The Journal of Systems & Software 202 (2023) 111706

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

LCVD: Loop-oriented code vulnerability detection via graph neural network☆

Mingke Wang [a], Chuanqi Tao [a,b,c,d,*], Hongjing Guo [a]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] Ministry Key Laboratory for Safety-Critical Software Development and Verification, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[c] Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing, China
[d] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 1 January 2023
Received in revised form 17 March 2023
Accepted 11 April 2023
Available online 19 April 2023

Keywords:
Loop-oriented vulnerability
Vulnerability detection
Deep learning
Code representation
Graph neural network

ABSTRACT

Due to the unique mechanism and complex structure, loops in programs can easily lead to various vulnerabilities such as dead loops, memory leaks, resource depletion, etc. Traditional approaches to loop-oriented program analysis (e.g. loop summarization) are costly with a high rate of false positives in complex software systems. To address the issues above, recent works have applied deep learning (DL) techniques to vulnerability detection. However, existing DL-based approaches mainly focused on the general characteristics of most vulnerabilities without considering the semantic information of specific vulnerabilities. As a typical structure in programs, loops are highly iterative with multi-paths. Currently, there is a lack of available approaches to represent loops, as well as useful methods to extract the implicit vulnerability patterns. Therefore, this paper introduces LCVD, an automated loop-oriented code vulnerability detection approach. LCVD represents the source code as the Loop-flow Abstract Syntax Tree (LFAST), which focuses on interleaving multi-paths around loop structures. Then a novel Loop-flow Graph Neural Network (LFGNN) is proposed to learn both the local and overall structure of loop-oriented vulnerabilities. The experimental results demonstrate that LCVD outperforms the three static analysis-based and four state-of-the-art DL-based vulnerability detection approaches across evaluation settings.

1. Introduction

With the increase in size and complexity of software, vulnerabilities have become major threats to software security. A loop is a program structure set up when a function needs to be executed repeatedly in a program, usually consisting of a loop control condition and a loop body that needs to be executed repeatedly. As the base code of software systems, loops are prone to various security vulnerabilities due to their unique loop mechanism and complex program structure.

Static program analysis is an efficient technique to ensure software security, while looping is the key challenge of program verification (Kroening et al., 2013). Loop unwinding is an easy approach to analyze loop code if the number of loop iterations is predictable. However, it is too expensive for large programs and many practical applications may have vulnerabilities of *infinite loop* already (Biere et al., 2009). Since in many cases it is not possible to calculate bounds on the number of loop iterations in advance, some researchers convert programs into logical formulas through mathematical analysis and then solve for the logical formulas, such as calculating the approximation of the loop (i.e.loop invariants) (Cousot and Cousot, 1977; Sharma et al., 2011) and loop summarization (Kroening et al., 2013; Godefroid and Luchaup, 2011). However, calculating loop invariants for complex programs requires many iterations, which is computationally expensive and introduces further inaccuracies (Kroening et al., 2013). Although loop summarization can calculate the outputs without executing the loop, it has great difficulty in handling multi-path loops (Godefroid and Luchaup, 2011). Therefore, analyzing vulnerabilities that are closely related to loop structures through traditional static program analysis is unreliable and difficult to apply to complex software systems. We refer to these types of vulnerabilities as "loop-oriented".

To address the above issues, there has been a large amount of recent work applying deep learning (DL) to code vulnerability detection (VD) (Li et al., 2018b; Russell et al., 2018; Zhou et al., 2019; Cao et al., 2021; Zou et al., 2021; Li et al., 2022). Nevertheless, one challenge of current DL-based VD approaches for loop-oriented vulnerabilities is **how to effectively learn the**

☆ Editor: Aldeida Aleti.

* Corresponding author at: College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.
E-mail address: taochuanqi@nuaa.edu.cn (C. Tao).

https://doi.org/10.1016/j.jss.2023.111706
0164-1212/

```
1   static int svg_probe(AVProbeData *p)
2   {
3       const uint8_t *b = p->buf;
4       const uint8_t *end = p->buf + p->buf_size;
5       if (memcmp(p->buf, "<?xml", 5))
6           return 0;
7       while (b < end) {
8   -       b += ff_subtitles_next_line(b);
9   +       int inc = ff_subtitles_next_line(b);
10  +       if (!inc)
11  +           break;
12  +       b += inc;
13          if (b >= end - 4)
14              return 0;
15          if (!memcmp(b, "<svg", 4))
16              return AVPROBE_SCORE_EXTENSION + 1;
17      }
18      return 0;
19  }
```

Fig. 1. An *Infinite Loop* vulnerability (CVE-2018-7751) in *FFmpeg*.

