network to effectively extract the code feature information in the CP data.

3. We propose and implement a vulnerability detection system, HCRVD, which can extract program CP data automatically and train TGGA for vulnerability detection. Tests conducted on the system showed that the HCRVD is more effective compared to the previous model.

4. Our experiments demonstrate that HCRVD has strong generality and practicality to detect multiple types of vulnerabilities in multiple languages at the same time without re-training the model, and has the ability to detect unknown languages. HCRVD performs more efficiently in real open-source projects in different languages. HCRVD is also robust and scalable, the code to be detected

does not need to be complete and compilable. This system can be trained on more languages to further extend the coverage of vulnerability detection.

The source code for HCRVD presented in this paper is available online at: https://github.com/ffff12138/HCRVD (Accessed: Apr. 10, 2024).

2 Related Works

In this section, we review related works on DL-based source code vulnerability detection, including early and recent works. We contrast them to the HCRVD's approach, code representation, neural network, and scope of application, which are summarized in Table 1.

Table 1: Summary of related works on DL-based source code vulnerability detection

Category	Year	Approach	Code representation	Neural network	Scope of detection
Text-based	2018	Russle et al. [9]	Text	CNN	C/C++
	2022	VulBERTa [31]	Text	RoBERTa	C/C++
	2022	VUDENC [11]	Text	LSTM	Python
	2023	Napier et al. [32]	Text	BERT & others	C/C++
Sequence-based	2018	Vuldeepecker [12]	Program slice	BLSTM	C/C++
	2021	SySeVR [14]	Program slice	BRNN	C/C++
	2022	Thapa et al. [20]	Program slice	Transformer	C/C++
Graph-based	2019	Devign [15]	Composite graph	GNN + conv	C/C++
	2021	Funded [16]	Augmented AST	GGNN	Cross-language
	2021	DeepWukong [17]	PDG	GCN	C/C++
	2021	BGNN4VD [21]	CPG	GNN + conv	C/C++
	2021	Reveal [22]	CPG	GGNN	C/C++
	2022	VulCNN [25]	PDG	CNN	C/C++
	2023	UCPGVul [23]	CPG	GCN	C/C++
	2023	CSGVD [18]	Sequence + CFG	GNN	C/C++
	2023	VulChecker [24]	PDG	GNN	C/C++
		HCRVD	CST-PDG	TGGA	Cross-language

2.1 Vulnerability Detection Approach

Early work [9] combined code text with CNN for vulnerability detection, which is the most intuitive application of deep learning. The proposal of Vuldeepecker introduced the concept of code gadget, which led to further research on methods based on program slice (or sequence) [13]. Devign proposed a graph-based approach, demonstrating that leveraging software structural information through program graph representations and GNN learning can achieve superior results.

The three classical works mentioned above have greatly inspired later research. For instance, in recent studies, VulBERTa and VUDENC are extensions of text-based approaches, while SySeVR and the work by Thapa et al. [20] are successors to Vuldeepecker. Additionally, numerous works [21–24] have further improved upon graph representation learning methods. Most research focuses on optimizing code representation or improving neural networks, and some research expands application scenarios. The essence of HCRVD is to optimize the design of code representation and neural networks based on graph-based methods, and to expand the ability to apply in cross-programming language scenarios.

2.2 Code Representation

The code representations used in research can be broadly categorized into three types: Text, sequence, and graph representations. Text representation is the simplest method among them, as it can be directly derived from the sequential text of the code. Napier et al. [32] concluded through experiments that text-based methods tend to yield inferior results. However, this approach offers good scalability, and a simple combination with language models can lead to effective models like Vulberta [31]. Methods based on sequence representations explore how to extract program slices. For instance, Vuldeepecker extracts slices from data flow graphs (DFG), while SySeVR and the slicing approach by Thapa et al. [20] rely on PDG.

Research on graph representations of code is the most extensive, as it has the greatest impact on detection methods. The focus of these studies is on how to leverage program logic structures to design graph representations. Funded utilizes augmented AST, CSGVD combines control flow graphs (CFG) with sequences, VulCNN and VulChecker use PDG as a graph representation. Among these graph representations, AST is a tree-like representation that describes the abstract syntax structure of source code. CFG demonstrates the program's execution flow and the conditions that need to be satisfied. PDG shows the control and data dependencies in the program. More advanced graph representations include code property graphs (CPG) and some composite graphs. CPG is a type of combined graph that integrates information from AST, CFG, and PDG into a single graph. It has been widely used in recent studies [21–23]. The graph representations used in Devign and FUNDED are composite graphs that are formed by combining multiple subgraphs. Although CPG and composite graphs contain information from many subgraphs, this does not mean that they are more suitable for model learning. For example, the information from AST nodes in CPGs cannot be effectively propagated throughout the graph. Composite graphs may connect semantically unrelated statements or token nodes without leveraging edge types, resulting in inappropriate information transmission paths. These shortcomings can hinder the neural network's ability to learn patterns of vulnerability propagation paths. In contrast, our proposed CP representation effectively transmits node information from the CST and selects a clear combination of graphs that are crucial for learning vulnerability propagation paths.

2.3 Neural Network

Text-based and sequence-based methods are often integrated with NLP models. They make use of traditional models like CNN [9], RNN [14], and LSTM [11,12], and they also benefit greatly from pre-trained language models based on the Transformer architecture, such as BERT [20,32,33] and RoBERTa [31,34]. Although sequence representations are extracted based on some graphical form of a program, their flat structure precludes the utilization of the graph-like structure and message passing paths inherent in software.

Graph-based methods typically rely on GNN for learning. DeepWukong and UCPGVul leverage graph convolutional neural networks (GCN) to propagate information to adjacent nodes and perform convolutional operations, learning graph embeddings for classification tasks. Devign, BGNN4VD, CSGVD, VulChecker, and DeepDFA utilize or enhance GNNs to perform message passing and learn global embeddings by leveraging the structure and attributes of the graph. However, it is difficult for GCN and GNN to learn long-distance relationships in the structure, so some works [16,22] introduced gated graph neural network (GGNN) to address this challenge. These network models are selected based on the structural features of the code representation. The TGGA neural network model used by HCRVD is specifically designed based on the hierarchical representation of CP, possessing the ability to propagate messages over long distances, thus making it more suitable for the task of vulnerability detection.

2.4 Scope of Detection

Previous researches have been extended in various application directions. Extensions in terms of detection granularity encompass research at the level of slices [12–14,20], functions [9,15,21,22], lines and statements [19,24,35]. Extensions in interpretability include the study of evaluation metrics and influencing factors for vulnerability detection [36]. Extensions in detection scope encompass the research of vulnerability detection across multiple types [13,24] and across projects [27,37]. There has been limited exploration in cross-language vulnerability detection. Most studies are based on C/C++ languages, some works [11] have also explored vulnerability detection in other programming languages. Recently, FUNDED has attempted cross-language vulnerability detection research, but its model requires transfer learning. In contrast, HCRVD achieves cross-language vulnerability detection without the need for transfer learning by extracting a unified and normalized CP representation and introducing the TGGA network, which possesses stronger generalization capabilities.

3 Methods

This section details the technical research and system implementation of HCRVD. We first introduce the CP code representation and TGGA neural network utilized in HCRVD, followed by a presentation of the overall framework and implementation process of HCRVD.

3.1 CP Code Representation

The various structured information derived from code analysis is employed to represent the inherent properties of programs. This encompasses AST, CST, and PDG, which effectively capture the syntactical and semantic relationships within the source code. CST is a tree representation that contains syntax information of the source code and is a direct translation of the code. The leaf nodes of the CST retain the source code information, while the non-leaf nodes provide the syntax information. PDG is a graph-structured data used to represent the dependencies between variables and statements during program execution. The edges of the flow from input data to sensitive operations can be obtained directly through PDG, which provides contextual information about the program, helps the model to understand the relationship between program statements to identify potential vulnerabilities.

In this section, we leverage the structured information of code to design a code representation, aiming to enhance the effectiveness and universality of the model in vulnerability detection tasks. Consequently, the design of the code representation needs to take into account various factors such as code semantics extraction, vulnerability trigger mechanisms, model generality, and performance. Detailed analysis from these perspectives is presented as follows:

First, we analyze from the perspective of code semantics. A statement is the basic unit to carry the semantics of the source code. Using syntax trees of statement granularity can effectively alleviate the gradient vanishing problem caused by the long-term dependency of complete syntax tree of function granularity, which has the large scale and the problem of structural distortion in encoding. And more syntax and semantic information can be captured compared to the token level of granularity [38]. Therefore, it is appropriate and effective to build CST embeddings at the statement granularity.

