Journal of Information Security and Applications 89 (2025) 103973

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

CodeSAGE: A multi-feature fusion vulnerability detection approach using code attribute graphs and attention mechanisms

Guodong Zhang [a], Tianyu Yao [a], Jiawei Qin [c], Yitao Li [c], Qiao Ma [d], Donghong Sun [b,*]

[a] Shenyang Aerospace University, School of Computer, Shenyang, 110000, Liaoning Province, China
[b] Tsinghua University, Institute for Network Sciences and Cyberspace, 100084, Beijing, China
[c] National Computer Network Emergency Response Technical Team/Coordination Center of China, 100029, Beijing, China
[d] National Computer Network Emergency Response Technical Team/Coordination Center of China, Liaoning Branch Company, Dalian, 116021, Liaoning Province, China

ARTICLE INFO

Keywords:
Software supply chain security
Code property graph
Simplification
Vulnerability detection
Feature fusion

ABSTRACT

Software supply chain security is a critical aspect of modern computer security, with vulnerabilities being a significant threats. Identifying and patching these vulnerabilities promptly can significantly reduce security risks. Traditional detection methods cannot fully capture the complex structure of source code, leading to low accuracy. The neural network capacity limits machine learning-based methods, hindering effective feature extraction and impacting performance. In this paper, we propose a multi-feature fusion vulnerability detection technique called CodeSAGE. The method utilizes the Code Property Graph (CPG)[1] to comprehensively display multiple logical structural relationships in the source code and combine it with GraphSAGE to aggregate the information of neighboring nodes in CPG to extract local features of the source code. Meanwhile, a Bi-LSTM combined with the attention mechanism is utilized to capture long-range dependencies in the logical structure of the source code and extract global features. The attention mechanism is used to assign weights to the two features, which are then fused to represent the syntactic and semantic information of the source code for vulnerability detection. A method for simplifying the CPG is proposed to mitigate the impact of graph size on model runtime and reduce redundant feature information. Irrelevant nodes are removed by weighting different edge types and filtering nodes exceeding a certain threshold, reducing the CPG size. To verify the effectiveness of CodeSAGE, comparative experiments are conducted on the SARD and CodeXGLUE datasets. The experimental results show that the CPG size can be reduced by 25%–45% using the simplified method, with an average time reduction of 20% per training round. Detection accuracy reached 99.12% on the SARD dataset and 73.57% on the CodeXGLUE dataset, outperforming the comparison methods.

1. Introduction

With the rapid development of Internet technology, the Internet has become integrated into people's daily lives and work. According to the International Telecommunication Union (ITU) [1], as of 2023, there are 4.7 billion Internet users globally, with a penetration rate of about 60%. Meanwhile, the popularity of video streaming services (e.g., Netflix (1997) [2], YouTube (2005) [3]) is driving rapid growth in global data traffic.

However, the widespread use of Internet technology poses challenges to software supply chain security. In July 2024, cryptocurrency exchange WazirX suffered a significant hacking attack with losses of $230 million. The attackers exploited a vulnerability in the WazirX platform and managed to steal many crypto assets, leading to a significant drop in user trust [4]. Therefore, it is better to identify and patch vulnerable code in the software supply chain ahead of time than to remediate vulnerabilities after they have caused financial losses and privacy breaches (Piergiorgio Ladisa et al., 2023) [5].

Aiming to address the above problems, this paper proposes a multi-feature fusion approach for vulnerability detection. The method not only considers the direct representation of the source code but also converts the source code into a CPG to illustrate the logical structure of the code. It also makes targeted simplifications to reduce the size of the CPG and remove information unrelated to vulnerabilities. Additionally, considering both global contexts information and local structural characteristics of the source code, Bi-LSTM (Sepp Hochreiter

* Corresponding author.
E-mail address: sundonghong@tsinghua.edu.cn (D. Sun).
[1] CPG: Code Property Graph.

https://doi.org/10.1016/j.jisa.2025.103973

Available online 28 January 2025

and Jürgen Schmidhuber, 1997) [6] is used to extract global features from the source code, while GraphSAGE (Will Hamilton et al. 2017) [7] is used to extract local features from the simplified CPG, achieving a more comprehensive feature extraction of the source code. Finally, the attention mechanism is utilized to effectively fuse the two feature vectors, enhancing the detection of vulnerabilities in the source code of the software supply chain. In summary, the main contributions of this paper can be summarized as follows:

(1) A method for simplifying the CPG was designed and implemented. The method selects specific code based on rules and traverses all code slices related to the selected code, excluding source code unrelated to the vulnerability. It effectively reduces the size of CPG and improves the runtime without affecting vulnerability detection accuracy.

(2) A vulnerability detection method called CodeSAGE is proposed. By using Bi-LSTM and GraphSAGE, which focus on extracting global and local features of the source code, respectively, features from different perspectives of the source code are obtained more comprehensively. On this basis, a fusion strategy using the attention mechanism (Ashish Vaswani et al., 2017) [8] is employed to combine feature vectors of different dimensions effectively, thereby avoiding the overfitting phenomenon caused by the model's complexity.

The rest of the paper is organized as follows: Section 2 describes the current approaches to vulnerability detection in the domain. Section 3 outlines the framework of the CodeSAGE vulnerability detection methodology. Section 4 details the code generation process for CPG and the principles of simplified CPG. Section 5 covers the detailed composition of the CodeSAGE model. Section 6 discusses the evaluation metrics and results of the experiments. Section 7 presents the conclusions and future work.

2. Background

Vulnerability detection is the process of identifying flaws and weaknesses in the source code of a software supply chain to improve its security. This process helps prevent information leakage and unauthorized data tampering. Currently, two main approaches are used for vulnerability detection: dynamic and static.

2.1. Dynamic detection methods

Dynamic detection refers to methods that execute programs in a controlled the environment and monitored their behavior to identify vulnerabilities. In this area, Janaka Senanayake et al. (2024) [9] proposed Defendroid, which detects Android code vulnerabilities in real time using a blockchain-based federated neural network with explainable AI (XAI). Mojtaba Eshghie et al. (2021) [10] proposed the Dynamit monitoring framework for detecting reentrant vulnerabilities in Ethereum smart contracts, which achieved more than 90% accuracy on 105 transactions. Li J. et al. (2016) [11] proposed a dynamic taint propagation method for detecting XSS vulnerabilities achieves automated detection through function hijacking of tracking data streams. However, the method may suffer from false positives and omissions when dealing with unfiltered samples.

In summary, dynamic detection focuses on identifying vulnerabilities along the actual execution path of a program but may not be able to address vulnerabilities under specific conditions. This limitation suggests that dynamic detection methods should be complemented by other detection methods for a more comprehensive security assessment.

2.2. Static detection methods

To address the limitations of dynamic detection, static detection methods have gained increasing attention. Unlike dynamic detection, which monitors programs in real time, static detection analyzes the source code before the program is executed. Early-stage analysis can detect vulnerabilities before deployment, reducing the complexity and cost of subsequent fixes. Static detection methods are currently categorized into two types: those based on code similarity and those based on specific patterns or rules.

2.2.1. Methods based on code similarity

Code similarity-based approaches aim to identify potential vulnerabilities by detecting code fragments similar to known security pitfalls. For instance, Kim et al. (2018) [12] developed the VUDDY tool, which pinpointed 144,496 potentially vulnerable functions across more than 14 billion lines of code. Lei Cui et al. (2020) [13] extended this concept to graph similarity, leveraging graphs to represent the logical relationships within code better. Hao Sun et al. (2021) [14] employed a small dataset focusing on vulnerabilities and their patches to enhance security similarity. However, like other code similarity approaches, it may still struggle with false negatives when identifying specific vulnerability types.

2.2.2. Detection methods for specific patterns or rules

Detection methods based on specific patterns or rules can be categorized into manually defined rules and machine learning techniques. The former helps developers identify security risks through predefined rules or patterns, as exemplified by tools like Checkmarx [15], Klocwork [16], SonarQube [17], and Flawfinder (2018) [18]. These tools mark vulnerability locations in code but are limited by their preset rules and cannot identify new vulnerability types. In contrast, machine learning-based methods learn vulnerability patterns from data using algorithms that do not rely on predefined rules. For instance, William Melicher et al. (2021) [19] explored using machine learning classifiers to enhance DOM and XSS vulnerability detection beyond traditional taint tracking methods. Russell et al. (2018) [20] achieved performance gains by combining convolutional neural networks with random forest classifiers for classification tasks. However, machine learning-based approaches require substantial training data and feature engineering, typically operating at a coarse-grained level (e.g., file or function), which may limit localization accuracy.

2.3. Deep learning methods

In recent years, the application of deep learning techniques to vulnerability detection has emerged as a leading-edge area of research. For instance, Li Z. et al. (2018) [21] extracted code fragments related to library/API function calls and trained them using Bi-LSTM. Building on this idea, Van-Anh Nguyen et al. (2022) [22] treated vulnerability detection as an inductive text classification problem, constructing graphs from source code tokens. Teresa K. George et al. (2018) [23] proposed TbD-NNbR, a token-based detection and neural network-based reconstruction framework to detect and block code injection with negligible processing overhead.