syntactic and semantic information in loop-oriented vulnerabilities. Existing VD approaches mainly focus on generic vulnerability patterns rather than specific vulnerabilities. However, it is unreasonable to apply generic VD approaches to different types of vulnerabilities, as the original cause of vulnerability formation is completely different. For example, Cao et al. (2022) focused on memory-related vulnerabilities and proposed specific flow-sensitive graph neural networks (FS-GNN) to detect memory-related vulnerabilities, which achieved better detection accuracy. Most of the existing DL-based VD approaches represent the source code as a flat sequence (Li et al., 2018b, 2022; Zou et al., 2021) or graph structure (Yamaguchi et al., 2014; Zhou et al., 2019; Cao et al., 2021). Loops are a typical structure of programs whose complexity increases along with the size of the software. As a special type of branch, loops have a variety of multi-paths and are highly iterative (Xiao et al., 2013). To take full advantage of the above characteristics of loops, we propose Loop-oriented Code Vulnerability Detection (*LCVD*), which consists of two new techniques:

A proposed graph representation that focuses on the structure of loop code. To represent the interleaving relationships between multi-paths in loops, the source code is modeled as a composite graph with an Abstract Syntax Tree (AST) as the underlying backbone, and control dependencies and data dependencies are added as heterogeneous edges. Furthermore, to highlight features of the loop structure, we simplified the graph according to a predefined graph cropping rule. Due to the lack of programming logic from the source code sequences in graph structures, we take the combination of type, text, and position information as the initializing feature of nodes in graphs. In this way, the structural information of code graphs and non-structural information of code sequences are united, resulting in a semantically comprehensive abstract graph representation for loop code.

A proposed specific graph neural network (GNN) that captures features of loop-oriented vulnerabilities. In the phase of vulnerability detection, we regard vulnerability detection as a graph-level classification task. Hence a novel Loop-flow Graph Neural Network (LFGNN) is constructed, which focuses on the loop structure by self-attention mechanism and captures more comprehensive semantics by a multi-layer graph neural network. Besides, Residual connections are added to mitigate the information loss and gradient disappearance caused by deep networks. A well-designed hybrid graph-pooling module then generates graph-level embedding by maximizing the information of the vulnerable nodes. Finally, the graph-level embedding is fed into the classification module to predict whether there have vulnerabilities.

As there is no available dataset of loop-oriented vulnerabilities, We manually constructed a dataset from 11 open-source projects of the C programming language, which contains 4488 loop-oriented function-level code blocks.

The contributions of this paper can be summarized as follows:

- We propose a graph representation applicable to loops named Loop-flow Abstract Syntax Tree (LFAST), which represents interleaving relationships of multi-paths in the loop structures, including both structural and unstructured information in the source code.
- We propose a novel Loop-Flow Graph Neural Network (LFGNN) by adding the self-attention mechanism and designing a specialized graph-pooling module to the original GNN framework, which comprehensively captures the implicit features in loop-oriented vulnerabilities.
- We evaluated *LCVD* on the constructed loop-oriented vulnerability dataset by comparing it with three static analysis-based and four state-of-the-art DL-based VD approaches. *LCVD* achieves the best performance with an improvement ranging from 36.36%–50.74% compared to the best-performing static analysis-based baseline and 7.13%–8.48% compared to the best-performing DL-based baseline on primary metrics.

2. Motivating example

Fig. 1 shows an *Infinite Loop* vulnerability (CWE-835) extracted from a real-world project *FFmpeg*, which is listed as a vulnerable code within Common Vulnerabilities and Exposures (CVE-2018-7751) in the National Vulnerability Database. The vulnerable function *svg_probe* allows remote attackers to

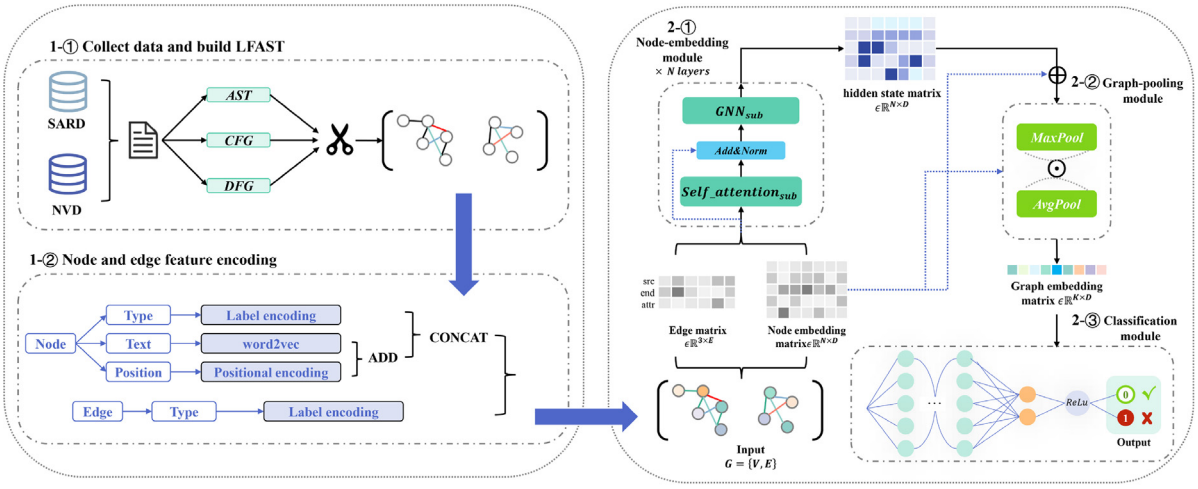

Fig. 2. Overall Architecture of *LCVD*.