Second, we analyze from the perspective of vulnerability trigger mechanisms. The location of the vulnerability is closely related to the data flow and control flow of the program. For example, the buffer overflow vulnerability is often caused by the variable being written beyond the defined scope limit, and the triggering of the vulnerability involves multiple statements. These statements is difficult to be directly connected through the syntax tree, because they need to be fed back to the upper-level node. Therefore, PDG is semantic information that should be retained. Moreover, the triggering of the vulnerability is also related to the syntax structure of the specific statement. For example, the statement where the vulnerability is triggered belongs to the function call statement, and PDG can only extract the code semantic information above the statement granularity, so it is likely to miss this important information. To prevent it, a syntax tree can be added to extract syntax and lexical information under the statement granularity. Reference [15] added various edges representing program structure information on basis of AST to form a graph, and used graph neural network for detection. But the graph built by this method is very complex, the graph neural network may suffer from over-smoothing problem in node propagation. To avoid these situations, in this paper, we propose a hierarchical code representation method. Each node in the PDG of a program is represented as a CST, and the CST is used to represent the syntax and semantic information of statements. Then the PDG is used to represent the semantic information of the functions. Fig. 1 shows source code of a vulnerability function and its PDG structure. The rightmost box shows the CST structure corresponding to the statement "strncat(data, source, strlen(source));" at one of the nodes in the PDG. Such a hierarchical representation allows the model to associate the vulnerable statement in line 13 with the defining variable statements in lines 4, 5, 7, and 8. At the same time, it focuses on specific details within these statements, such as the length of the array and the vulnerable function strncat. This approach enables the model to identify the root cause of the vulnerability, which is the copying of a string that exceeds the buffer size.

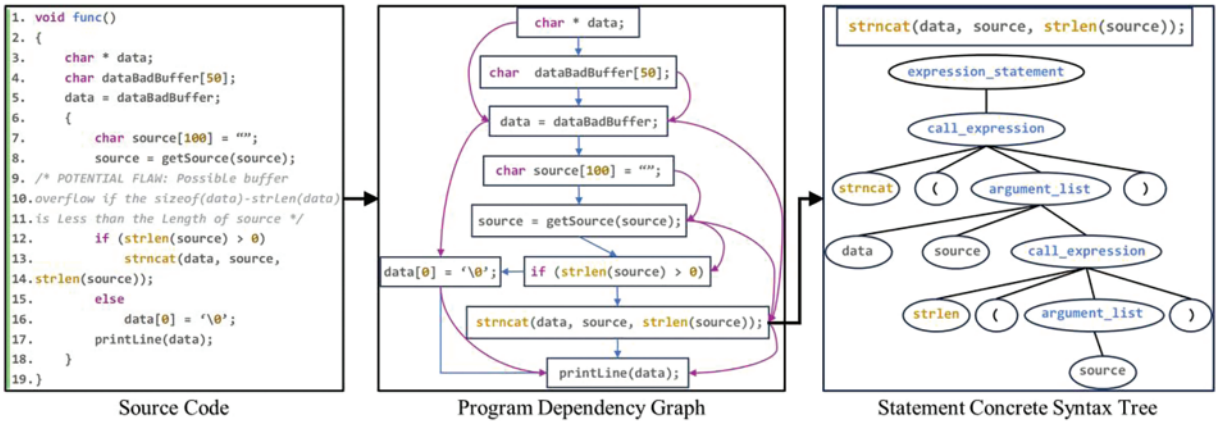

Figure 1: Vulnerability function, PDG and statement CST

Third, we analyze from the perspective of model generality. Using sequence-based detection methods such as [12–14] often requires the use of existing vulnerability-related knowledge to extract code slices, the model only works on the code that contains the defined features. Moreover, although the use of graph-based detection methods does not suffer from such a problem, their performance on cross-language tasks is still limited, because the heterogeneity of different programming languages poses a great difficulty in generalizing the model. PDG has abstracted the code and weakened this heterogeneity to a certain extent, and adding CST can further reduce the impact of heterogeneity and improve the model's generality for cross-language detection.

Lastly, we analyze from the perspective of model performance. The design of graph representations for code does not necessarily benefit from incorporating as much structural information as possible. A simple combination of graphs with different meanings can potentially damage the original code semantics, leading to a decrease in model accuracy. The existence of numerous connections in combined graphs can introduce inappropriate information propagation paths, which can also hamper model performance. Additionally, complex combined graph representations can significantly increase the cost of model training. It is only through the design of appropriate graph representations that these issues can be avoided. The aforementioned hierarchical code representation retains only the most crucial structural information of CST and PDG pertinent to the task of vulnerability detection. This approach not only reduces complexity but also facilitates the enhancement of model performance.

To sum up, we use statements as the hierarchical granularity and design a CP hierarchical code representation by combining function PDG and statement CST. CP is a directed graph in which all non-root nodes of each CST have an edge pointing to their parent node. All CST root nodes are connected together according to the directed edges of the PDG. Fig. 2 shows the corresponding CP hierarchical data structure of the source code, where each function is parsed as a PDG which preserves the dependencies between statements, and each PDG node is further parsed as a statement CST which preserves the syntax and semantics of the statements. In this process, the code is disassembled from function granularity to statement granularity and finally to token granularity, forming a hierarchical parsing structure. In the concrete implementation, the PDG records all edges in the form of an adjacency matrix.

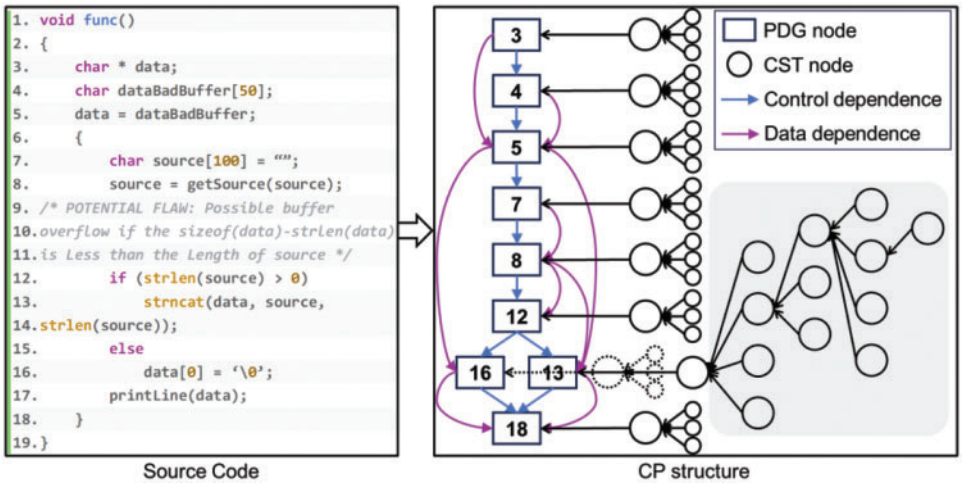

Figure 2: CP hierarchical data structure of the code

3.2 TGGA Network Architecture

To make use of the hierarchical code structure, in this section, we design a hierarchical Tree-Graph-Gated-Attention neural networks based on GRU [39] with attention [40] to learn the vector representation of the code. Using GRU to construct the TGGA network can alleviate the long-term dependence problem [41] caused by too large CP. Moreover, GRU actually takes much less time than the long short-term memory network with almost the same performance [39], which can greatly accelerate our training process. Attention mechanism is used to optimize the readout [42] of different layers of vectors in the code, as deep learning models based on the attention mechanism are able to extract important features related to vulnerabilities by training to identify the importance of the token in the code. We show how they work in Fig. 3. GRU introduces two gates, the update gate z_t and the reset gate r_t, to control the proportion of hidden states that will be updated and generate candidate hidden states $\widehat{h}_t$, respectively. Eqs. (1)–(4) show the forward propagation process of GRU, where x_t denotes the input information at the current moment, h_{t-1} denotes the hidden state at the previous moment, h_t denotes the hidden state passed to the next moment and also denotes the output at the current timestep, W, U and b are the trainable parameters, σ and tanh are the activation functions. The attention mechanism learns three vectors Q, K, V from the input and obtains different weights a_i for each element of the input by calculating similarity, which is used to selectively enhance the influence of certain features. Its forward propagation process is shown in Eqs. (5)–(6). The three vectors Q, K, and V are obtained by multiplying the input vector and different trainable weight matrices, which represent the information used for querying and accepting the matching for adjusting weights between the vectors, and the information of the input features, respectively.

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{1}$$

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \tag{2}$$

$$\widehat{h}_t = \tanh\left(W_h x_t + U_h\left(r_t \odot h_{t-1}\right) + b_h\right) \tag{3}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \widehat{h}_t \tag{4}$$

$$a_i = softmax\left(Q \cdot K_i\right) = \frac{exp\left(Q \cdot K_i\right)}{\sum_i exp(Q \cdot K_i)} \tag{5}$$

$$Attention\ Value = \sum_i a_i V_i \tag{6}$$

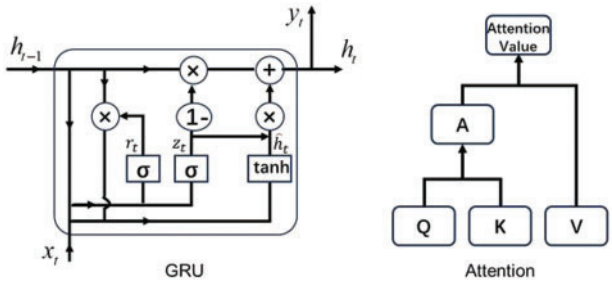

Figure 3: Structure of GRU and attention

We construct TGGA hierarchical network based on CP data structure, which consists of four layers: Word Embedding (WE), Tree-Gated-Attention (TGA), Graph-Gated-Attention (GGA) and Multi-Layer Perceptron (MLP) classifier. TGGA uses GRU for message passing between nodes in the same code hierarchy and attention for vector readout between hierarchies. Fig. 4 illustrates the