Studies have shown that using graphs to model logical structures in source code is more effective than directly converting them to an intermediate language. For instance, Yaqin Zhou et al. (2019) [24] achieved C language vulnerability detection by integrating multiple graphical representations and employing graph neural networks to learn semantic information from the source code. Hoang Viet Nguyen et al. (2022) [25] enhanced the graphical representation of source code using CPG (Bowman et al., 2020) [26], incorporating a dominance tree and constructing a classifier with multiple words embedding methods and a deep Graph Neural Network(GNN) to provide a multidimensional view of code structure. Yueming Wu et al. (2022) [27] analyzed code as an interpersonal graph, efficiently identifying vulnerability patterns and offering new perspectives in vulnerability detection. It is even possible to focus on detecting a specific type or category of vulnerability by improving the graphical structure. For example, Feng Luo et al. (2024) [28] designed a vulnerability detection approach

for smart contracts, incorporating a heterogeneous semantic graph construction phase based on intermediate representations, followed by a vulnerability detection phase using a heterogeneous graph attention network. However, graphical representations often yield large-scale graphs, leading to high computational complexity, increased time, and memory overhead. For instance, analyzing the Linux kernel source code (Fabian Yamaguchi et al., 2014) [29], which spans over 20 million lines, generated a CPG with millions of nodes and edges, requiring substantial computational resources, tens of gigabytes of memory, and processing times spanning days.

Furthermore, after analyzing the detection results, it was found that a single neural network model could not adequately capture all the feature information in the code, which affects the vulnerability detection results. To address this, Min Ling et al. (2023) [30] represent source code as an Abstract Syntax Tree (AST), employs Graph Convolutional Networks (GCN) to extract structural features and utilizes Bi-LSTM to capture relational and sequential information from the graph. In a similar vein, Wenbo Wang et al. (2023) [31] extracted information from three separate graphs using distinct models and aggregated information to enhance comprehensiveness. Mingwei Tang et al. (2024) [32] proposed a Residual Graph Attention Network (RGAN), which improves the combination of local and global feature information, thereby enriching the feature set. Zhonglin Liu et al. (2023) [33] used GCN and Bidirectional Recurrent Neural Networks (Bi-RNN) to extract and combine the logical call and contextual execution features of the source code for detecting XSS vulnerabilities in JavaScript code.

In summary, this paper aims to address the current challenges in detecting source code vulnerabilities in the software supply chain, including inadequate representation of logical relationships in code (Section 4.1), the impact of large method sizes on run rates and the presence of extraneous code affecting detection accuracy (Section 4.2), and incomplete feature extraction by models (Section 5).

3. Overall framework

This section outlines the framework of the CodeSAGE approach, which aims to address the above challenges. The method utilizes CPG to preserve the logical structure of the code and applies algorithmic rules to filter specific code lines for traversal, resulting in a simplified CPG. Concurrently, Bi-LSTM and GraphSAGE extract different levels of code logic features, which are then fused to enhance feature diversity. The basic flow of the method is shown in Fig. 1. It can be subdivided into four essential parts.

The Source Code Normalization and Input section involves preprocessing operations on the source code dataset through standardization to ensure uniformity in the input source code format. This step simplifies subsequent processing and reduces issues related to linguistic variations caused by user-defined names.

The main goal of the second part is to simplify the CPG and extract local feature vectors. The source code is converted to CPG, and an algorithm is used to streamline it. After simplification, the GraphCodeBERT (Daya Guo et al., 2021) [34] preprocessing model encodes the node information and captures the syntactic and semantic details. GraphSAGE's operations such as sampling and aggregation, integrate the node information with that of the neighboring nodes. Unlike GCN, GraphSAGE involves all nodes in the training process but can be trained locally on selected nodes, making it more suitable for extracting local features. The convolutional layer further enriches the code features after aggregation and uses the resultant output as input for the fusion part.

The third section focuses on extracting global feature vectors from the source code. The code is encoded using the pre-trained model CodeBERT (Zhangyin Feng et al., 2020) [35], a Bi-LSTM layer is then used to capture the contextual relationships in the code. An attention layer is subsequently employed to address the limitations of Bi-LSTM in capturing long-sequence dependencies. This section is not coded using the simplified CPG because we focus on extracting global features, and a simplified CPG would compromise the integrity of these features. The output of the Attention Layer is then passed through a Dropout Layer to minimize overfitting before being used as input to Part IV.

The fourth part integrates the local and global features extracted from the second and third parts use respective attention mechanisms to combine their weighted contributions. These fused features are dimensionally reduced through a fully connected layer, which then outputs the final vulnerability classification results.

Table 1
Different codes indicate the type of vulnerability identified.

Vulnerability Type	AST	AST + DFG	AST + CFG	AST + CFG+DFG
Memory Disclosure	no	no	no	yes
Buffer Overflow	no	yes	no	yes
Resource Leaks	no	no	yes	yes
Design Errors	no	no	no	no
Null Pointer Dereference	no	no	no	yes
Missing Permission Checks	no	yes	no	yes
Race Conditions	no	no	no	no
Integer Overflows	no	no	no	yes
Division by Zero	no	yes	no	yes
Use After Free	no	no	yes	yes
Integer Type Issues	no	no	no	yes
Insecure Arguments	yes	yes	yes	yes

4. Constructing and simplifying CPG

This section proposes a simplified CPG approach to address the challenges of inadequate representation of the code's logical structure and the large size of graphical representations of source code. The impact of irrelevant code on vulnerability detection is mitigated by identifying code segments relevant to vulnerability detection and traversing these segments to obtain a CPG subgraph while preserving the code pertinent to the vulnerability.

4.1. Constructing CPG

CPG serves as a robust data structure for software security analysis, amalgamating various program analysis techniques to uncover vulnerability patterns in software. CPG comprises the following components: (1) AST: The AST presents a tree-based representation of source code, serving as the foundational element of CPG that reflects the code's syntactic structure. (2) Control Flow Graph (CFG): The CFG delineates all program execution paths and control flow transfers, facilitating the analysis of execution sequences and conditional evaluations within the program. (3) Data Flow Graph (DFG): The DFG illustrates the flow of data throughout the program, showing the processing of data and the relationships between variables. Table 1 [36] shows the types of vulnerabilities that different code representations can recognize. The table demonstrates that incorporating multiple types of edges better captures code information and identifies more vulnerability types than single-edge representations of code logic. Thus, CPG can represent the logical relationships of codes more adequately than other code representation methods.

An example of CPG generation for a code snippet is depicted in Fig. 2. In this study, the Joern (Fabian Yamaguchi et al., 2024) [37] tool is utilized to construct the CPG, which incorporates control flow and data flow into the AST, collectively representing the code's logical structural relationships. Nodes in the CPG denote code segments or tags, while edges denote iso-logical relationships. However, when dealing with complex or large software projects, using AST nodes in CPGs may result in an oversized graph and lead to redundant feature information. For example, in Fig. 2, the variable "num" appearing in rows 2 and 3 is represented by different nodes in the AST. To address this issue, we reconsider each line of code as a node, reducing excessive information duplication between nodes and decreasing the number of nodes.

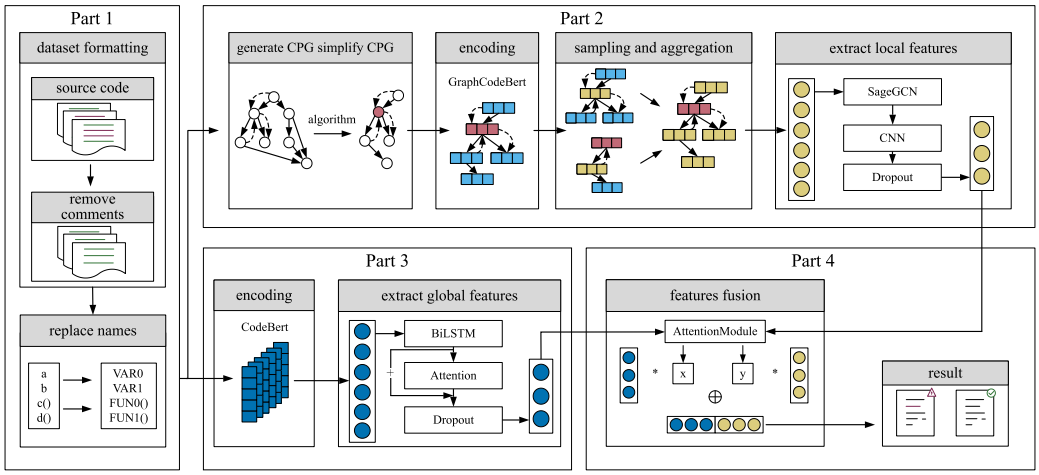


Fig. 1. Model architecture diagram.