cause a denial of service (Infinite Loop) via a crafted XML file. *ff_subtitles_next_line()*, which appears on lines 8 and 9, is used to get the number of characters to increment to jump to the next line or to the end of the string. However, if a \0 byte is Received, a negative error code will be returned. If a malicious attacker were to insert \0 byte, this code would cause an error that would lead to an infinite loop. To fix this vulnerability, control conditions are added inside the loop structure as shown in the red highlight on lines 9–12, to determine when an error may be thrown to exit the loop in time.

By analyzing loop-oriented vulnerabilities such as the one in Fig. 1, we find that those vulnerable parts are usually located inside or near the loop structure. While to fix them, patches need to be applied within this context as well. For example, to fix an infinite loop vulnerability, a counter may need to be declared outside the loop to limit the number of iterations of the loop. The reasons for the occurrence of loop-oriented vulnerabilities are summarized as follows: (1) inability to enter the loop resulting in missing some important operations; (2) inability to exit the loop (infinite loop); (3) forced exit loop. Vulnerable loops can cause damage such as denial of service, integer overflows, buffer overflows, system crashes, etc. Based on these observations, we propose a new graph representation for loop-oriented vulnerabilities called LFAST, and a specific graph neural network LFGNN to learn the implicit patterns in loop-oriented vulnerabilities.

3. Approach

In this section, we first show an overview of the *LCVD* approach. Next, we describe the process of code graph representation, i.e. how loop code is parsed into LFAST. Then, we describe the process of vulnerability feature extraction and detection in detail.

3.1. Approach overview

The overall architecture of *LCVD* is shown in Fig. 2, including two steps as follows.

In Step 1, loop-oriented code samples were collected and pre-processed. Firstly, function-level code blocks containing loop structures are collected from existing data sources. Each code block is parsed into LFAST and then transformed into a low-dimensional vector as input to the neural network model.

In Step 2, LFGNN is trained to learn the implicit features of loop-oriented vulnerabilities and detect the vulnerable functions. The low-dimensional matrix of LFASTs is first passed to the node-embedding module, where a new node-embedding matrix is obtained after message passing and updating. After that, the node-level embedding is aggregated to obtain the graph-level embedding through the graph-pooling module. Finally, The graph-level embedding is fed into a classification module, which predicts the target function-level loop samples whether vulnerable or not.

3.2. Loop-code representation

3.2.1. LFAST

To contain sufficient semantic properties and syntactic information, a common pre-processing is to represent the program as a structured intermediate form. Thus, we propose a graph representation called Loop-flow Abstract Syntax Tree (LFAST), which is defined as follows:

Definition 1 (*Loop-Flow Abstract Syntax Tree, LFAST*)**.** For a program $P = \{f_1, f_2, \ldots, f_n\}$, where f_n denotes the function contained in the program. For function f_i in program P, its LFAST is defined as $G = \{V, E\}$, which is a directed, edge-labeled, node-attributed, composite graph.

Definition 2 (*Loop-Oriented Nodes*)**.** $V = \{v_1, v_2, \ldots, v_i\} \in \mathbb{R}^{I \times J}$ denotes the node set consisting of I nodes inside or in a neighborhood of a loop structure extracted from AST, wherein I is the number of nodes and J is the dimension of the node features.

Definition 3 (*Loop-Oriented Edges*)**.** $E = \{E_{index}, E_{attr}\} \in \mathbb{R}^{3 \times D}$ denotes the edge set of D edges, wherein $E_{index} = \{[s_1, s_2, \ldots, s_d], [t_1, t_2, \ldots, t_d]\} \in \mathbb{R}^{2 \times D}$ denotes the index set of source and target edges, $E_{attr} = \{a_1, a_2, \ldots, a_d\} \in \mathbb{R}^{1 \times D}$ denotes the attributes of each edge.