TGA-based statement encoder on a CST structure used to learn statement vectors. Starting from the one-hot vectors, the statement CST is first initialized by WE to embed the vectors and then fed into the TGA network. The middle part of the image shows the TGA network structure, which consists of GRUs with the same tree structure as the CST, and the basic GRU forward propagation process is shown in the gray box. The TGA neural network gets the output state of the statement CST shown on the right side of the figure (including the vectors of all the nodes) and the hidden state of the root node, and the final output of the statement vector is read out using the attention mechanism. More specifically, for a statement CST k, the initial input to the network is a one-hot vector of all nodes of a CST $\boldsymbol{x} = [x_0, [x_1, [\ldots]]]$. Each node n first undergoes word embedding to get the initialized embedding vector representation v_n of the node:

$$v_n = W_e x_n \tag{7}$$

where W_e is the trainable parameter of embedding, after which information is passed to the root node through TGA starting from leaf nodes. Each node n receives the sum of hidden states h_i of its children nodes as the previous state $h_n^{'}$ to the GRU, and the initialization vector v_n of the node serves as the current input to the GRU. The output of the GRU serves as the hidden vector h_n of the current node as well as the output vector. The forward propagation process of TGA is denoted as:

$$h_n^{'} = \sum\nolimits_{i \in C_n} h_i \tag{8}$$

$$r_n = \sigma\left(W_r v_n + U_r h_n^{'} + b_r\right) \tag{9}$$

$$z_n = \sigma\left(W_z v_n + U_z h_n^{'} + b_z\right) \tag{10}$$

$$\widehat{h_n} = \tanh\left(W_h v_n + U_h\left(r_n \odot h_n^{'}\right) + b_h\right) \tag{11}$$

$$h_n = z_n \odot h_n^{'} + (1 - z_n) \odot \widehat{h_n} \tag{12}$$

where C_n denotes the set of its children, $h_n^{'}$ is the sum of the hidden states of all children of the current node, h_n is the hidden state (i.e., the vector representation) of the current node, and $h_n^{'}$ is the zero vector for leaf nodes. The TGA after forward propagation will get the hidden state h_r of the root node as well as the output vectors of all nodes. They are read out using attention mechanism to obtain the vector representation s_k for the statement CST:

$$s_k = \sum\nolimits_i softmax(Q \cdot K_i) \cdot V_i \tag{13}$$

where query vector Q is set to the hidden state h_r of the root node, vector $K_i = h_i$ and $V_i = h_i$.

On the PDG structure, we use the GGA network layer to encode the function vectors. The left side of Fig. 5 shows the initial state of a PDG node. After the TGA network layer, the vector of statements corresponding to PDG node k at this point is s_k (the output vector of the CST), and these node vectors with the adjacency matrix make up the PDG that is the input to the GGA network layer. The middle part of Fig. 5 shows the network structure of GGA, which consists of GRUs with the same graph structure as the PDG, and the GGA updates the state of the nodes after forward propagation over multiple time steps, which is the same methodology used in the Gated Graph Neural Network (GGNN) [43]. The right side of the figure shows the final state of the nodes at the output of the network obtained after T time-step iterations, which is merged with the initial state and finally read out as a graph vector representation using the self-attention mechanism.

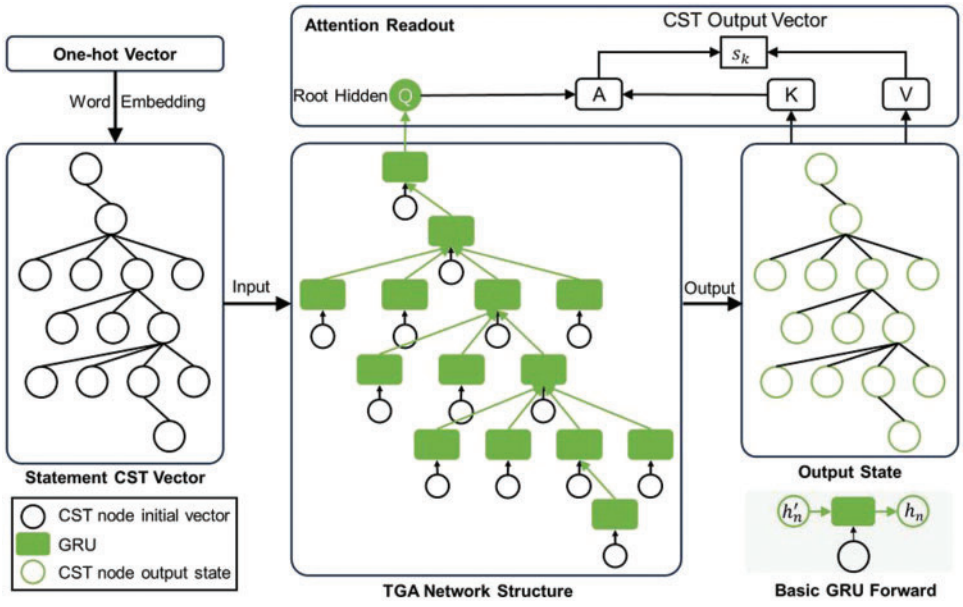

Figure 4: Statement encoder based on tree-gated-attention network

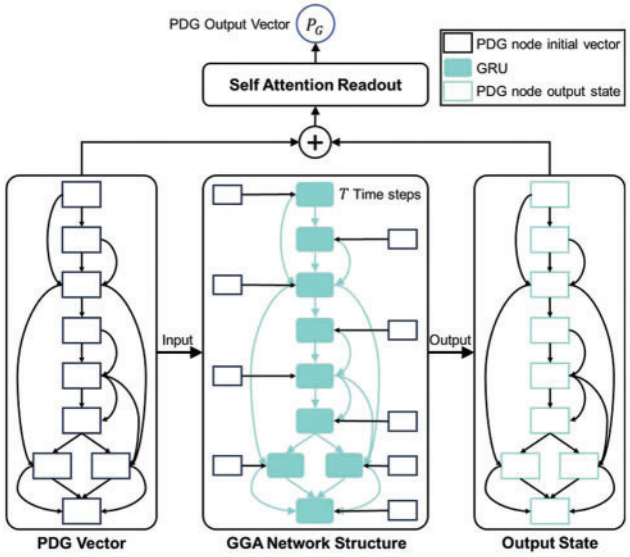

Figure 5: Function PDG encoder based on graph-gated-attention network

More specifically, for a PDG G, $\mathcal{K}$ is the set of its nodes, A is the concatenation of the adjacency matrix with its transpose tensor, and the initial input to the GGA layer at the time step $t = 1$ is a tensor $S^{(1)} = [s_1^{(1)\top}, \ldots, s_{|\mathcal{K}|}^{(1)\top}]^\top$ containing all the PDG node vectors. $\mathcal{N}_{(k)}$ is the set of neighboring nodes of PDG node k, then the information aggregation of this node at time step $t = 2$ comes from the sum of the hidden states of neighboring nodes: $a_k^{(2)} = 1 \cdot \sum_{u \in \mathcal{N}_{(k)}} h_u^{(1)} + 0 \cdot \sum_{u \notin \mathcal{N}_{(k)}} h_u^{(1)} = A_k S$. The forward propagation of GGA is described by Eqs. (14)–(19) as follows:

$$h_k^{(1)} = s_k \tag{14}$$

$$a_k^{(t)} = A_k[h_1^{(t-1)\top}, \ldots, h_{|\mathcal{K}|}^{(t-1)\top}]^\top + b_k \tag{15}$$

$$z_k^{(t)} = \sigma(W_z a_k^{(t)} + U_z h_k^{(t-1)} + b_z) \tag{16}$$

$$r_k^{(t)} = \sigma(W_r a_k^{(t)} + U_r h_k^{(t-1)} + b_r) \tag{17}$$

$$\hat{h}_k^{(t)} = \tanh(W_h a_k^{(t)} + U_h(r_k^{(t)} \odot h_k^{(t-1)}) + b_h) \tag{18}$$

$$h_k^{(t)} = (1 - z_k^{(t)}) \odot h_k^{(t-1)} + z_k^{(t)} \odot \hat{h}_k^{(t)} \tag{19}$$

where $h_k^{(1)} = s_k$ is the initial state of node k. At the t-th time step, node k receives the sum $a_k^{(t)}$ of the hidden states of its neighboring nodes as the input vector to the GRU at the current time step, and $h_k^{(t-1)}$ is the state vector of node k at the previous time step, which is used as the hidden state input to GRU. Next, the update gate z and the reset gate r are computed, and finally the state vector $h_k^{(t)}$ of the node at the t-th time step is obtained, and the final node vector $h_k^{(T)}$ is obtained after T time steps. During training, all nodes take this way forward propagation and the vector of PDG is read out using the neural network based self-attention mechanism shown in Eq. (20):

$$P_G = \tanh\left(\sum\nolimits_{k \in |\mathcal{K}|} \sigma(MLP_1(h_k^{(T)}, s_k)) \odot \tanh(MLP_2(h_k^{(T)}, s_k))\right) \tag{20}$$

where MLP is a linear layer network used to map the dimension of concatenation of $h_k^{(T)}$ and s_k to 1, σ and tanh are the activation functions, and $\sigma(MLP_1(h_k^{(T)}, s_k))$ denotes the neural network based attention mechanism used to decide which nodes are relevant to the current graph-level task. It ultimately yields the $\mathcal{K}$-dimensional P_G as a representation of the PDG.

Note that the network structures composed of GRUs in different layers changes dynamically from batch to batch, which we will discuss further in Section 3.4.4. After the TGGA network, we can get the vector representations of all code tokens, statements, and functions, on the basis of which we can further carry out the subsequent tasks. Here, we aim to provide function level vulnerability detection, so the vector representation of the function, P_G, is fed into the MLP classifier or classified by a logit obtained by summing all dimensions of the vector, and the output is the result of whether the function contains a vulnerability.