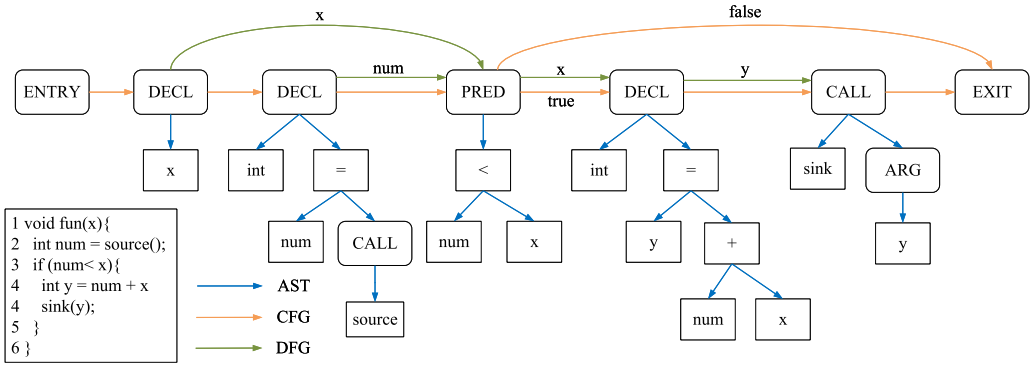


Fig. 2. CPG example.

Table 2
The logical structure of the relationships among different vulnerabilities and their respective levels of severity.

Vulnerability type	Related edge	Danger level
SQL Injection	DFG	high
Cross-Site Scripting	DFG	high
Buffer Overflow	DFG	high
Memory Disclosure	DFG	high
Missing Permission Checks	CFG	high
Race Conditions	CFG	high
Use After Free	DFG	high
Design Errors	AST	middle
Integer Overflows	DFG	middle
Resource Leaks	CFG	middle
Insecure Arguments	DFG	middle
Null Pointer Dereference	DFG	low
Integer Type Issues	DFG	middle
Division by Zero	DFG	middle

4.2. Simplifying CPG

When the source code contains a large number of snippets that are not related to vulnerabilities, the accuracy of the model's detection will be affected. Moreover, the CPG generated from the source code is often large, which leads to low training efficiency. For this reason, this study proposes a simplified CPG method aims at reducing the size of the CPG and filtering out non-essential code segments that do not contribute to vulnerability detection.

Each type of edge in the CPG is assigned a weight based on its importance in representing a logical relationship. The AST captures hierarchical code structures and detects issues such as mishandling of exception paths. The CFG identifies vulnerabilities such as logic errors and concurrency issues in multithreaded environments. The DFG traces variable scopes, lifecycles, and data flows and plays a crucial role in data dependency and taint analysis, outperforming the CFG in detecting vulnerabilities. Vertex Graph (CVG) edges are introduced to rank the lines of code in the CPG. Weights are assigned based on vulnerability types and dataset-specific risk levels (see Table 2): 40% for DFG, 30% for CFG, 20% for AST, and 10% for CVG. Additionally, if a node connects to multiple edges of the same type except the first one, the weight of the subsequent edges of the same type will be 0.1 times the original weight. To further assess the importance of CFG and DFG, this study counts the occurrences of CFG and DFG in the CPGs generated from the dataset we use, excluding the function header rows of internal function control flow. The DFG occurs about 1.7 times more often than the CFG, which prompted our decision to assign a higher weight to the DFG. By analyzing the connection of each node to the edges, the total weight of each node is calculated, and nodes with high weights are identified as being particularly useful for vulnerability detection.

Algorithm 1 details the process of simplifying the CPG. It takes the source code Sc as input and produces the simplified CPG G as output. Line 1 defines the weights $W = (w1, w2, w3, w4)$ for the four types of edges, the threshold P, the initial node weights $q = 0$, and the empty list D and GS. The threshold P is used to compare each node's weight q_i to identify "dangerous nodes", which potentially indicates vulnerabilities. Line 2 utilizes the Joern tool to initially generate $CPG = (v_i, e_j)$ and adjusts node information based on the CPG interpretation document. In lines 3–10, each node v_i is sequentially traversed, and the cumulative weight q_i of node v_i is computed using the edge weights w_j connected

Algorithm 1: Simplifying CPG process

Data: source code Sc
Result: simplified CPG G
1 define edge weights $W = (w_1, w_2, w_3, w_4)$, threshold P, node weight $q = 0$, the empty list D and GS;
2 using Joern generate $CPG = (v_i, e_j)$ and adjust node information;
3 **for** *traverse all* v_i *in the* CPG **do**
4 **for** *traverse the edges* e_j *connected by the nodes* v_i **do**
5 assign a weight $q_i = q_i + w_j$ to node v_i;
6 **end**
7 **if** *weight* q_i *is greater than threshold* P **then**
8 add node v_i to the danger list D
9 **end**
10 **end**
11 **for** *traverse all nodes* v_i *in the* D **do**
12 traverse v_i to generate slice result g_i;
13 add slice result g_i to the list GS;
14 **end**
15 **for** *traverse all slice result* g_i *in* GS **do**
16 merge subgraphs and remove duplicate nodes save the result to G;
17 **end**

Algorithm 2: Traversal code process

Data: dangerous node v_i
Result: slice result g_i
1 find the first-order neighbor node list $M_i = \{m_1, m_2 \ldots\ldots m_n\}$ of v_i;
2 **for** *traverse list* M_i **do**
3 traverse m_i using multi-type edges to obtain list l_i;
4 add the results from list l_i to list j_1;
5 remove duplicate nodes from list j_1;
6 **end**
7 traverse the list j_2 of CFG edges of v_i using single-type edges;
8 traverse the list j_3 of DFG edges of v_i using single-type edges;
9 merge three lists $J = j_1 \cap j_2 \cap j_3$;
10 remove nodes without AST from list J to obtain slice results g_i ;

to v_i. If q_i exceeds the threshold P, node v_i is added to the danger list D. Lines 11–14 iterate over nodes in the danger list D to obtain sliced subgraph g_i, where each subgraph corresponds to a potentially vulnerable region. To prevent redundant analysis of the same code, lines 15–17 merge all subgraphs g_i into the final simplified CPG G.

The traversal algorithm detailed in line 12 is described in Algorithm 2. The definition of multi-type edges allows traversal propagation over multiple types of edges, whereas single-type edges restrict traversal propagation to only one kind of edge at a time. The traversal process consists of the following steps: (1) Traverse each node v_i in the danger list D to collect its first-order neighbor nodes M_i. Using the multi-type edges principle, traverse nodes in M_i separately to obtain lists l_i, then merge all results into list j_1 while removing duplicates. (2) Traverse the danger list D along the CFG using single-type edges to obtain list j_2. (3) Traverse the danger list D along the DFG uses single-type edges to obtain list j_3. Steps (2) and (3) complement nodes potentially missed due to edge-type restrictions. (4) Combine lists j_1, j_2, and j_3 into a complete list J, removing nodes lacking ASTs, which contain only symbolic information and lack real code significance. All steps work together to generate the final CPG subgraph g_i and ensure that irrelevant information is removed.

The rationale behind traversing the first-order neighbor nodes of a node in the danger list D in step (1) is rooted in scenarios where a vulnerable node and the line of vulnerability resides in different functions. Directly traversing the node alone may not pinpoint the vulnerable line located in another function, necessitating the traversal of its first-order neighbor nodes. Listing a node in the danger list D does not necessarily indicate a vulnerability within the node's code but signifies its critical role in linking other lines of code throughout the program or being interconnected with multiple lines of code. Therefore, nodes listed in the danger list D in this study refer to lines of code occupying pivotal positions in the code's logical flow.

For example, the line of code that calls the 'validateAndCopy' function at line 12 in Fig. 3 is identified as a dangerous node. Traversing row 12 through the data flow and control flow of the graph, respectively, yields data flow results for nodes 10 and 11 and control flow results for node 4. Therefore traversing this node alone may not directly identify the potentially vulnerable code at line 2, 'strcpy(dest, source);'. Further traversal within the 'copyData' function, which is the first-order neighbor node of line 12, is necessary to locate this vulnerability. The result for first order nodes is 4, 10, 11 and further traversal through the first order nodes yields 1, 2, 5, 6. This approach ensures that traversal includes the vulnerable code at line 2. Many detection methods face challenges detecting vulnerabilities in real-world environments for similar reasons (Xiao Cheng et al., 2021) [38]. They typically use functions as the basic unit of vulnerability detection, overlooking vulnerabilities spanning multiple functions, which are prevalent in real-world scenarios (Adriana Sejfia et al., 2024) [39].

Fig. 4 shows an example of simplifying CPG using our approach. On the far left is a code fragment with a buffer overflow vulnerability at line 6. To its right is the initial CPG generated; the AST and node details are omitted for a clearer presentation of the CPG. Algorithm 1 determines that node 7 exceeds the weight threshold and is categorized as dangerous. Code fragments are extracted from traversing node 7's first-order neighbor nodes, resulting in (3, 5, 6, 8, 9), and from traversing node 7 itself, resulting in (3, 5, 6, 7, 9). These two results are merged to obtain a simplified CPG, as shown in column 3 of Fig. 4. The rightmost side displays the simplified code results, visually demonstrating the effectiveness of the simplification method proposed in this study.

5. CodeSAGE

To address the issue of incomplete feature extraction in neural network models, the vulnerability detection method, CodeSAGE, is proposed. By integrating features extracted from multiple neural network models, CodeSAGE enhances feature comprehensiveness and improves the accuracy of vulnerability detection.