The pseudo-code for building LFAST is described in Algorithm 1. Firstly, we parse the source code into an Abstract Syntax Tree (AST), a Control Flow Graph (CFG), and a Data Flow Graph (DFG). A heterogeneous composite graph is built with the AST as the skeleton, while control and data flow are added as edges (line 2). Secondly, We obtain the set of loop-oriented nodes by extracting nodes inside the loop structure and their k-hop neighbors on the graph from the heterogeneous graph (lines 4–14). After that, we obtain the set of loop-oriented edges by extracting all the edges between the loop-oriented nodes from the heterogeneous graph

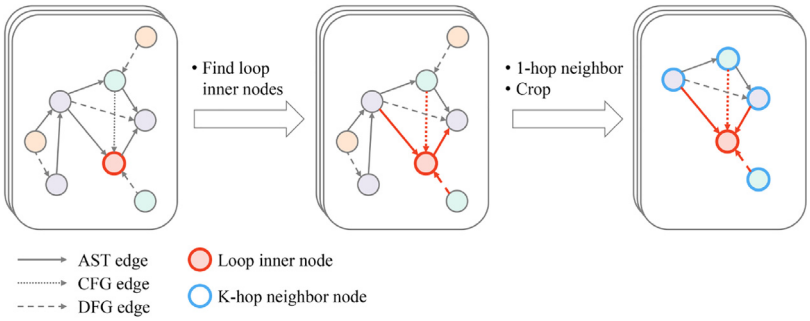

Fig. 3. Cropping of the composite graph through the loop-oriented nodes and edges to obtain LFAST with 1-hop neighbors.

Algorithm 1 build LFAST

Input: f: the function-level loop code
K: number of hops in the neighbourhood
Output: G_{LFAST}
1: // Initialize $V_{inner}, V_{neighbor}, V_{loop}, E_{loop}$ as empty list
2: $V_{inner}, V_{neighbor}, V_{loop}, E_{loop} \leftarrow [],[],[],[]$
3: // Step1: Generate composite graphs
4: $G'_f = \{V', E'\} \leftarrow \{V_f^{AST}, E_f^{AST+CFG+DFG}\}$
5: // Step2: Find nodes inside the loop or in the k-hop neighborhood
6: **for** each leaf node $v_i \in V'_f$ **do**
7: **if** v_i insides loop construction **then**
8: $V_{inner}.append(v_i)$
9: **for** each node $v_j \in V_{inner}$ **do**
10: $V_{neighbor}.append(k\_hop(V_{inner}))$
11: **end for**
12: **end if**
13: **end for**
14: $V_{loop} \leftarrow V_{inner} \cup V_{neighbor}$
15: // Step3: Add edges
16: **for** each edge $e_{ij} \in E'_f$ **do**
17: **if** $v_i \in V_{loop}$ or $v_j \in V_{loop}$ **then** $E_{loop}.append(e_{ij})$
18: **end if**
19: **end for**
20: // Step4: construct the LFAST by the sets of loop-oriented nodes and edges
21: $G_{LFAST} \leftarrow \{V_{loop}, E_{loop}\}$
22: **return** G_{LFAST}

(lines 16–19). The two aforementioned steps share similarities with the process of graph cropping, which involves generating a sub-graph with k-hop neighbors. The example of 1-hop neighbors is shown in Fig. 3. Finally, we build LFAST via the set of loop-oriented nodes and edges (lines 21–22).

3.2.2. Initialization of node and edge features

To construct the inputs to the neural network, graphs (LFAST) need to be converted to low-dimensional feature vectors. Although graph representations contain rich structural information, they may lack programming logic in the source code. To address this limitation, a positional encoding is added to the initial node features. We flattened the source code into a sequence of tokens in a row and numbered the tokens from left to right as positions. We then encode them by Position Encoding based on the sine and cosine functions (Vaswani et al., 2017).

The embedding vector of a source code sequence is assumed to be $X \in \mathbb{R}^{n \times d}$, which means a d-dimensional embedding of n tokens contained in a source code sequence. The position encoding vector has the same shape as the source code sequence vector, denoted as $P \in \mathbb{R}^{n \times d}$. The location vector of the ith token in the source code sequence is denoted as $p_i \in \mathbb{R}^d$. p_i^j is the element of the ith row and jth column in the location vector P, which is defined as:

$$p_i^{2j} = sin(w_j i) \tag{1}$$

$$p_i^{2j+1} = cos(w_j i) \tag{2}$$

where w_j denotes the frequency of the trigonometric function, here we quote the frequency used by Vaswani et al. (2017). Position Encoding uses interleaved encoding of sine and cosine functions, which can be linearly transformed to obtain the relative position between the markers.

Further, we encode types of tokens by Label Encoding and text (i.e. source code sequence) by *Word2Vec* (Mikolov et al., 2013). We add the text encoding $X \in \mathbb{R}^{n \times d}$ and positional encoding $P \in \mathbb{R}^{n \times d}$, and then combine the result with the type encoding $T \in \mathbb{R}^n$ to obtain the initial node feature vector as follows:

$$n_{emb} = [T \parallel (X + P)] \tag{3}$$

For edge features that indicate the relational properties between nodes, we divide them into three classes: syntax, control dependencies, and data dependencies. The initial feature vector of edges is encoded by Label Encoding.