3.3 HCRVD System Framework

As shown in Fig. 6, the HCRVD consists of five phases: Data preparation, Function PDG extraction, Statement CST parsing, Vector transformation, TGGA Construction and Classification.

Data preparation: Extracting functions from source code, labelling whether functions contain vulnerabilities, and normalizing them.

Function PDG extraction: A PDG is generated from each normalized function, and subgraphs of the PDG are extracted as a means of data augmentation based on possible vulnerability-sensitive words for that function.

Statement CST parsing: The code statements of each node extracted from the PDG are parsed into CSTs individually and depth-first traversal is performed from each CST to extract the information of the node, generating token sequences and statement CST tree-structure data.

Dynamic vector transformation: A dictionary is built and CP data are transformed into vectors. Gated Recurrent Units (GRU) are used to dynamically build the network structure based on the maximum statement CST and the maximum number of PDG nodes (or thresholds) on batch to enable batch processing.

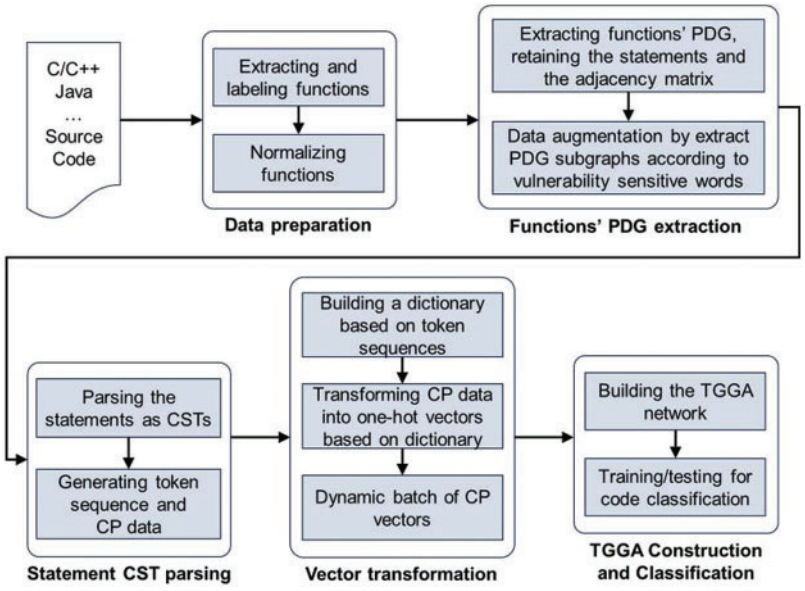

Figure 6: HCRVD system framework

TGGA construction and classification: Building a TGGA network and using GRUs to transfer information in different levels according to data structures. The function representation learned by TGGA is used for function granularity vulnerability detection.

In summary, HCRVD accepts input from code files, automates the extraction of functions from source code, and accomplishes PDG extraction and CST parsing. It detects the presence of vulnerabilities through trained models. The HCRVD system functions as an alternative to manual code auditing, statically analyzing code to detect potentially vulnerable functions. In addition, HCRVD is generalizable and can detect vulnerabilities in multiple languages without re-training the model, and has some ability to detect vulnerabilities in unknown languages. HCRVD is also robust and scalable, the code to be detected does not need to be complete and compilable. This system can be trained on more languages to further extend the coverage of vulnerability detection.

3.4 HCRVD Implementation Process

3.4.1 Data Preparation

In the data preparation phase, we extract the functions from the source code and label the functions containing vulnerabilities as 1 or else 0. In order to extract useful features from the original code of each function, we normalize the code from different languages by lexical analysis and naming transformation. We remove code that does not affect compilation (e.g., comments). User-defined variable and function names are mapped to normalized names in a one-to-one manner. Strings, characters, and floating-point numbers were lexically analyzed for specific identifiers. Integers as well as standard library functions were left intact, as they are often associated with vulnerabilities. Ultimately, we reduced the overall number of words contained in the code to 292 (including all the basis keywords, operators, separators, etc.). Fig. 7 illustrates the process of normalizing a function from its source code.

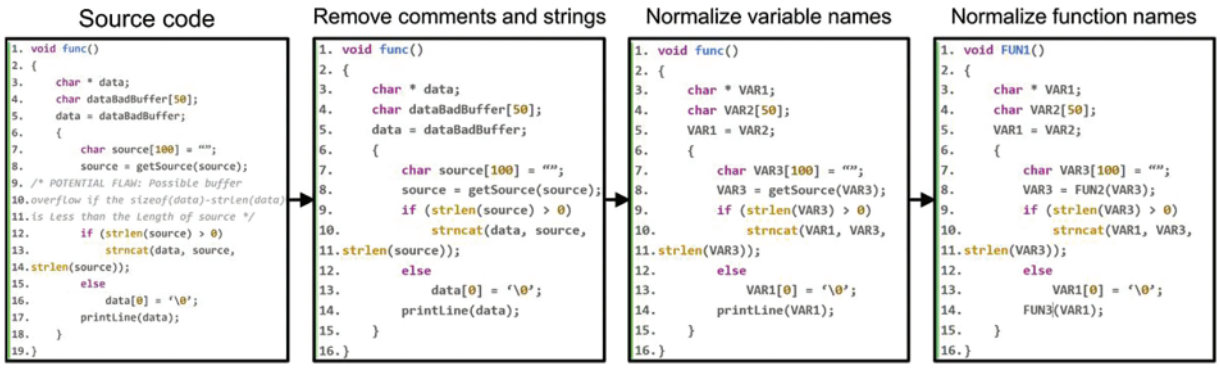

Figure 7: Data normalization

3.4.2 Functions PDG Extraction

After normalizing the source code, we extracted the program dependency graphs of the functions using the open-source code analysis tool Joern [44,45], which allows code analysis in multiple languages. In order to make the learning of the model can be improved by the existing knowledge related to vulnerabilities, we refer to the method of [12,46], which utilizes vulnerability-sensitive keywords to extract the PDG subgraphs of functions. The so-called vulnerability-sensitive keywords are some functions that are known to be potentially vulnerable as well as identifiers that appear frequently in vulnerable programs. The vulnerability sensitive words are localized to their sensitive statement nodes in the PDG, from which the reachable paths and nodes are recursively searched forward and backward in a single direction, respectively, and the subgraphs of the PDG formed by these nodes and paths represent the statements that are dependent on the sensitive statements. Fig. 8 illustrates the process of extracting the subgraph from the PDG, where line 10 is defined as a vulnerability-sensitive statement, and line 13 cannot reach line 10 nor be reached by line 10, so it is discarded. Unlike references [12,46], considering that the function containing vulnerability-sensitive words is only a part of all functions, if we train our model only on this part of the function, the model will not be able to detect a large number of functions that do not contain sensitive words. Therefore, we only use this part of the extracted subgraph as an expansion of the original PDG as a means of data augmentation in the training phase. We extract the statements of all the nodes of the graph and compute their adjacency matrices. This approach contributes to the model's validity and generality.

3.4.3 Statement CST Parsing

The statements extracted from nodes of PDG form a statement set $S = [s_1, \ldots, s_{|K|}]$, and each statement is parsed into a concrete syntax tree using the cross-language open-source code parsing tool Tree-sitter [47], respectively. As shown in Fig. 9, the statement set of a PDG is parsed to generate a set of statement CSTs, after which a depth-first traversal is performed for each CST starting from the root node, recording the code identifiers of the leaf nodes and the statement types of the non-leaf nodes, and generating a set of token sequences as well as a tree structure data for the statement CSTs. The sequence information corresponding to each statement contains the original code as well as the statement type information of the CST node, the token sequence data will be used to construct the dictionary, and the set of statement CST data from the same PDG together with the adjacency matrix of that PDG form the CP data.