5.1. Dataset preprocessing

A preprocessing step is performed before feature extraction with the model to standardize the source code. The specific preprocessing operations include: (1) Standardization of user-defined identifiers: User-defined variable names are uniformly mapped to predefined symbolic names, such as "VAR1", "VAR2", and so on. Similarly, user-defined function names are mapped to specified symbolic names, like "FUN1", "FUN2", etc. This step ensures a consistent naming convention across the dataset and prevents vocabulary explosion. (2) Remove annotations: All annotations, including single-line and multi-line comments, are removed from the source code. This operation reduces the amount of non-code text processed by the model, allowing it to focus on the code's structure and semantics. (3) Blank lines and spaces cleanup: Unnecessary blank lines and spaces are removed from the source code while preserving its logical structure.

Fig. 5 shows a comparison of a sample of source code before and after preprocessing. The original code on the left contains various inconsistencies, while the code on the right is clearer and more standardized after preprocessing. Through these preprocessing operations, this paper constructs a source code dataset with clear content and a consistent format, providing high-quality data input for the model.

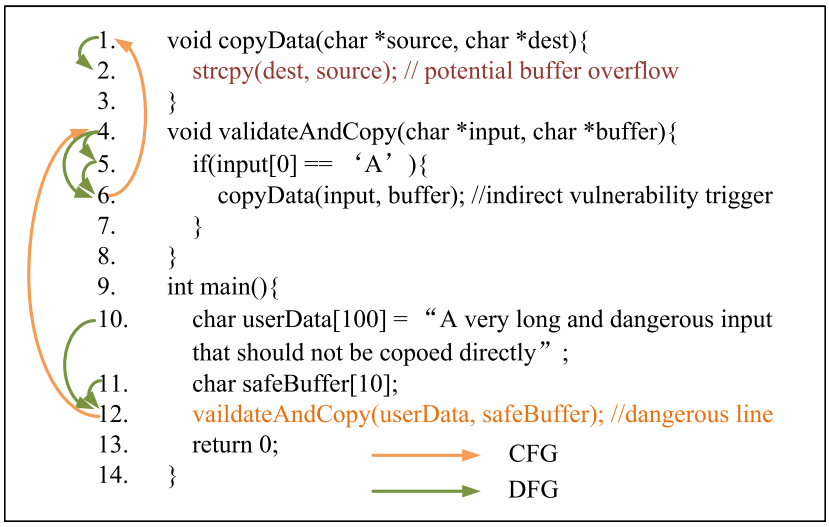


Fig. 3. Code example.

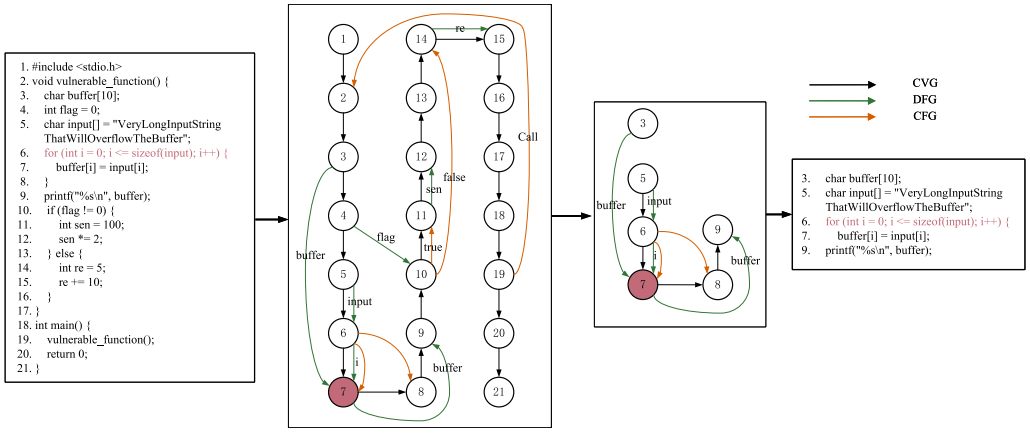


Fig. 4. Example of CPG simplified process.

source code	formatted source code

#include <string.h>
void processBuffer(const char *buffer, size_t length)
{
 	
printf("%s\n", buffer);
}
int main(int argc, char *argv[])
{
size_t len;
char src[4106];
char buf[10];
memset(src, 'A', 4106);
src[4106 - 1] = '\0';
len = 4106;
 	
/* BAD */
strncpy(buf, src, len);
processBuffer(src, len);
return 0;
}
	#include <string.h>
	void FUN1(const char *VAR1, size_t VAR2)
	{
	printf("%s\n",VAR1);
	}
	int main(int argc, char *argv[])
	{
	size_t VAR3;
	char VAR4[4106];
	char VAR5[10];
	memset(VAR4, 'A', 4106);
	VAR4[4106 - 1] = '\0';
	VAR3 = 4106;
	strncpy(VAR5, VAR4, VAR3);
	FUN1(VAR4, VAR3);
	return 0;
	}

Fig. 5. Code comparison chart.

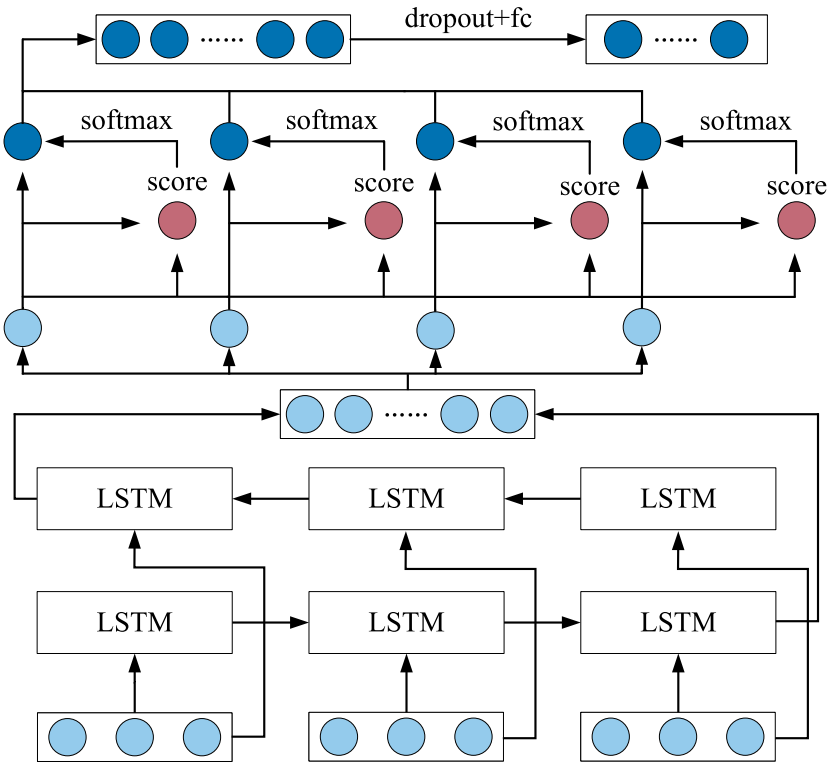


Fig. 6. Bi-LSTM framework.

5.2. Pre-trained model encoding

To meet the input requirements of the vulnerability detection models, this study utilizes the pre-trained models CodeBERT and GraphCodeBERT to encode node information in the source code and simplified CPG, respectively. CodeBERT and GraphCodeBERT are pure encoder models [40] trained on a large number of codebases and documents (Hamel Husain et al., 2020) [41], offering significant advantages in code vulnerability detection. They provide a solid foundation for feature extraction in subsequent neural network models and enhance the potential to generalize the models to other programming languages.

In this study, the maximum length of the code tokens is set to 512, and the feature length of each token is 768. If the token length is less than 512, zero padding is used; if the token length exceeds 512, the middle portion is used. Therefore, the vector input format for the vulnerability detection model is 512×768.

5.3. Bi-LSTM extracts global features

In this section, global features are extracted using Bi-LSTM in combination with an attention mechanism. Bi-LSTM can capture both forward and backward dependencies in a code sequence, making it very effective for source code analysis. However, it has limitations in capturing long-term dependencies, particularly in code logic analysis where a variable defined at the beginning of the code may not be used until hundreds of lines later. To address this, Bi-LSTM is combined with an attention mechanism. This combination allows the model to identify which parts of the sequence are critical, thus enhancing its ability to capture long-distance dependencies. The attention mechanism dynamically assigns weights to the sequence, focusing on important information and improving the model's ability to capture global features of the source code. Additionally, Bi-LSTM has fewer computational parameters and runs faster than a straightforward Transformer model.

Fig. 6 illustrates the Bi-LSTM framework. The encoded vectors are fed into the Bi-LSTM layer, which captures the contextual information of the source code. The output of the Bi-LSTM layer is then passed to the Attention layer, which computes a weighted average for each time step, focusing on the most important parts of the sequence. The output of the Attention layer is processed through the Dropout layer to prevent overfitting. Subsequently, the Fully Connected (FC) layer extracts higher-level features and normalizes the output format. The output of the Fully Connected layer serves as the global feature vector f_l for the extracted source code and as part of the input for feature fusion.