3.3. Vulnerability feature extraction and detection

In order to uncover deeper features of loop-related vulnerabilities, we constructed and trained a new graph neural network called the Loop-Flow Graph Neural Network (LFGNN). We learned syntax and semantic features of loop-related vulnerabilities from the graph structure and ultimately obtained vulnerability detection results through a linear classification module. During the training phase, we used the labeled training dataset to train LFGNN and optimized model parameters by minimizing the cross-entropy loss function. In the detection phase, the well-trained model is applied to detect potential loop-related vulnerabilities in the testing dataset. In the following, we will describe the construction details of LFGNN first, and then introduce its application in the vulnerability detection phase.

3.3.1. Loop-Flow Graph Neural Network

The Loop-Flow Graph Neural Network (LFGNN) targets to capture the implicit loop-oriented vulnerability features. LFGNN is an end-to-end graph classification model, which contains a N-layers stacked node-embedding module, a graph-pooling module, and a classification module. The input is graphs parsed from the source code. The output is the predicted result of whether the source code represented by the graph is vulnerable or non-vulnerable.

(1) Node-embedding Module

The node-embedding module consists of two sub-layers: the self-attention sub-layer and the GNN sub-layer, which have added residual connections and layer normalization operations.

Self-attention sub-layer. The attention mechanism allows different importance to be assigned to different nodes, which compensates for the shortcoming that traditional GNNs do not have a focus but collect all domain information equally. Inspired by Velickovic et al. (2018), we designed the self-attention sub-layer to receive an initial graph embedding $G = \{V, E\}$ as input

and extract important features from the domain for each node along the edges of the graph.

First, for any node i in the set V of all nodes in G, we compute its similarity coefficient to its neighbor node j:

$$e_{ij} = a([Wh_i \parallel Wh_j \parallel w_{ij}]) \tag{4}$$

where h_i and h_j are the feature vectors of node i and j; w_{ij} is the attributes (weights) of the edges between node i and j; $W \in \mathbb{R}^{d \times d}$ denotes the shared weight parameter of the node features; $[\cdot \parallel \cdot]$ is the concatenation operation. Finally, a mapping $a(\cdot)$, the attention mechanism, is used to obtain the attention coefficients between node i and j. Here different network structures can be chosen as the attention mechanism function, we refer to Velickovic et al. (2018) using a simple single-layer feedforward neural network implementation. Then, the similarity coefficients of all nodes were normalized by textitsoftmax layer to obtain the attention coefficients:

$$a_{ij} = softmax(e)ij) = \frac{exp(LeakyReLU(e_{ij}))}{\sum_{k \in N_i} exp(LeakyReLU(e_{ik}))} \tag{5}$$

where N_i is the set of all nodes in the first-order domain connected to node i; e_{ij} is the similarity coefficients of node i and j; $LeakyReLU(\cdot)$ denotes a non-linear activation function. The feature vectors are then weighted and summed using the normalized attention coefficients, and the features of all nodes in the graph are updated to:

$$h_i' = \sigma(\sum_{j \in N_i} a_{ij} W h_j) \tag{6}$$

To stabilize the learning process in the self-attention sub-layer, we introduced multi-head attention (Vaswani et al., 2017):

$$h_i'(K) = \sigma(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in N_i} a_{ij}^k W_k h_j) \tag{7}$$

where K is the kth attention mechanism, $W_k \in \mathbb{R}^{d \times d}$ is the weight matrix of the kth head, and a_{ij}^k is the normalized attention coefficient obtained from the kth attention head.

GNN sub-layer. GNN sub-layers aim to iteratively aggregate features from local neighborhoods to learn new node embedding. There have been many excellent works on the graph including GCN (Kipf and Welling, 2017), GAT (Velickovic et al., 2018), GraphSage (Hamilton et al., 2017), GGNN (Scarselli et al., 2009), etc. We have chosen GCN and GGNN as the basic models for our GNN sub-layer. GCN is a popular graph neural network model that is widely used and GGNN performs well in the vulnerability detection task. Both networks are well suited for our data with graph structures.

For each graph convolution operation performed by the GCN, the node features are updated as follows:

$$h_i^{(l+1)} = \sigma(\sum_{j \in N_i} \frac{1}{c_{ij}} h_i^{(l)} w^{(l)}) \tag{8}$$

where $\frac{1}{c_{ij}} = \widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}}$ denotes the Laplacian re-normalized adjacency matrix with inserted self-loops and $\widetilde{D} \in \mathbb{R}^{d \times d}$ is the diagonal node degree matrix of $\widetilde{A}$. $\widetilde{A} = A + I_N \in \mathbb{R}^{d \times d}$, wherein I_N is the unit matrix. $h_i^{(l)}$ is the feature vector obtained at lth layer for the ith node, $w^{(l)}$ is the weight matrix at lth layer, and σ is a non-linear activation function.