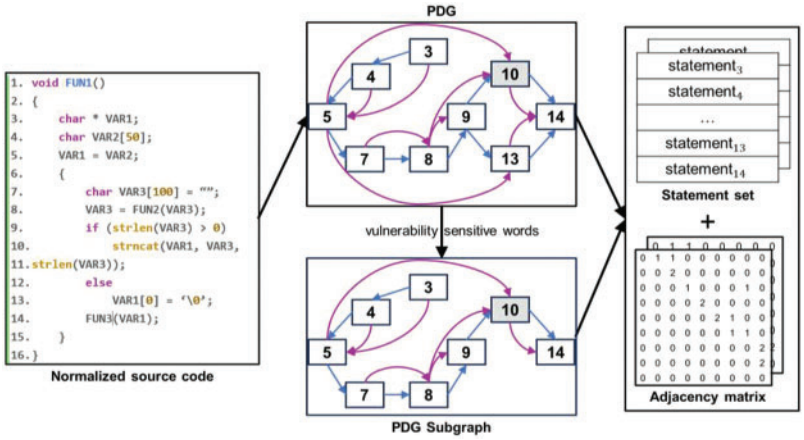

Figure 8: Function PDG extraction and subgraph generation

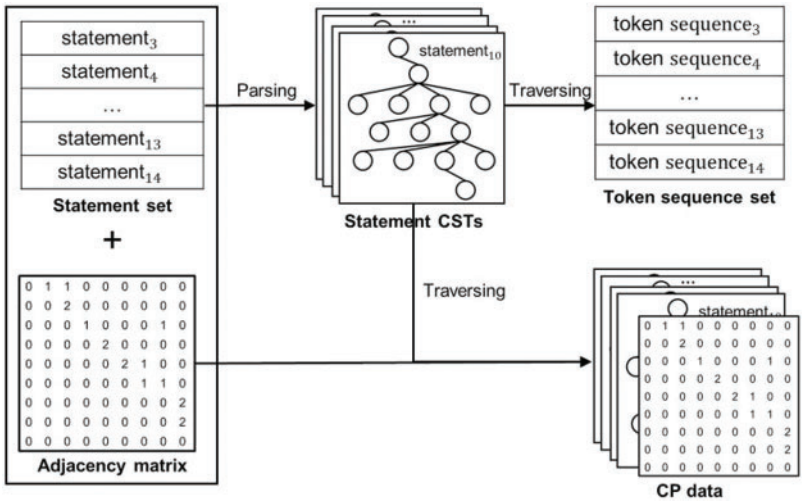

Figure 9: Statement CST parsing

3.4.4 Dynamic Vector Transformation

The next step is to build a dictionary based on the token sequence, the nodes of the statement CST are transformed into one-hot vectors using that, and the obtained CP one-hot vectors are the initial inputs to the TGGA network. There exists the problem of inconsistent length of the vectors, resulting in the inability of batch processing, which will lead to an increase in the training cost of the neural network. The reason for the inconsistent dimension of the CP vectors is that the number of children of each node in these CSTs is different, and the number of nodes in different PDGs is also different. It is necessary to fix the number of CST nodes and PDG nodes in a batch before concatenating the vectors for batch processing. A simple method [25] is to statistically analyze the data before starting training to obtain the cumulative distribution function of nodes in each layer. We adjust the threshold of retained nodes to truncate or pad vectors to a consistent dimension. The performance of the model using this approach is affected by the threshold.

Here, we use a dynamic batch approach to dynamically adjust the vector dimensions in each batch of training. Different tuning strategies are used at different layers in a batch. At the CST layer, since there will be no statements that exceed the expected length, no truncation is performed and the method of dynamically generating the maximum statement CST is used for padding. As shown in Fig. 10, for all $\sum_i^B |\mathcal{K}_i|$ CSTs in a batch (batchsize is B), the nodes of CST with the same location are stacked together, and the union of the node locations of all the CSTs is obtained to construct the maximal statement CST on this batch. All data in the same batch are padded according to it. The maximum statement CST is dynamically generated during the forward propagation of depth-first traversal of the CST for this batch. At the PDG layer, the vectors of the same CST are reintegrated and padded according to the maximum number of nodes in that batch of PDGs, but the padded nodes are not connected to the original PDG (degree 0). Due to the large variation in the size of the function, PDGs with a high number of nodes may appear, so we set a global truncation threshold M = 200, and let the number of nodes in a PDG be K. The number of nodes in the batch of PDGs with K<M will all be padded to $min(max(K_i), M)$, and the PDGs with K>M will compute node degree based on the adjacency matrix, and the nodes with a low degree will be truncated first. The vector dimension transformation in batch processing of a network is intended to speed up the training.

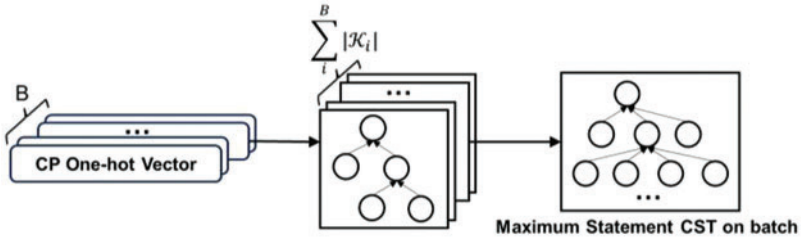

Figure 10: Maximum statement CST on batch

3.4.5 TGGA Construction and Classification

The TGGA structure have been described in Section 3.2 and we will not repeat it. It is worth noting that the GRUs used by different layers are not the same, and the network structure composed of GRUs in different batches changes dynamically as mentioned in Section 3.4.4. We show in Fig. 11 the process of CP data input into the TGGA network for classification. At the CST layer, the statement CSTs passing through the WE and TGA layers of the network will yield the statement representation vectors that correspond to the nodes in the PDG, which are connected according to the adjacency matrix to form the function PDG. At the PDG layer, the representation vector of the function PDG can be obtained after the GGA layer network. The GGA iteration has a time step of 4. The statement or function representation vectors can be used depending on the specific task. Our task is to provide function-level vulnerability detection, so classification can be performed using function representation vectors passing through a classification layer. After the model is trained in the training phase, the prediction phase still requires CP data to be extracted from the source code in order to achieve vulnerability detection. It is worth noting that TGGA can extract PDG vectors for classification as well as vectors of node statements for node prediction, thus using more finely labeled datasets can result in more fine-grained vulnerability detection capabilities.

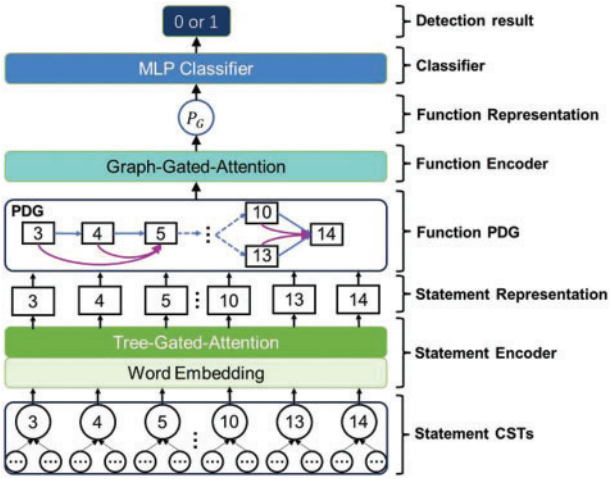

Figure 11: Classification process of TGGA network

4 Results

4.1 Research Questions and Experiment Designs

In order to evaluate the validity, generality, and practicality of the HCRVD system and the hierarchical code representation learning method for vulnerability detection tasks, we need to answer the following **Research Questions:**

- **RQ1:** How effective is the HCRVD in vulnerability detection? Is it better compared to other models?
- **RQ2:** Does hierarchical code representation learning make HCRVD more effective? How much better?
- **RQ3:** Does HCRVD have the ability to detect vulnerabilities across languages? How generalizable is it?
- **RQ4:** Does HCRVD have the ability to detect vulnerabilities on unknown programming languages?
- **RQ5:** Does HCRVD work in real projects? How practical is it?

To this end, we designed the following experiments:

Validity testing experiment: Test the validity of HCRVD against other vulnerability detection models on a dataset of source code for a single programming language, compare and analyze the differences between the models on each metric.

Hierarchical code representation learning method ablation study: The validity of HCRVD is tested against other code representation methods on a source code dataset of a single language to compare and analyze the differences between the various representation methods on each metric.

Cross-language vulnerability detection experiment: Test the generality of HCRVD system on C/C++ & Java cross-language source code datasets. Analyze the validity and generality of HCRVD for cross-language vulnerability detection in comparison with baseline.

Unknown programming language vulnerability detection experiment: Train the model on the C/C++ dataset and verify that the model can detect vulnerabilities on the Java dataset, analyze the

vulnerability detection performance and generality of HCRVD in unknown languages in comparison with baseline.

Practicality testing experiment: Verify the detection effect of the model in real projects with different languages and analyze the practicality of HCRVD in comparison with baseline.

4.2 Datasets and Experimental Configuration

4.2.1 Datasets

We collected vulnerability datasets from SARD, which is a program maintained by the National Institute of Standards and Technology, and NVD, which collects vulnerabilities from real-world software. They both contain a large number of vulnerability functions and non-vulnerability functions. By removing a small number of functions that contain only references to other functions (they are not inherently vulnerable), we constructed a C/C++ language dataset and a Java language dataset. The C/C++ language dataset has a total of 33,360 functions from SARD and NVD, including 12,303 vulnerability functions and 21,057 normal functions. The Java language dataset contains a total of 31,362 functions, including 15,608 vulnerability functions and 15,754 normal ones.

We label the extracted data to generate the initial label. Specifically, if the function name contains "bad" then it means that the function contains a known vulnerability and is labeled as 1, or else 0. We conducted our experiments using 5-fold cross-validation, where the dataset was divided into 5 parts, each part is divided into training, validation and test set with a ratio of 4:1:1.

4.2.2 Configuration

The hardware for our experiments included a server with an 8-core 3.60 GHz Intel(R) Core(TM) i7-7820X CPU and a TITAN RTX GPU. The server has 16 GB RAM and 24 GB VRAM.

The software environment for the experiments consisted of Pytorch-1.8.1, Joern-2.0.92 and Tree-sitter-0.20.1 running on Ubuntu 18.04.

Hyper-parameter configurations: We use Adam optimizer with a learning rate of 1e-3 and a weight decay of 1e-8. Batchsize is 64 and epoch is 50, and the training takes about 540 s per epoch.