5.4. GraphSAGE extracts local features

In this section, the GraphSAGE model is used to extract local features of the CPG. It optimizes local feature extraction of graph structures through unique neighborhood sampling and diverse aggregation strategies. During the sampling process, We perform two operations on the nodes of the CPG: sampling the neighboring nodes of a node to obtain first-order neighboring nodes and sampling the first-order neighboring nodes to obtain second-order neighboring nodes. We set the selection to at most three neighboring nodes at a time. If there are more than three nodes, three nodes are selected randomly. Conversely, if the number of nodes is less than three, the random selection of one or more nodes is repeated until the desired number of nodes is reached. The aggregation process enriches the information of sample nodes by integrating data from neighboring nodes. The information on the second-order neighbor nodes are aggregated into the corresponding first-order neighbor nodes. The rich first-order neighbor information is further aggregated to the sampled nodes themselves. This order aggregation ensures that each node's data contains insights from its surrounding nodes, enhancing the local feature representation.

Since the pre-trained model enriches node information during the encoding process, the relatively computationally simple pooling aggregation method was chosen to mitigate the overfitting caused by the pre-trained model. For a target node v with a set of neighboring

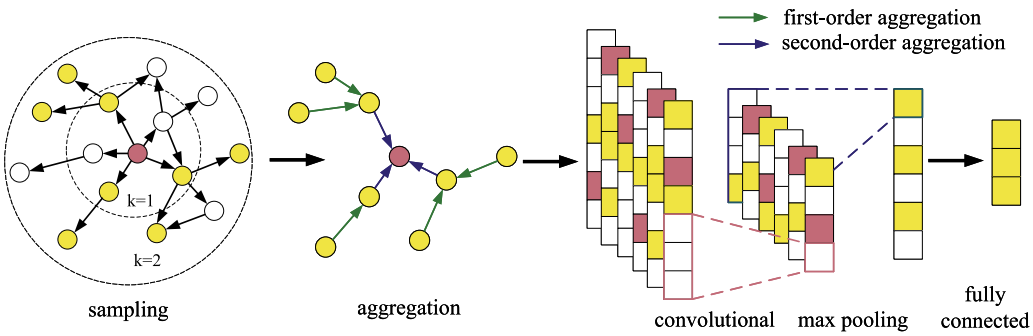


Fig. 7. GraphSAGE framework.

nodes $N(v)$, the pooling aggregation process is outlined as follows: each neighboring node's feature vector h_u undergoes a nonlinear transformation using a fully connected layer and a ReLU activation function, as represented by Formula (1):

$$h_{u'} = \mathrm{RELU}(W \cdot h_u + b) \tag{1}$$

W and b denote the weights and biases of the fully connected layer, respectively, with ReLU representing the activation function applied to derive $h_{u'}$, the transformed feature vector of each neighboring node u. Subsequently, a maximum pooling operation is applied to the nonlinearly transformed information of the neighbor nodes. The resulting pooled feature vector $h_{N(v)'}$ is expressed as Formula (2):

$$h_{N(v)'} = \max(h_{u'} | u \in N(v)) \tag{2}$$

The max function indicates that the maximum value is independently selected for each dimension of the feature vector of the neighbor nodes. The pooled feature vectors of the neighbors are then aggregated with the feature vector h_v of the target node through summation, resulting in the updated node information, as depicted in Formula (3):

$$H_v = \sigma(W \cdot (h_v + h_{N(v)'}) + b) \tag{3}$$

Fig. 7 illustrates how the GraphSAGE model extracts local features. After the sampling and aggregation operations, each node's feature vector incorporates information from its neighbors. A Convolutional Neural Network (CNN) is then employed to further refine these features. The output from the convolutional layer undergoes Dropout regularization to mitigate overfitting, followed by flattening. These flattened features are processed through a dynamically created fully connected layer to normalize vector sizes and apply batch normalization, enhancing model stability. Finally, the normalized output serves as the extracted local feature vector f_g, which along with f_l is used as input for the feature fusion stage.

5.5. Feature fusion

In automatic vulnerability detection, neural network models play a crucial role in feature extraction. These features are multidimensional, each representing different aspects with varying dimensions. Therefore, two independent fully connected layers are used to resize the extracted feature vectors f_l and f_g to a uniform dimension. However, even after resizing, differences in vector sizes remain, and simply adding or concatenating them can bias the model's decision-making process. Specifically, certain features might dominate while other important features are overshadowed.

To address this, an attention-based feature fusion approach is used. By introducing learnable weight parameters for different feature vectors, the model is adapted to reflect the contribution of each feature to the prediction results. This adaptation enables the model to handle diverse code effectively features and accurately detect potential vulnerabilities. The fusion process is illustrated in Formula (4), where w_g and w_l denote the weights for local and global features, respectively, and $f_{attention}$ represents the resulting fused feature vector.

$$f_{attention} = e^{w_g}/(e^{w_g} + e^{w_l}) \cdot f_g + e^{w_l}/(e^{w_g} + e^{w_l}) \cdot f_l \tag{4}$$

5.6. Loss improvement

An effective vulnerability detection method should not only identify potential vulnerabilities but also distinguish effectively between positive samples (secure code) and negative samples (vulnerable code), establishing clear boundaries within the potential code space. To address this challenge, this study employs a combination of Cross-Entropy Loss and Triplet Loss (Chengzhi Mao et al., 2019) [42]. Cross-Entropy Loss is widely used in classification tasks to quantify the disparity between the predicted probability distributions and the actually labeled distributions. It encourages the model to minimize this disparity by penalizing deviations from the correct predictions. The formula for Cross-Entropy Loss is expressed as follows (see Formula (5)):

$$L = -1/N \sum_{0<i<N} [y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)] \tag{5}$$

Consider N as the total number of samples in a dataset. For each sample i, y_i indicates its true label, and p_i represents the predicted probability assigned by the model for the sample belonging to the positive class.

Triplet Loss is commonly used in tasks such as image and face recognition to improve classification accuracy by learning the relative distances between different categories in the data. The Triplet Loss method is applied to vulnerability detection, with the aim of training the model to distinguish between securely effective code and code containing vulnerabilities in the potential space. The effectiveness of this approach depends on the efficient selection and representation of code segments, enabling the model to group secure code samples closely together and similarly group code samples containing vulnerabilities. The specific formula for the Triple Loss is represented by Formula (6):

$$L = \max(0, d(A, P) - d(A, N) + \mathrm{margin}) \tag{6}$$

Here, A represents the anchor point (the selected code snippet), P represents the positive example (a code snippet with the same label as the anchor point), N represents the negative example (a code snippet with a different label than the anchor point), and $d(x, y)$ denotes the distance metric. The hyperparameter margin defines the minimum distance that must be maintained between positive and negative examples.

This study opts not to include regularization loss as a third loss, as suggested by Saikat Chakraborty et al. (2021) [43], due to findings that

the characteristics of the data and the problem itself may render regularization loss less effective. In cases where the data lacks significant large-scale features or suffers from overfitting issues, regularization terms may hinder the model's ability to fit the data effectively, potentially reducing overall performance. Additionally, different types of datasets require different loss weight ratios, and improper parameter tuning can lead to underestimating or overestimating the impact of each component, which can hinder effective learning.

6. Experimental evaluation and analysis of results

In order to deeply evaluate the effectiveness of CodeSAGE in vulnerability detection, this section systematically validates its components and methods through experiments. A series of experiments is designed to explore the performance of CodeSAGE, focusing on its performance under different simplified conditions, comparison results with existing tools and models, and the specific contributions of each component in vulnerability detection. Next, the experimental analysis is developed around the following research questions.

6.1. Research questions

In this section, the following three research questions will be addressed experimentally.

Question 1: Does simplifying the CPG enhance the effectiveness of CodeSAGE in detecting vulnerabilities?

Question 2: Does CodeSAGE offer superior detection capabilities compared to existing tools or models?

Question 3: Are the various components of CodeSAGE effective in detecting vulnerabilities?

6.2. Experimental setup and metrics

In this study, experiments were conducted using a device equipped with a 13th Gen Intel(R) Core(TM) i9-13900K CPU running at 3.00 GHz, 64 GB of RAM, and an NVIDIA GeForce RTX 4090 GPU with 24 GB of video memory. A total of 30 training epochs were completed with a batch size of 64. The learning rate for the Bi-LSTM model was set to 0.001, while for the GraphSAGE model was set to 0.0001. The dataset was split randomly, with 80% used for training and the remaining 20% for testing.

In evaluating the performance of CodeSAGE, metrics relevant to each vulnerability categories are calculated individually and then averaged across all categories to determine the overall performance. For instance, the evaluation metrics for detecting vulnerabilities in category i are determined as follows.

True Positive (TP_i): The sample belongs to class i vulnerability, and the model correctly predicts it as class i vulnerability.

False Negative (FN_i): The sample belongs to class i vulnerability, but the model predicts a result that is not class i vulnerability.

False Positive (FP_i): The sample does not belong to class i vulnerability, but the model predicts it as a class i vulnerability.

True Negative (TN_i): The sample does not belong to class i vulnerability, and the model correctly predicts it as not belonging to class i vulnerability.