GGNN adds GRU(Gated Recurrent Unit) (Cho et al., 2014) to GCN to enhance the long-term memory of the networks. We initialize the node state as $h_i^{(1)} = [x_i^T \parallel 0]^T \in \mathbb{R}^d$ for each time step $t \leq T$ within T time steps and the GGNN can obtain information about neighboring nodes through different types of edges, and the hidden state of the node at moment t is updated according to the following:

$$h_i^{(t)} = GRU(h_i^{(t-1)}, \sum_{j \in N_i} w_{ij} h_j^{(t-1)}) \tag{9}$$

where w_{ij} is the edge weight from source node j to target node i, since different edge attribute has different weight. We have implemented the aggregation function as a SUM function so that the hidden state of node i at time t needs to be weighted and summed over the neighborhood node features at time $t-1$. The above propagation process is iterated over T time steps and the state vector with the last time step $H^{(T)} = \{h_i^{(T)} \mid i = 1, 2, \ldots, n\}$ being the final node representation matrix of the node set V.

Residual connection. Residual connection (He et al., 2016) is used to merge information learned in lower layers to higher layers. Adding Residual connections allows gradients to pass directly through the layers to mitigate gradient disappearance or gradient explosion due to multiple iterations. We fix the same hidden size for different layers and add residual connections after the output of each of the self-attention sub-layers as shown in Fig. 2.

(2) Graph-Pooling and classification Module

As we consider loop-oriented code vulnerability detection as graph classification, a graph-level embedding is required to complete the downstream task. This module uses a hybrid of sum-pooling and max-pooling to generate the graph-level embedding, defined as follows:

$$g_i^{(Pool_m)} = MLP(\sigma(Pool_m([H_i^{(l)}, x_i])) * \sigma(Pool_m(x_i))) \tag{10}$$

$$g_i = g_i^{Pool_1} \odot g_i^{Pool_2} \tag{11}$$

where $H_i^{(l)}$ denotes the node-embedding matrix of the ith graph $G_i = \{V_i, E_i\}$ at the lth layer after passing the node-embedding Module. x_i is the original node-embedding matrix. $Pool_m$ denotes the chosen pooling method such as max-pooling, σ is the activate function and there we use $LeakyRelu$, $\odot$ is the mix function, e.g. addition or multiplication, and g_i is the final graph-level embedding obtained by graph-pooling Module.

After that, we feed the graph-level embedding into a classification module consisting of several MLP layers and activation functions to obtain the predictive label $\hat{y}_i$ as follows. In addition, we train the *LCVD* by minimizing the binary cross-entropy loss function.

$$\hat{y}_i = Sigmoid(MLP(Relu(g_i))) \tag{12}$$

3.3.2. Vulnerability detection

During the vulnerability detection phase, we apply the trained model to identify potential loop-related vulnerabilities in the input. The operations in this phase are similar to those in the training phase. First, we parse the input function-level code segment into the LFAST as a graph structure, and embed it into a low-dimensional vector through encoding, which serves as the input to the neural network. Next, we input the graph data into the well-trained LFGNN for vulnerability detection. The model outputs the predictive label $\hat{y}$ as a decimal number between 0 and 1, which we interpret as the probability that the input function is vulnerable. If the output is greater than 0.5, the function is predicted as "1" and considered vulnerable to attacks. On the other hand, if the output is less than or equal to 0.5, the function is predicted as "0" and considered not vulnerable.

Table 1
Details of loop-oriented vulnerability dataset.

Project	Version	Samples
Juliet Test Suite C/C++	1.2	3975
Linux kernel	2.6–5.9.2	264
PostgreSQL	9.2.4	44
Gimp	2.8.8	43
OpenSSL	1.0.1e	34
Subversion	1.8.3	31
Wireshark	1.10.2	31
FFMpeg	1.2.2	32
C-Tree	1.7.0	20
Others	/	14
Total	/	4488

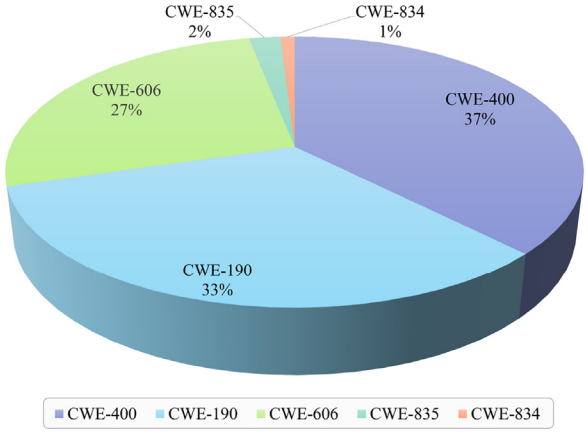

Fig. 4. Distribution of loop-oriented vulnerability types.