4.2.3 Evaluation Metrics

We use three metrics including False Positive Rate (FPR), False Negative Rate (FNR) and Accuracy (Acc) commonly used in this field as shown in from Eqs. (21) to (23) to quantify the detection capability of HCRVD. Where True Positive (TP) reports the number of functions correctly classified as vulnerable, and False Positive (FP) reports the number of functions incorrectly classified as vulnerable. True Negative (TN) reports the number of functions correctly detected as non-vulnerable, and False Negative (FN) reports the number of functions incorrectly detected as non-vulnerable. Then FPR, FNR and Acc are calculated using them. FPR denotes the proportion of non-vulnerable functions misclassified as vulnerable, FNR denotes the proportion of vulnerable functions misclassified as non-vulnerable, and Acc denotes the proportion of correctly classified samples to all samples. They are important metrics for quantifying vulnerability detection systems. Higher Acc, lower FPR and FNR mean better detection performance of the system.

$$FPR = \frac{FP}{FP + TN} \tag{21}$$

$$FNR = \frac{FN}{TP + FN} \tag{22}$$

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

4.3 Experimental Results and Analysis

4.3.1 Results and Analysis of Validity Testing Experiment

For the RQ1, we first test our model on the C/C++ dataset, and we compare HCRVD with five other vulnerability detection methods as shown in Fig. 12, including TokenCNN [9], VulDeePecker [12], SySeVR [14], Devign [15], and VulCNN [25]. We choose these methods as baselines for comparison with HCRVD because they have similar detection granularity among all code representation learning open-source methods. And each of them is the most representative method in different code representations.

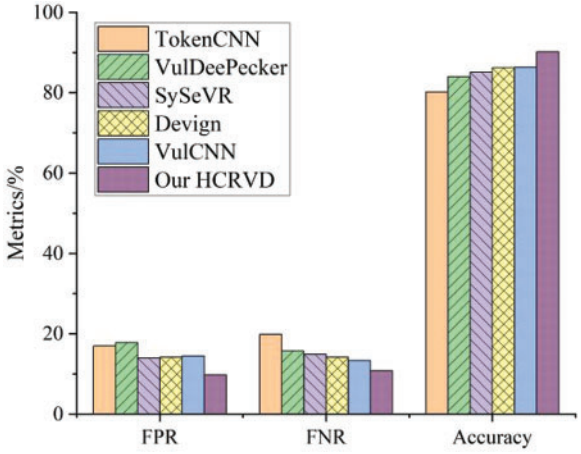

Figure 12: Comparison of false positives rate, false negative rate, and accuracy of TokenCNN, VulDeePecker, SySeVR, Devign, VulCNN, and HCRVD on C/C++ test dataset

These vulnerability detection models represent the detection capabilities under different code representation methods. TokenCNN classifies code by converting it into tokens encoded into fixed-length vector for TextCNN. It does not take advantage of the structure of the code that is rich in syntax and semantic information, and thus has the worst detection performance. VulDeePecker uses sequence-based code representation to slice the code into code gadgets based on the vulnerability information, the code gadgets are converted into token encoded as vectors to be input into bidirectional RNN for classification and detection. This approach further extracts vulnerability features using the original vulnerability-related information of the code (API, etc.) and the structural information of the code (data flow, etc.) before encoding, so it is better than TextCNN. SySeVR is a further optimized version of VulDeePecker, which takes into account the data and control dependency to obtain slices, and therefore improves the classification performance. Devign represents a class of representations that directly use the code structure to construct neural network structures. Devign expands the edges related to the program structure to form a graph based on an AST, which is fed into the GNN for detection. But it has the drawback that the code features are flattened and all code structures are treated as the same connection. And the nodes of the graph take the nodes of the AST as the smallest granularity,

but the smallest granularity that carries the semantics is the statement. Even so, it is still able to extract more features relative to the sequence representation method, so it outperforms the sequence-based representation detection method in terms of classification performance. VulCNN encodes the codes as images for classification. Its core feature is the utilization of the degree-weighted node vectors of the PDG to generate multi-channel data. It uses a graph-based code representation, which is still essentially the utilization of PDG, and its detection ability is second only to HCRVD, with an accuracy of 86.381%, a FPR of 14.45%, and a FNR of 13.345%.

Our system HCRVD based on CP hierarchical code representation takes statements as the basic semantic nodes, and for the information above the granularity of statements, PDG is utilized to extract more effective structural information, while CST is used for the hierarchy below the statements, which not only preserves the lexical information of the code text itself, but also abstracts a lot of code syntax information of the code. Therefore, HCRVD achieves 90.19% in terms of validity, which is at least 3.81% higher than the existing methods; the FPR is 9.792%, and the FNR is 10.809%, which are all significantly reduced compared to other detection methods.

Therefore, for **RQ1**, we can get:

Insight1: HCRVD system has excellent validity in vulnerability detection. Compared to other methods, there is a significant improvement of at least 3.81% in accuracy, and FPR and FNR of detection are decreased by at least 4.173% and 2.536%. in the same monolingual dataset.

4.3.2 Results and Analysis of Hierarchical Code Representation Learning Method Ablation Study

HCRVD excels in validity, but it remains uncertain whether this stems from the fact that the features extracted by the hierarchical code representation learning approach are more effective. We performed different ablation experiments for the code hierarchical representation learning approach, and in Table 2, we show the performance of the model using different combinations of code representations and neural networks for vulnerability detection. We set up these experiments to explore whether the proposed CP data structure (compared to CST, PDG, Code Property Graph (CPG)), and TGGA network (compared to Bidirectional GRU (BGRU), GGNN, and Tree-GRU) are the key factors for the model performance improvement. The tests also use the same C/C++ dataset as in Section 4.3.1.

Table 2: Performance of models based on different code representation learning methods

Code representation learning method	FPR (%)	FNR (%)	Acc (%)
CST with BGRU	16.186	15.986	84.016
PDG with BGRU	14.971	15.764	84.381
PDG with GGNN	14.307	14.196	85.504
CST with Tree-GRU	14.161	13.709	85.891
CPG with GGNN	15.884	13.040	86.853
CP with Tree-Graph-GRU	11.296	11.259	88.741
Our HCRVD (CP with TGGA)	**9.792**	**10.809**	**90.191**

Among them, CP with TGGA is the code representation and neural network used in HCRVD, and CP with Tree-Graph-GRU does not use attention readout between layers. The performance of the model is degraded compared to HCRVD, which demonstrates the effect of using attention.

PDG with GGNN is a model that uses only the PDG structure to extract the graph data and encodes the node statements as vectors using the sent2vec model, and then inputs it into GGNN for classification and detection. CST with Tree-GRU is a model that encodes the whole code as a CST, and each node adopts word2vec to encode the identifications as vectors, which are input into Tree-GRU for classification and detection. They have similar detection performance, but are worse compared to CP with Tree-Graph-GRU. In addition, we further explore the use of CPG representation for detection. CPG contains the edges of CFG, AST and PDG, which has more information, and thus CPG with GGNN achieves better performance than the previous two. However, it is not the case that the more edges included are more helpful for the improvement of the model detection capability because they may not be suitable for the vulnerability although they represent the code abstract structure. These edges may provide connection information outside the propagation path of the vulnerability, affecting the information transmission of model nodes. So CPG with GGNN has worse detection performance compared to CP with Tree-Graph-GRU. These results suggest that the code representation affects the effectiveness of vulnerability detection, and that CP hierarchical code representation can improve the performance of the system.

PDG with BGRU is a model that extracts graph structural information based on PDG and encodes nodes with sent2vec for classification detection using BGRU. Compared to the model using GGNN, this model directly inputs the vectors of each node into the BGRU network without building the network based on the representation structure of the code, losing the network's ability to learn the structural information, and decreasing the model detection ability. A similar situation occurs for CST with BGRU relative to CST with Tree-GRU. This suggests that constructing neural networks having the same structural features as the code representation helps the network to learn the semantics of the code structure and improves the detection, and Tree-Graph-GRU is able to effectively learn the code representation of the CP structure.

Therefore, for **RQ2**, we can get:

Insight2: The hierarchical code representation learning method makes HRCVD more effective. CP structure can extract more code syntax and semantic information, Tree-Graph-GRU can effectively learn the data of this structure, and the attention mechanism can further improve the connection between layers to enhance the system performance. Using the hierarchical code representation learning method improves the accuracy by at most 6.175% than a single feature.

4.3.3 Results and Analysis of Cross-Language Vulnerability Detection

Previous vulnerability detection systems often performed well in single language, but performance dropped when applied to multiple languages and required additional effort. The reason is that, first, in the extraction of code abstraction data, different tools are often required for different programming languages, which means repeated and tedious work. Second, different programming languages produce different tokens, which makes cross-language feature extraction difficult. Furthermore, the features of different programming languages differ, and their input and target distributions are also different, leading to the fact that it may be difficult for neural networks to learn common features.

To address these issues, first, our HCRVD normalizes all code data, removing as much information as possible that is irrelevant to program semantics, and avoiding models that associate vulnerabilities with particular patterns. Second, using the same code analysis tool to perform PDG extraction and AST generation for datasets in different languages separately normalizes the distribution of the dataset. Third, HCRVD encodes code through CP structure, which introduces syntactic structure information (non-leaf nodes) in the lowest level CST structure and adds numerous tokens, thus

reducing the proportion of cross-code differentiated feature identifiers, and the dictionary difference between different languages only lies in the standard library functions. The upper level PDG structure extracts the code abstractions common to all programming languages, which somewhat weaken the heterogeneity of languages. Finally, the use of the TGGA network for training also to a certain extent shielded the cross-language differentiated features in the leaf nodes at the bottom of the network structure.