Precision (P): The proportion of samples predicted as truly vulnerable, as shown in Formula (7).

$$P = \sum_{0<i<N} [TP_i/(TP_i + FP_i)/n] \tag{7}$$

F-Measure ($F1$): A harmonized average of precision and recall, calculated as shown in Formula (8) and Formula (9).

$$TPR = \sum_{0<i<N} [TP_i/(TP_i + FN_i)/n] \tag{8}$$

$$F1 = \sum_{0<i<N} [2P_i \cdot TPR_i/(FP_i + TN_i)/n] \tag{9}$$

Accuracy (Acc): The ratio of correctly classified samples (both vulnerable and non-vulnerable) to the total number of samples, as shown in Formula (10).

$$Acc = \sum_{0<i<N} [(TP_i + TN_i)/(TP_i + TN_i + FN_i + FP_i)/n] \tag{10}$$

6.3. Dataset preparation

To accurately identify vulnerabilities in code, this study utilizes the NIST Software Assurance Reference Dataset Project (2018) [44] and a benchmark dataset from real-world environments, CodeXGLUE (Shuai Lu et al., 2021) [45]. The dataset for this paper can be found at [dataset] [46].

SARD is a comprehensive vulnerability dataset widely used in software security analysis, providing a diverse collection of software application source code samples with documented vulnerabilities. For this study, a dataset was constructed using 29,689 C/C++ source code files sourced from SARD. Of these, 11,240 files were identified as containing at least one vulnerability (labeled as 1), while 18,449 files were labeled as not containing vulnerabilities (labeled as 0).

The CodeXGLUE benchmark dataset is designed for function-level vulnerability detection and contains 27,318 functions manually labeled as either vulnerable or non-vulnerable. Vulnerable functions are labeled as 1, indicating the presence of potential vulnerabilities, while secure functions are labeled as 0.

A significant class imbalance was found in the SARD and CodeXGLUE datasets, where the number of non-vulnerable samples is considerably larger than the number of vulnerability samples. This imbalance can skew the feature extraction of the model towards the majority class (non-vulnerable samples) and affect the accuracy of vulnerability detection. To address this issue, the Synthetic Minority Oversampling Technique (SMOTE) (Nitesh V Chawla et al., 2002) [47] is employed. The class proportions are adjusted by oversampling the minority class (vulnerability-containing codes) and undersampling the majority class (security codes). SMOTE selects a vulnerability-containing the sample and identifies its k neighboring vulnerability samples. A new sample is generated by interpolating between the original sample and its neighbors. Simultaneously, the number of non-vulnerable samples is randomly reduced. This process is repeated until the dataset achieves a balanced number of vulnerability-containing and security samples.

6.4. Experimental process

During the course of the experiments, the following three experiments were designed to fully validate the effectiveness of CodeSAGE in vulnerability detection. These experiments aim to evaluate the impact of different factors on the performance of CodeSAGE and demonstrate its advantages over existing models and tools through comparative analyses.

6.4.1. Validation of CPG simplification

To validate the efficacy of CPG simplification in vulnerability detection using CodeSAGE, this study examines two aspects: scaling down the graph size to improve computational efficiency and filtering out irrelevant code to enhance detection accuracy.

The number of CPG nodes screened is controlled by setting different CPG node thresholds Pi (30, 40, 50, 60, 70, 80). Simplified CPG were then generated at each threshold condition and compared to the unsimplified CPG, recording the size of the simplification. The CodeSAGE model was then trained and the time for each round of training was recorded. Finally, the detection accuracies of the models with different thresholds on the validation set are observed to measure the impact of CPG simplification on model performance.

6.4.2. CodeSAGE comparison experiments with existing tools and models

To demonstrate CodeSAGE's superior capability in vulnerability detection, this study compares it with several state-of-the-art models and tools using the SARD and CodeXGLUE datasets.

Firstly, the open-source static analysis tool Flawfinder is evaluated as a benchmark. It is widely used to detect vulnerabilities in C/C++ source code. Secondly, two deep learning-based methods, VulDeePecker and SySeVR (Zhen Li et al., 2021) [48], are considered. Both methods, like CodeSAGE, employ code-slicing techniques to focus on essential code segments, with SySeVR represents an advancement over VulDeePecker. Lastly, ReGVD and Devign are included, which utilize graph representations to capture the internal structure of code and employ graph neural networks for vulnerability detection. This selection ensures a comprehensive evaluation covering various detection methods and techniques, providing a robust comparison against the proposed methods in this study. The remaining four methods were reproduced using the dataset prepared in this paper for comparison with CodeSAGE, except for the Devign method. The source code for Devign is not provided in the Devign paper and, therefore cannot be reproduced. However, since the CodeXGLUE dataset references the dataset used by ReGVD, the results of the Devign experiments from ReGVD are used for comparison.

To ensure the fairness of the experiments and the comparability of the results, each model is run on the SARD and CodeXGLUE datasets separately, and a series of key metrics, including Acc, $F1$, and P, are recorded as a basis for comparison.

6.4.3. Analysis of the contribution of CodeSAGE components to vulnerability detection

This study explored the contribution of CodeSAGE components to assay validity in three ways: the impact of simplified CPG on vulnerability detection the effect of multi-angle feature extraction and fused patterns on vulnerability detection, and the impact of combined loss methods on vulnerability detection. The code structure and logical complexity of the SARD dataset are relatively simple, and the differences in the results of the component comparison experiments are not sufficient to highlight the contribution of each component to the detection model. Therefore, the CodeXGLUE dataset will be utilized to further explore these aspects.

The detailed process is to split the model step by step, test Acc, $F1$ and P of different component combinations as a basis for comparison, and visualize and analyze by t-SNE (Laurens Van der Maaten and Geoffrey Hinton, 2008) [49] to verify the effect of the combination loss method on the classification effect.

6.5. Experimental results and analysis

In this section, the experimental results demonstrate the specific effects of each experiment and the advantages of the CodeSAGE model. In the following section, several key experimental results are analyzed in detail and the results are used to answer the three questions posed in Section 6.1.

6.5.1. CPG simplified experimental results

Fig. 8(a) illustrates the degree of simplification of CPG size for different threshold settings. Where the x-axis represents the effect of CPG size simplification, the y-axis represents the size of the dataset, and the different colored dotted lines represent different thresholds. From the results depicted in Fig. 8(a), it can be seen that the size of the simplified code CPG has generally been significantly reduced by approximately between 25% and 45%. This result indicates that the originally complex code graph was effectively streamlined by adjusting the node thresholds and simplifying the graph structure. Additionally, this study evaluated the average training time for 30 rounds under different threshold Pi conditions. As shown in the figure, CPG simplification reduces the average training time per round by 20% compared to using the full CPG.

Based on the aforementioned approach, the accuracy of vulnerability detection using simplified CPG with different threshold settings was evaluated. Each threshold value was applied consistently across the same model architecture in 30 rounds of training experiments. The results of these experiments are depicted in Fig. 8(c). The accuracy of vulnerability detection stabilizes around 15 rounds of training for different threshold sizes. The accuracy rates do not vary significantly once stabilized, with the highest accuracy reaching 73.57% observed when the threshold is set to 30. The results of the detection without CPG simplification are also depicted in Fig. 8(c), showing a 1.69% improvement in accuracy when the threshold is set to 30 compared to using no CPG simplification.

In summary, through the above experiments, it can be concluded from Question 1 that simplifying the code CPG improves the efficiency of model training and the accuracy of vulnerability detection. However, further refinement and validation are needed to understand the relationship better between the impact of simplification and detection accuracy.

Table 3
Test results on the SARD dataset.

Method	ACC(%)	$F1$(%)	P(%)
Flawfinder (2018)	56.27	52.86	59.49
VulDeePecker (2018)	88.51	69.06	84.95
SySeVR (2021)	97.32	91.21	90.10
ReGVD (2023)	98.72	93.50	92.60
CodeSAGE	99.12	97.12	95.21

Table 4
Test results on the CodeXGLUE dataset.

Method	ACC(%)	$F1$(%)	P(%)
Flawfinder (2018)	45.23	52.78	33.16
VulDeePecker (2018)	51.94	47.19	51.00
SySeVR (2021)	52.52	56.03	48.34
Devign (2019)	59.77		
ReGVD (2023)	66.17	61.52	60.87
CodeSAGE	73.57	70.13	71.66

6.5.2. Results of CodeSAGE compared to other models

The accuracy results of CodeSAGE and the other tools/models after 30 rounds of training on the SARD dataset are depicted in Fig. 8(d) and the best vulnerability detection results for each method are summarized in Table 3. CodeSAGE outperforms all other methods across all metrics, achieving an accuracy of 99.12%. This superiority can be attributed to several factors: Flawfinder's superficial code analysis and lack of deep code logic evaluation lead to significant accuracy gaps compared to other methods; VulDeePecker focuses only on vulnerabilities related to library/API function calls, limiting its effectiveness in detecting other types of vulnerabilities; SySeVR relies on code sequence design and may not effectively capture code logic features and vulnerability semantic information; and ReGVD's approach may overlook distant logical relationships. In contrast, CodeSAGE achieves higher accuracy on the SARD dataset.

Since the SARD dataset mainly consists of synthesized code, it tends to be simpler compared to real-world code environments and does not fully reflect real-world vulnerability detection scenarios. For this reason, CodeSAGE was evaluated on the CodeXGLUE dataset, which represents a more realistic software development environment. Fig. 8(e) shows the accuracy results after 30 rounds of training on the CodeXGLUE dataset, while Table 4 presents the results of the best comparative experiments of our method for detecting vulnerabilities in real-world projects, with an accuracy of 73.57%, an F1 score of 70.13%, and a precision of 71.66%.

Taken together, it can be concluded that CodeSAGE demonstrates superior detection capabilities compared to existing tools and models.