4. Experimental evaluation

4.1. Datasets

Since there is no accessible dataset of function-level loop-oriented vulnerabilities, we manually constructed a dataset to evaluate the proposed approach. We collected loop-oriented vulnerabilities from two existing data sources: SARD (Anon, 2022g) and NVD (Anon, 2022f), which are maintained by the National Institute of Standards and Technology (NIST) (Anon, 2022e). SARD contains a large number of generated, synthesized and academic security vulnerabilities (Li et al., 2018b), from which we collected loop-oriented test cases and extracted those labeled "bad" as vulnerable samples and those labeled "good" as non-vulnerable samples. For the real-world vulnerabilities included in the NVD, we built an automated crawler tool to collect information from the public dataset published by the NVD from 2011 to 2022 based on Common Weakness Enumeration IDentifier (CWE ID) (Anon, 2022a). We then extracted the loop-oriented vulnerable functions and corresponding patched functions as the vulnerable and non-vulnerable samples. Finally, We have collected a loop-oriented vulnerability dataset containing 4488 function-level samples in the C programming language, including 1561 loop-oriented vulnerable functions and 2927 loop-oriented non-vulnerable functions. The details of our dataset are shown in Table 1. Column 1 lists the names of the projects from which the vulnerabilities originate, including Linux kernel, PostgreSQL, Wireshark, and many other popular open-source projects in the C programming language. Column 2 lists the version numbers of the projects. Column 3 lists the number of loop-oriented vulnerabilities, which includes both vulnerable and non-vulnerable samples.

As there is no available label for loop-oriented vulnerabilities, we looked at the names and descriptions of CWE IDs to determine whether the sample was loop-oriented. The distribution of different types of loop-oriented vulnerabilities is shown in Fig. 4. From the perspective of the loop structure, we consider CWE-IDs that contain the word "loop" in their name or description. Thus We have selected CWE-606, -834, and -835. On the other hand, consider the consequences of loop-oriented vulnerabilities, which usually lead to problems with value overflows or resource leaks. For this, we have selected CWE-190 and CWE-400. Since not all vulnerabilities are related to loops, we manually filtered out non-loop-related vulnerabilities by examining the description of the vulnerability and the patch file.

We performed multiple filtering steps on the constructed dataset. Firstly, we removed all comments and references from the code. Secondly, we extracted multiple function-level blocks from a source code file to obtain vulnerable and non-vulnerable functions. Thirdly, to avoid the problem of vocabulary explosion and to exclude the effect of specific names, we map user-defined function names and variable names to symbolic names in a one-to-one way (e.g., "FUNC1", " VAR1") refer to Li et al. (2018b).

4.2. Implementations

We implemented *LCVD* in Python using PyTorch (Anon, 2022h), with PyTorch Geometric(PyG) (Fey and Lenssen, 2019) as the graph neural network framework. In the data-preprocessing phase, we parse source code by Joern (Anon, 2022d), and used a random ratio of 8:1:1 to split the dataset. We used the WeightedRandomSampler to balance the vulnerable and non-vulnerable samples in each batch in training. In the experiments, considering GPU memory and training time, the max number of nodes in a graph is set to 500 and the embedding size of *Word2Vec* is set to 100. For hyper-parameters of LFGNN, the node-embedding module combined with residual connections is stacked with 2 layers, and the hidden state size is set to 300. The number of layers of GRU units in the GGNN is set to 2. The neural network was trained in batches until convergence, with an early stop to 100 epochs, batch size of 32, dropout of 0.2, and the random seed of 123456. We used the ADAM (Kingma and Ba, 2015) optimization algorithm for training models with a learning rate of 1e−4 and weight decay of le−3.

4.3. Evaluation metrics

We use the following widely used evaluation metrics to measure the effectiveness of the proposed approach.

Accuracy (A) refers to the proportion of the number of samples correctly predicted to the total number of samples. It is calculated as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{13}$$

Precision (P) refers to how many of the samples that we predict to be having vulnerabilities are vulnerable. It is calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{14}$$

Recall (R) refers the proportion of correctly classified vulnerability samples to all actual vulnerability samples. It is calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{15}$$

F1 score (F1) can be seen as a weighted average of the model's precision and recall. It is calculated as:

$$F1score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{16}$$

where TP indicates the number of vulnerable samples detected as vulnerable; FP indicates the number of samples that are non-vulnerable but are detected as vulnerable; TN indicates the number of samples that are non-vulnerable and are detected as non-vulnerable; and FN indicates the number of vulnerable samples that are detected as non-vulnerable.