Using the HCRVD system identical to Section 4.3.1, we train and test on the full C/C++ & Java dataset to test the generality of HCRVD for cross-language vulnerability tasks. To make the results more concise, we only select VulCNN as the baseline for this experiment since it is the most effective among all baseline methods in Section 4.3.1. Another reason is that the PDG code representation based on sent2vec and graph centrality used by VulCNN allows the extraction of common features in cross-language codes, which makes it more effective in cross-language testing than other methods. The results in Table 3 show that the trained HCRVD can effectively detect vulnerabilities on both languages at the same time, and can achieve 88.417% accuracy on the C/C++ dataset, which is only 1.774% lower than the accuracy on a single dataset. On the Java dataset, it can achieve 94.187% accuracy. Thus HCRVD is shown to have cross-language vulnerability detection capability. While the trained VulCNN performs far worse than HCRVD on both datasets, and its detection performance on the C/C++ dataset shows a significant decline, with a 7.736% decrease in classification accuracy, an 11.925% increase in FPR, and a 7.99% increase in FNR. This suggests that VulCNN has difficulty in learning vulnerability features from different programming languages, and the heterogeneity of the languages has an impact on the performance of the model. Comparing HCRVD with baseline, it can be further concluded that HCRVD has excellent generality in cross-language code vulnerability detection tasks. The CP hierarchical code representation adopted by HCRVD can better represent the general features of the code, and the extracted code vulnerability features combined with the TGGA network are more universal, which can shield the heterogeneity of different programming languages to a certain extent.

Table 3: Model performance in cross-language vulnerability detection

Model	FPR (%)	FNR (%)	Acc (%)
Baseline tests on C/C++	26.375	21.355	78.645
Baseline tests on Java	17.630	17.632	82.368
Trained HCRVD tests on C/C++	**11.109**	**11.583**	**88.417**
Trained HCRVD tests on Java	**7.012**	**5.813**	**94.187**

Therefore, for **RQ3**, we can get:

Insight3: The trained HCRVD system have the ability to detect vulnerabilities across languages, and the detection performance is only decreased by 1.774% compared to that on a single language dataset. HCRVD is 9.772% and 11.819% better than baseline in cross-language vulnerability detection in C/C++ & Java datasets, respectively. HCRVD can weaken the heterogeneity of language through CP hierarchical code representation and TGGA network, and has better generality in cross-language vulnerability detection.

4.3.4 Results and Analysis of Unknown Programming Language Vulnerability Detection

In general, it is not possible to detect whether code contains vulnerabilities on unknown programming languages. The so-called unknown programming languages are languages other than that used in the training set. In other words, data from unknown languages are not involved in composing the training set at all, and they are completely unfamiliar to the model before participating in detection. First, because the model lacks a dictionary on that language, this results in much of the valid information being encoded as <UNK> tokens. Second, the feature extraction way learned by the model on one language may not be applicable to another programming language, which makes the model's effect greatly reduced or even ineffective.

Given HCRVD's strong detection capabilities on cross-language datasets, we wanted to test whether a HCRVD system trained on one language could work on another never-before-seen language without transfer learning. This can only be done if the model truly learns the logical structure associated with the triggering of vulnerability. Such an idea comes from the fact that, first, HCRVD normalizes the code and extracts CP data using unified tools, which allows codes in different languages to produce dictionary with many shared tokens. For example, the number of tokens shared in the dictionaries of the C/C++ and Java datasets reached 93, which means even with an unknown language, the HCRVD still retains a great portion of the code text information. Second, the CP hierarchical code representation generates a highly unified data structure, and the TGGA network not only benefits from the lexical information provided by the token of the code, but also benefits from this CP hierarchical structure. This abstract feature common to all code ensures the consistency of the extracted code features on the unknown language, and the model has the potential to detect vulnerabilities on code in unknown languages. These features of HCRVD solve the two major problems mentioned above. To validate our idea, we use the HCRVD system trained on the C/C++ dataset, without migration learning and completely unknown to the Java data, to detect vulnerabilities on the Java dataset. We use the same HCRVD as mentioned in Section 4.3.1 for testing on the Java full dataset, which is encoded using the dictionary on C/C++.

The results in Table 4 shows that even though the Java dataset has never been seen before, HCRVD still achieves a test accuracy of 88.964% on the full Java dataset. This demonstrates the strong generality of HCRVD. Consistent with our idea, although the unknown language Java part of the code is encoded as a vector corresponding to <UNK>, there is still most of the structural information preserved, including the semantic information of the statements extracted by CST. HCRVD can use not only the code text information but also the structured information of the code when detecting vulnerabilities, so even if a part of the text information is lost, HCRVD is still able to obtain the characteristic paradigm of the vulnerability from the unified structured information and detect the vulnerability effectively. Like the previous section, we only select VulCNN, which is the most effective method of all the baselines in Section 4.3.1 and uses a more generalized code representation than other baselines, as the baseline method for this experiment to make the results more concise. VulCNN is trained on the C/C++ dataset and tested on the Java dataset. The test results show that VulCNN is almost impossible to detect vulnerabilities without training on the Java dataset. The accuracy of the experiment is only slightly better than that of the random (50%) and the FPR is more than 50%. Such a model is unusable. Thus, we see that HCRVD enables vulnerability detection on unknown programming languages, which far exceeds baseline's near-random detection performance.

Table 4: Model performance in unknown programming language vulnerability detection

Model	FPR (%)	FNR (%)	Acc (%)
Baseline train on C/C++ and test on Java	67.042	41.331	54.189
HCRVD train on C/C++ and test on Java	**9.138**	**11.036**	**88.964**

Therefore, for **RQ4**, we can get:

Insight4: HCRVD is able to detect vulnerabilities in an unknown programming language to a certain extent without having seen the language, and the accuracy exceeds the baseline by 34.775%. HCRVD is effective and has strong generality.

4.3.5 Results and Analysis of Practicality Testing Experiment

To validate the practicality of HCRVD in detecting vulnerabilities in real-world software, we applied HCRVD to real open-source projects, Libav and Novel-plus. We chose these two projects as examples because they are open-source software written in C and Java respectively and they are reported to have vulnerabilities. Table 5 lists the major programming languages, versions, and number of function-level vulnerabilities for each project. We use the HCRVD trained in Section 4.3.3 to detect vulnerabilities in them, and still use the best performing VulCNN from Section 4.3.1 as the baseline for this experiment to make the results more concise. In order to test whether the model can truly understand the vulnerability pattern, we use the five model weights obtained from training in five-fold cross-validation to perform the tests separately, and only keep the results that are the same in five tests, and the other cases are labeled as undetermined, which can eliminate the chance of the test results.

Table 5: Open-source project validation dataset

Project	Language	Version	#vuln.
Libav	C	v12.3	12
Novel-plus	Java	v3.6.2, v3.6.1	5

We extracted CP data from their source code, collected all the reported vulnerability warnings, and manually analyzed these warnings. Table 6 summarizes the vulnerabilities that were successfully identified for each project, where "✓" denotes a successfully detected vulnerability, "×" denotes a function identified as non-vulnerable, and "◯" denotes an undetermined detection results. Each vulnerability corresponds to a Common Vulnerabilities and Exposures Identifier (CVE ID). The results showed that HCRVD detected 14 out of 17 vulnerabilities in two open-source projects written in different languages, while 1 out of the other 3 vulnerabilities was incorrectly identified as a non-vulnerable function, and 2 were flagged as undetermined due to different results of multiple identifications. In addition, HCRVD has identified 2 vulnerabilities in other versions of Novel-plus that have been reported in previous versions but remain unpatched. In contrast, baseline VulCNN detected only 6 vulnerabilities in Libav and 2 vulnerabilities in Novel-plus, with a total of 6 vulnerable functions labeled as undetermined, while other vulnerabilities were not detected. Such results show that HCRVD can function in real-world open-source projects and has more practicality in cross-language tasks without the need for repetitive and cumbersome system migrations.

Table 6: Vulnerabilities identified in open-source projects using different models

CVE ID	VulCNN	HCRVD	CVE ID	VulCNN	HCRVD
CVE-2018-6621	✓	✓	CVE-2020-18775	✓	✓
CVE-2018-11102	×	×	CVE-2020-18776	✓	✓
CVE-2018-18828	◯	✓	CVE-2020-18778	✓	◯
CVE-2018-18829	◯	✓			
CVE-2018-19128	✓	✓	CVE-2021-30048	✓	✓
CVE-2018-19130	✓	✓	CVE-2021-42967	◯	✓
CVE-2019-9720	×	✓	CVE-2022-35121	✓	✓
CVE-2019-14442	◯	✓	CVE-2023-1594	×	◯
CVE-2019-14443	◯	✓	CVE-2023-1606	◯	✓

Therefore, for **RQ5**, we can get:

Insight5: HCRVD can be used in real projects in different languages with practicality, and it has more effective detection ability in real-world cross-language vulnerability detection tasks.

5 Discussion

In summary, the model proposed in this paper can improve detection performance to a certain extent in the task of source code vulnerability detection, and demonstrates significant advantages and immense potential in the field of cross-language vulnerability detection.

However, this paper also has certain limitations. HCRVD currently only provides functional-level source code vulnerability detection, and manual analysis is still required to find more fine-grained vulnerability locations. Although hierarchical code representation learning method has the potential to detect vulnerabilities at a finer granularity, further efforts are needed to reach this point.