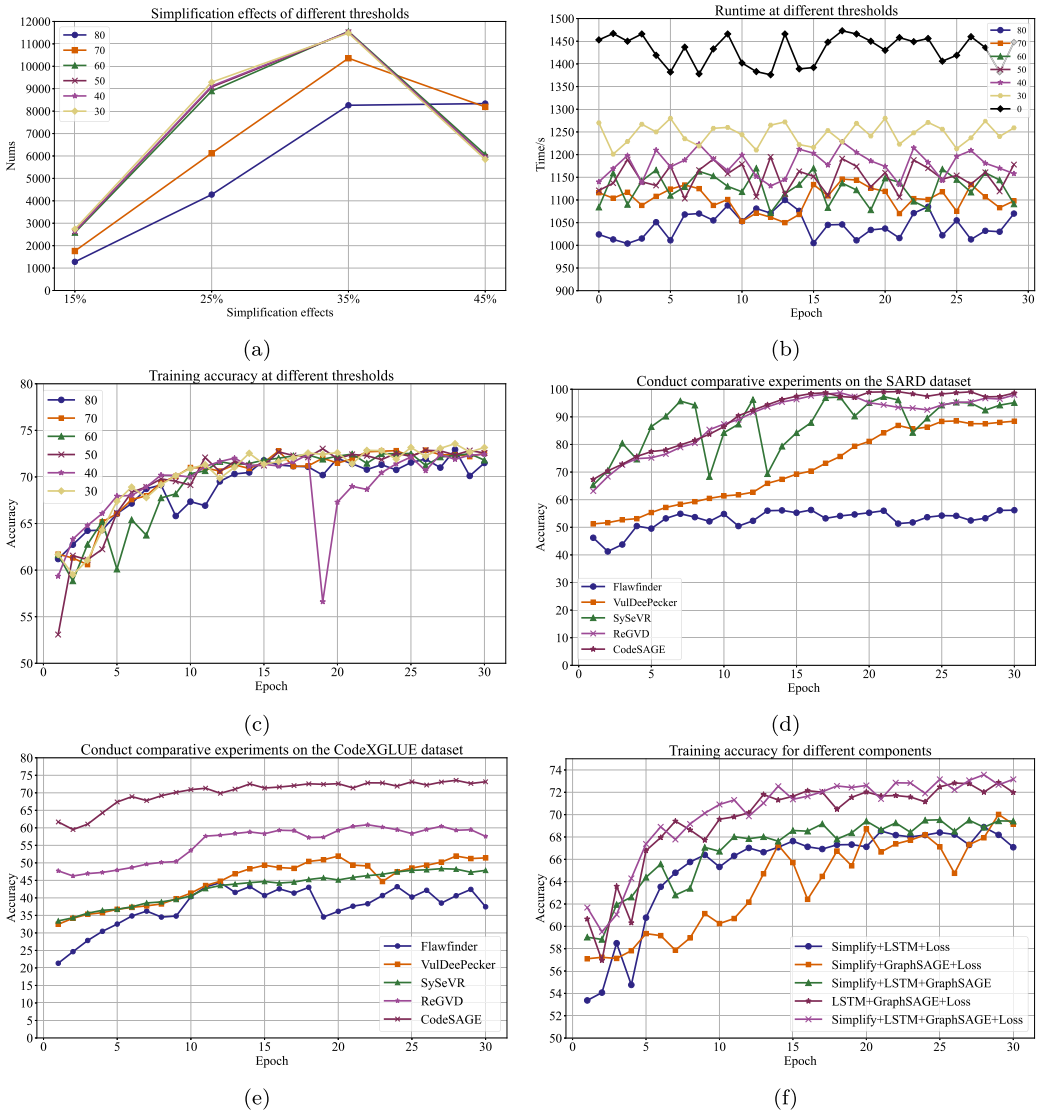


Fig. 8. Experimental results.

Table 5
Comparative results of CodeSAGE components.

Method	*ACC*(%)	*F*1(%)	*P*(%)
Simplify+LSTM+Loss	68.88	68.73	68.74
Simplify+GraphSAGE+Loss	70.02	64.72	68.03
Simplify+LSTM+GraphSAGE	69.54	69.2	69.34
LSTM+GraphSAGE+Loss	71.88	70.6	69.71
Simplify+LSTM+GraphSAGE+Loss	73.57	70.13	71.66

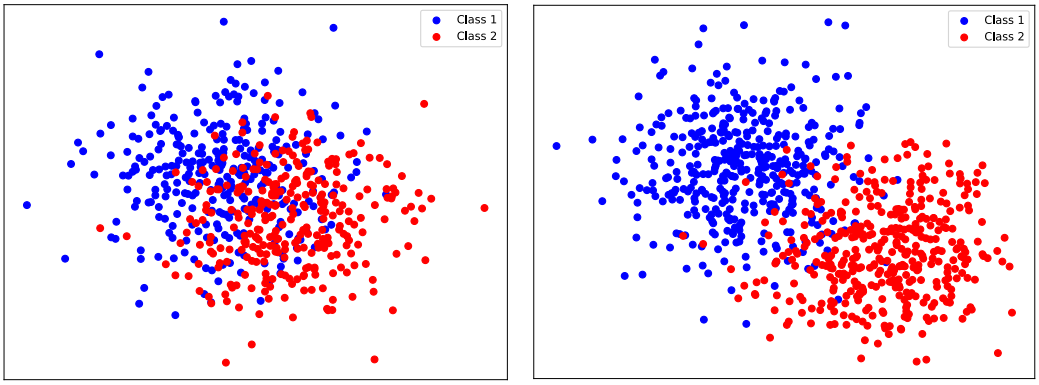
It achieved a detection accuracy of 99.12% on the SARD dataset and 73.57% on the CodeXGLUE dataset, outperforming the other comparison methods chosen in this study.

6.5.3. Experimental results of component contribution analysis

To facilitate result presentation, the components are briefly defined as follows: 'Simplify' refers to code CPG simplification, 'LSTM' denotes the Bi-LSTM model, 'GraphSAGE' represents the GraphSAGE model, and 'Loss' indicates the combined loss methods. Comparative test results between these components are documented in Table 5, with changes in accuracy per round illustrated in Fig. 8(f). The results show that code CPG simplification improves vulnerability detection accuracy by 1.69%, attributed to the removal of non-relevant code. Employing combined loss methods further enhances class separation, resulting in a 4.03% accuracy improvement. Among single models, GraphSAGE outperforms Bi-LSTM, highlighting the effectiveness of using graphs to represent code logical relationships over intermediate languages or direct source code adoption. Combining multiple models provides more comprehensive feature extraction, increasing accuracy by 4.69% compared to Bi-LSTM and 3.55% compared to GraphSAGE.

To further investigate the validity of the integrated loss, t-SNE mapping is employed to visualize the model classification results. t-SNE is a dimensionality reduction technique suitable for visualizing high-dimensional dataset distributions in the feature space. Fig. 9 compares t-SNE mappings before and after applying the combined loss feedback. Before applying the combined loss feedback, there is a significant overlap between the two categories in the feature space. However, after applying the feedback, distinct separation between the categories is observed, indicating an improved model ability to discern differences.

Therefore, through the above experiments, it can be concluded from Question 3 that each of the components of CodeSAGE (CPG simplification, multi-feature fusion, and combining loss functions) contributes to



(a) Before (b) After

Fig. 9. t-SNE comparison chart.

improved vulnerability detection.

7. Conclusions

In this paper, CodeSAGE is presented as an innovative approach for identifying source code vulnerabilities in the software supply chain using deep learning techniques. CodeSAGE converts source code into CPG and leverages GraphSAGE for local feature extraction while using Bi-LSTM to capture global sequence features. The comprehensiveness of the features is enhanced by fusing local graph features from GraphSAGE and global sequence features from Bi-LSTM. To address the scalability issue of CPG, a simplified CPG method is proposed to reduce graph size and exclude code that interferes with detection results. Additionally, a combined loss function strategy is employed to optimize class boundary delineation in the feature space and strengthen model classification capability. The experimental results demonstrate that CodeSAGE achieves high efficiency and accuracy in vulnerability detection, outperforming existing methods.

In future research, we intend to incorporate additional pre-trained models and advanced neural network architectures to achieve more robust coding and feature fusion. Simultaneously, the performance of CPG will be enhanced to ensure efficient representation across multiple files. Moreover, for the identified vulnerable code fragments, large models are planned to be fine-tuned to automatically generate vulnerability patches, further bolstering the security of the code supply chain.

CRediT authorship contribution statement

Guodong Zhang: Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization. **Tianyu Yao:** Writing – review & editing, Writing – original draft, Visualization, Software, Resources, Formal analysis, Data curation. **Jiawei Qin:** Writing – review & editing, Visualization, Validation, Resources, Formal analysis, Data curation. **Yitao Li:** Writing – review & editing, Visualization, Validation, Resources, Formal analysis, Data curation. **Qiao Ma:** Writing – review & editing, Visualization, Validation, Resources, Formal analysis, Data curation. **Donghong Sun:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by National Key Research and Development Program of China (No. 2022YFB3103901) and Zhongguancun Laboratory (No. ZGC-02-20220211).

Data availability

Data will be made available on request.

References

[1] International Telecommunication Union. Global Internet Penetration and Digital Infrastructure Development Report. Technical Report, International Telecommunication Union; 2023, [Accessed: 23 December 2023].
[2] Netflix. Continuous update. 2023, https://www.netflix.com. [Accessed: 16 October 2023].
[3] YouTube. Continuous update. 2023, https://www.youtube.com. [Accessed: 16 October 2023].
[4] Lakshmanan R. Wazirx cryptocurrency exchange loses $230 million in major security breach. 2024, https://thehackernews.com/2024/07/wazirx-cryptocurrency-exchange-loses.html. [Accessed: 19 July 2024].
[5] Ladisa P, Plate H, Martinez M, Barais O. SoK: Taxonomy of attacks on open-source software supply chains. In: 2023 IEEE symposium on security and privacy. 2023, p. 1509–26. http://dx.doi.org/10.1109/SP46215.2023.10179304.
[6] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput 1997;9(8):1735–80. http://dx.doi.org/10.1162/neco.1997.9.8.1735.
[7] Hamilton W, Ying Z, Leskovec J. Inductive representation learning on large graphs. Adv Neural Inf Process Syst 2017;30.
[8] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. Adv Neural Inf Process Syst 2017;30.
[9] Senanayake J, Kalutarage H, Petrovski A, Piras L, Al-Kadri MO. Defendroid: Real-time android code vulnerability detection via blockchain federated neural network with XAI. J Inf Secur Appl 2024;82:103741. http://dx.doi.org/10.1016/j.jisa.2024.103741, URL https://www.sciencedirect.com/science/article/pii/S2214212624000449.
[10] Eshghie M, Artho C, Gurov D. Dynamic vulnerability detection on smart contracts using machine learning. In: Proceedings of the 25th international conference on evaluation and assessment in software engineering. 2021, p. 305–12. http://dx.doi.org/10.1145/3463274.3463348.
[11] LI J, YU Y, WU J. Vulnerability detection algorithm of DOM XSS based on dynamic taint analysis. J Comput Appl 2016;36(5):1246.
[12] Kim S, Lee H. Software systems at risk: An empirical study of cloned vulnerabilities in practice. Comput Secur 2018;77:720–36.
[13] Cui L, Hao Z, Jiao Y, Fei H, Yun X. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. IEEE Trans Inf Forensics Secur 2020;16:2004–17. http://dx.doi.org/10.1109/TIFS.2020.3047756.
[14] Sun H, Cui L, Li L, Ding Z, Hao Z, Cui J, et al. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. Comput Secur 2021;110:102417. http://dx.doi.org/10.1016/j.cose.2021.102417.
[15] Checkmarx. Continuous update. 2023, https://www.checkmarx.com/. [Accessed: 27 July 2023].
[16] Perforce Software. Continuous update, klocwork: Static code analysis tool. 2023, https://www.perforce.com/products/klocwork. [Accessed: 27 July 2023].
[17] SonarSource. Continuous update, SonarQube: Continuous code quality and security. 2023, https://www.sonarqube.org. [Accessed: 27 July 2023].
[18] Wheeler D. FlawFinder. 2018, http://www.dwheeler.com/flawfinder. [Accessed: 23 April 2023].
[19] Melicher W, Fung C, Bauer L, Jia L. Towards a lightweight, hybrid approach for detecting DOM xss vulnerabilities with machine learning. In: Proceedings of the web conference 2021. New York, NY, USA: Association for Computing Machinery; 2021, p. 2684–95. http://dx.doi.org/10.1145/3442381.3450062.

[20] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, et al. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications. IEEE; 2018, p. 757–62. http://dx.doi.org/10.1109/ICMLA.2018.00120.

[21] Li Z, Zou D, Xu S, Ou X, Jin Y, Wang Y, et al. VulDeepecker: A deep learning-based system for vulnerability detection. In: Proceedings of the 25th annual network and distributed system security symposium. 2018, p. 1–15. http://dx.doi.org/10.14722/ndss.2018.23158.

[22] Nguyen V-A, Nguyen DQ, Nguyen V, Le T, Tran QH, Phung D. Regvd: Revisiting graph neural networks for vulnerability detection. In: Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings. 2022, p. 178–82. http://dx.doi.org/10.1145/3510454.3516865.

[23] George TK, Jacob KP, James RK. Token based detection and neural network based reconstruction framework against code injection vulnerabilities. J Inf Secur Appl 2018;41:75–91. http://dx.doi.org/10.1016/j.jisa.2018.05.005, URL https://www.sciencedirect.com/science/article/pii/S2214212617300480.

[24] Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv Neural Inf Process Syst 2019;32.

[25] Nguyen HV, Zheng J, Inomata A, Uehara T. Code aggregate graph: Effective representation for graph neural networks to detect vulnerable code. IEEE Access 2022;10:123786–800. http://dx.doi.org/10.1109/ACCESS.2022.3216395.

[26] Bowman B, Huang HH. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In: 2020 IEEE European symposium on security and privacy (euroS&p). IEEE; 2020, p. 53–69. http://dx.doi.org/10.1109/EuroSP48549.2020.00012.

[27] Wu Y, Zou D, Dou S, Yang W, Xu D, Jin H. Vulcnn: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th international conference on software engineering. 2022, p. 2365–76. http://dx.doi.org/10.1145/3510003.3510229.

[28] Luo F, Luo R, Chen T, Qiao A, He Z, Song S, et al. Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In: Proceedings of the IEEE/ACM 46th international conference on software engineering. 2024, p. 1–13.

[29] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE symposium on security and privacy. IEEE; 2014, p. 590–604. http://dx.doi.org/10.1109/SP.2014.44.

[30] Ling M, Zhang Y. VulScan: A vulnerability detection model based on deep learning. In: 2023 international conference on blockchain technology and information security. 2023, p. 88–93. http://dx.doi.org/10.1109/ICBCTIS59921.2023.00021.

[31] Wang W, Nguyen TN, Wang S, Li Y, Zhang J, Yadavally A. DeepVD: Toward class-separation features for neural network vulnerability detection. In: 2023 IEEE/ACM 45th international conference on software engineering. 2023, p. 2249–61. http://dx.doi.org/10.1109/ICSE48619.2023.00189.

[32] Tang M, Tang W, Gui Q, Hu J, Zhao M. A vulnerability detection algorithm based on residual graph attention networks for source code imbalance (RGAN). Expert Syst Appl 2024;238:122216.

[33] Liu Z, Fang Y, Huang C, Xu Y. MFXSS: An effective XSS vulnerability detection method in JavaScript based on multi-feature model. Comput Secur 2023;124:103015.

[34] Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, et al. GraphCodeBERT: Pre-training code representations with data flow. In: International conference on learning representations. 2021, http://dx.doi.org/10.48550/arXiv.2009.08366.

[35] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the association for computational linguistics: EMNLP 2020. 2020, p. 1536–47. http://dx.doi.org/10.48550/arXiv.2002.08155.

[36] Schuckert F. Opportunities of insecurity refactoring for training and software development. 2024, p. 28, https://hdl.handle.net/11250/3127615.

[37] Yamaguchi F, Pollmeier M, Steward S, et al. Joern. 2024, https://github.com/joernio/joern.

[38] Cheng X, Wang H, Hua J, Xu G, Sui Y. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans Softw Eng Methodol (TOSEM) 2021;30(3):1–33. http://dx.doi.org/10.1145/3436877.

[39] Sejfia A, Das S, Shafiq S, Medvidović N. Toward improved deep learning-based vulnerability detection. In: Proceedings of the 46th IEEE/ACM international conference on software engineering. 2024, p. 1–12. http://dx.doi.org/10.1145/3597503.3608141.

[40] Zhou X, Cao S, Sun X, Lo D. Large language model for vulnerability detection and repair: Literature review and roadmap. 2024, http://dx.doi.org/10.48550/arXiv.2404.02525, arXiv preprint arXiv:2404.02525.

[41] Husain H, Wu H-H, Gazit T, Allamanis M, Brockschmidt M. CodeSearchNet challenge: Evaluating the state of semantic code search. 2020, https://arxiv.org/abs/1909.09436, arXiv:1909.09436.

[42] Mao C, Zhong Z, Yang J, Vondrick C, Ray B. Metric learning for adversarial robustness. Adv Neural Inf Process Syst 2019;32.

[43] Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: Are we there yet? IEEE Trans Softw Eng 2021;48(9):3280–96. http://dx.doi.org/10.1109/TSE.2021.3087402.

[44] Software assurance reference dataset. 2018, https://samate.nist.gov/SRD/index.php. [Accessed: 23 April 2023].

[45] Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. 2021, http://dx.doi.org/10.48550/arXiv.2102.04664, arXiv preprint arXiv:2102.04664.

[46] Tianyu Y. Codesage dataset. 2024, https://github.com/fgdyx/CodeSAGE/tree/master.

[47] Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. SMOTE: synthetic minority over-sampling technique. J Artificial Intelligence Res 2002;16:321–57. http://dx.doi.org/10.1613/jair.953.

[48] Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans Depend Secur Comput 2021;19(4):2244–58. http://dx.doi.org/10.1109/TDSC.2021.3051525.

[49] Van der Maaten L, Hinton G. Visualizing data using t-SNE. J Mach Learn Res 2008;9(11).