Therefore, our future work will focus on further refining the detection granularity of HCRVD to the statement or even token level. In addition, considering the effectiveness of hierarchical code representation learning method, it is also a promising direction to apply it to other code-related tasks.

6 Conclusion

In this paper, we propose a new CP hierarchical code representation which an reflect the structure and characteristics of source code more comprehensively. Additionally, we develop a novel TGGA neural network to learn vulnerability patterns from the CP representation. We further design the cross-language source code vulnerability detection system HCRVD. This system converts source code into the CP representation and feeds it into the TGGA neural network. The trained model is then utilized to detect vulnerabilities. Extensive experiments demonstrate the validity, generality and practicality of HCRVD in cross-language vulnerability detection.

Acknowledgement: The authors would like to express their gratitude for the valuable feedback and suggestions provided by all the anonymous reviewers and the editorial team.

Funding Statement: This study was funded by the Major Science and Technology Projects in Henan Province, China, Grant No. 221100210600.

Author Contributions: The authors confirm contribution to the paper as follows: Study conception and design: Zhihui Song, Zheng Shan; data collection: Zhihui Song, Kewei Li; conduct of the experiments: Zhihui Song; analysis and interpretation of result: Zheng Shan, Jinchen Xu; draft manuscript preparation: Zhihui Song. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author, Zheng Shan, upon reasonable request.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Synopsys.com, "Open-source security and analysis report," Accessed: Oct. 18, 2023. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[2] "Flawfinder," Accessed: Oct. 18, 2023. [Online]. Available: https://dwheeler.com/flawfinder/

[3] "Clang static analyzer," Accessed: Oct. 18, 2023. [Online]. Available: https://clang-analyzer.llvm.org/

[4] "Application security testing company, Checkmarx," Accessed: Oct. 18, 2023. [Online]. Available: https://checkmarx.com/

[5] "Fortify," Accessed: Oct. 18, 2023. [Online]. Available: https://www.microfocus.com/en-us/cyberres/application-security

[6] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 595–614. doi: 10.1109/SP.2017.62.

[7] Y. Xiao *et al.*, "MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures," presented at the 29th USENIX Secur. Symp. (USENIX Security 20), 2020, pp. 1165–1182.

[8] H. Sun *et al.*, "VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Comput. Secur.*, vol. 110, no. 5–6, pp. 102417, Nov. 2021. doi: 10.1016/j.cose.2021.102417.

[9] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Orlando, FL, USA, Dec. 2018, pp. 757–762. doi: 10.1109/ICMLA.2018.00120.

[10] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Appl. Sci.*, vol. 10, no. 5, pp. 1692, 2020. doi: 10.3390/app10051692.

[11] 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, pp. 106809, 2022. doi: 10.1016/j.infsof.2021.106809.

[12] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *2018 Proc. NDSS, Internet Society*, Reston, VA, USA, 2018. doi: 10.14722/ndss.2018.23158.

[13] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Depend. Secure.*, vol. 18, no. 5, pp. 2224–2236, Sep. 2021. doi: 10.1109/TDSC.2019.2942930.

[14] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Depend. Secure.*, vol. 19, no. 4, pp. 2244–2258, Jul. 2022. doi: 10.1109/TDSC.2021.3051525.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2019.

[16] H. Wang *et al.*, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Trans. Inf. Foren. Sec.*, vol. 16, pp. 1943–1958, 2021. doi: 10.1109/TIFS.2020.3044773.
[17] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, 2021. doi: 10.1145/3436877.
[18] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *J. Syst. Software*, vol. 199, no. 3, pp. 111623, May 2023. doi: 10.1016/j.jss.2023.111623.
[19] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. 19th Int. Conf. Mining Softw. Repositories*, New York, NY, USA, Association for Computing Machinery, Oct. 2022, pp. 596–607. doi: 10.1145/3524842.3527949.
[20] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Austin, TX, USA, ACM, Dec. 2022, pp. 481–496. doi: 10.1145/3564625.3567985.
[21] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, no. 106576, pp. 106576, 2021. doi: 10.1016/j.infsof.2021.106576.
[22] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, 2022. doi: 10.1109/TSE.2021.3087402.
[23] W. Li, X. Li, W. Feng, G. Jin, Z. Liu and J. Jia, "Vulnerability detection based on unified code property graph," in L. Yuan, S. Yang, R. Li, E. Kanoulas, X. Zhao (Eds.), *Web Information Systems and Applications, Lecture Notes in Computer Science*, Singapore: Springer Nature, 2023, pp. 359–370. doi: 10.1007/978-981-99-6222-8_30.
[24] Y. Mirsky *et al.*, "VulChecker: Graph-based vulnerability localization in source code," presented at the 32th USENIX Secur. Symp. (USENIX Security 23), 2023.
[25] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proc. 44th Int. Conf. Softw. Eng.*, New York, NY, USA, Association for Computing Machinery, Jul. 2022, pp. 2365–2376. doi: 10.1145/3510003.3510229.
[26] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger, and S. Chakraborty, "Automatic identification of vulnerable code: Investigations with an AST-based neural network," in *2021 IEEE 45th Annu. Comput., Softw. Appl. Conf. (COMPSAC)*, Madrid, Spain, Jul. 2021, pp. 1475–1482. doi: 10.1109/COMPSAC51774.2021.00219.
[27] C. Zhang, B. Liu, Y. Xin, and L. Yao, "CPVD: Cross project vulnerability detection based on graph attention network and domain adaptation," *IEEE Trans. Software. Eng.*, vol. 49, no. 8, pp. 4152–4168, Aug. 2023. doi: 10.1109/TSE.2023.3285910.
[28] N. T. Islam, G. D. L. T. Parra, D. Manuel, E. Bou-Harb, and P. Najafirad, "An unbiased transformer source code learning with semantic vulnerability graph," Accessed: Jan. 02, 2024. [Online]. Available: https://arxiv.org/abs/2304.11072v1
[29] SARD, "NIST software assurance reference dataset," Accessed: Oct. 18, 2023. [Online]. Available: https://samate.nist.gov/SRD/index.php
[30] Nvd.nist.gov, "National vulnerability database," Accessed: Oct. 18, 2023. [Online]. Available: https://nvd.nist.gov
[31] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in *2022 Int. Joint Conf. Neural Netw. (IJCNN)*, Padua, Italy, Jul. 2022, pp. 1–8. doi: 10.1109/IJCNN55064.2022.9892280.
[32] K. Napier, T. Bhowmik, and S. Wang, "An empirical study of text-based machine learning models for vulnerability detection," *Empir. Softw. Eng.*, vol. 28, no. 2, pp. 38, Feb. 2023. doi: 10.1007/s10664-022-10276-6.
[33] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in J. Burstein, C. Doran, T. Solorio (Eds.), *Proc. 2019 Conf. North American*

Chapter Assoc. Comput. Linguist.: Hum. Lang. Technol., Minneapolis, Minnesota, Association for Computational Linguistics, 2019, vol. 1, pp. 4171–4186. doi: 10.18653/v1/N19-1423.

[34] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A robustly optimized BERT pre-training approach with post-training," in G. Li *et al.* (Eds.), *Proc. 20th Chinese National Conf. Computat. Linguist.*, Huhhot, China: Chinese Information Processing Society of China, 2021, pp. 1218–1227. Accessed: Mar. 19, 2024. [Online]. Available: https://aclanthology.org/2021.ccl-1.108

[35] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Depend. Secure.*, vol. 19, no. 4, pp. 2821–2837, Jul. 2022. doi: 10.1109/TDSC.2021.3076142.

[36] Y. Hu *et al.*, "Interpreters for GNN-based vulnerability detection: Are we there yet?," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, New York, NY, USA, Association for Computing Machinery, Jul. 2023, pp. 1407–1419. doi: 10.1145/3597926.3598145.

[37] X. Li, Y. Xin, H. Zhu, Y. Yang, and Y. Chen, "Cross-domain vulnerability detection using graph embedding and domain adaptation," *Comput. Secur.*, vol. 125, pp. 103017, 2023. doi: 10.1016/j.cose.2022.103017.

[38] J. Zhang *et al.*, "Novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Montreal, QC, Canada, May 2019, pp. 783–794. doi: 10.1109/ICSE.2019.00086.

[39] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014. doi: 10.48550/arXiv.1412.3555.

[40] A. Vaswani *et al.*, "Attention is all you need," *Adv. Neural Inf. Process. Syst.*, vol. 30, pp. 6000–6010, Dec. 2017.

[41] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Networ.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. doi: 10.1109/72.279181.

[42] D. Buterez, J. P. Janet, S. J. Kiddle, D. Oglic, and P. Liò, "Graph neural networks with adaptive readouts," *Adv. Neural Inf. Process. Syst.*, vol. 35, pp. 19746–19758, Dec. 2022.

[43] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015. doi: 10.48550/arXiv.1511.05493.

[44] "Joern–The bug hunter's workbench," Accessed: Oct. 18, 2023. [Online]. Available: http://joern.io

[45] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604. doi: 10.1109/SP.2014.44.

[46] L. Cui, Z. Hao, Y. Jiao, H. Fei, and X. Yun, "VulDetector: Detecting vulnerabilities using weighted feature graph comparison," *IEEE Trans. Inf. Foren. Sec.*, vol. 16, pp. 2004–2017, 2021. doi: 10.1109/TIFS.2020.3047756.

[47] "Tree-sitter," Accessed: Oct. 18, 2023. [Online]. Available: https://tree-sitter.github.io