4.4. Research questions

RQ1. How does *LCVD* perform in loop-oriented vulnerability detection compared to other vulnerability detection approaches?

To answer this question, We compared *LCVD* with three open-source static analysis tools, including *CppChecker* (Anon, 2022b), *Flawfinder* (Anon, 2022c), and *RATS* (Anon, 2022i). We also compared *LCVD* with the following four state-of-the-art DL-based VD approaches:

- ***Vuldeepecker*** (Li et al., 2018b) represents source code as sequences and uses 2-layers Bi-LSTM for vulnerability detection.
- ***CodeBERT*** (Feng et al., 2020) is a pre-trained model based on BERT for 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go), and we use the implementation by Lu et al. (2021).
- ***Devign*** (Zhou et al., 2019) represents the source code as a joint graph of AST, CFG, PDG, and NCS (natural code sequence) and updates the node representation using GGNN. It uses the 1-D CNN-based Conv module to generate graph representation of the source code and finally performs vulnerability detection via a fully connected layer. Since Zhou et al. (2019) do not publish the code implemented by *Devign*, we have reproduced from the original paper.
- ***ReGVD*** (Nguyen et al., 2022) represents the source code as a flat set of token sequences and constructs the graph in the form of a sliding window. It uses residual connections among GNN layers for node updates and then generates a graph-level embedding by a mixture between sums and max pools with a soft attention mechanism.

RQ2. Which graph-based representations of source code are more advantageous for representing loop-oriented vulnerabilities?

This question aims to investigate the performance of different graph-based representations of source code in loop-oriented vulnerability detection. We have considered different graph structures and encoding approaches separately.

RQ3. Does each component of LFGNN contribute to the performance of loop-oriented vulnerability detection?

This question aims to investigate the effectiveness of the node-embedding module and the graph-pooling module in extracting features of loop-oriented vulnerabilities since both play important roles in LFGNN. We do not consider the classification module because it is similar to other works.

5. Experimental results & analysis

5.1. Comparison with other VD approaches (RQ1)

Results. Table 2 shows the performance of some static analysis-based and DL-based approaches compared to *LCVD* in terms of evaluation metrics. Overall, *LCVD* outperformed the above seven approaches on the loop-oriented vulnerability dataset while using LFAST as a graph representation of the source code. *LCVD(GGNN)*, which refers to *LCVD* using GGNN as the GNN sub-layer in the node-embedding module, achieves the best results with 89.29% in Accuracy, 90.57% in Precision, 90.06% in Recall, and 89.28% in F1 score. Compared to the static analysis-based baselines, *LCVD(GCN)* averagely improves by 36.90%–47.56% and *LCVD(GGNN)* improves 37.55%–48.21%. DL-based approaches are generally more accurate than static analysis-based ones. *LCVD(GGNN)* outperforms four DL-based baselines with an average improvement of 17.49%–18.54% on evaluation metrics and 7.15%–8.48% compared to the best-performing baseline *CodeBERT*. *LCVD(GCN)* performs slightly worse than *LCVD(GGNN)* but still outperforms all the four DL-based baselines, averaging 16.84%–17.88% improvement over the four baselines and 6.49%–7.82% improvement over the best-performing baseline in terms of each metric.

Table 2
Comparison with other vulnerability detection approaches.

Approach	A(%)	P(%)	R(%)	F1(%)
Static analysis-based approaches				
CppCheck	52.92%	69.75%	52.00%	38.54%
Flawfinder	51.30%	51.56%	51.50%	50.89%
RATS	50.97%	25.49%	50.00%	33.76%
DL-based approaches				
Vuldeepecker	64.45%	65.63%	65.41%	64.43%
CodeBERT	82.14%	82.09%	82.32%	82.10%
Devign	59.42%	59.34%	57.92%	57.00%
ReGVD	81.17%	81.05%	80.98%	81.01%
LCVD(GCN)	88.64%	89.91%	89.41%	88.63%
LCVD(GGNN)	**89.29%**	**90.57%**	**90.06%**	**89.28%**

Analysis. It is clear from the experimental results that *LCVD* outperforms all the baselines, whether using GGNN or GCN as the node-embedding layer. This demonstrates that *LCVD* can learn the internal structure of the loop code to better distinguish between loop-oriented vulnerabilities and non-vulnerabilities. Compared to static analysis-based approaches, *LCVD* does not rely on explicit features to detect vulnerabilities, but rather learns implicit loop-oriented vulnerability patterns from large amounts of data, resulting in significant improvements. Compared to the DL-based approaches, we will analyze the reasons for the better performance of *LCVD* in terms of both the code representation and the deep neural network:

- **Code representation.** Compared to sequence-based VD approaches (*Vuldeepecker*, *CodeBERT*), *LCVD* uses a graph structure as a source code representation which contains more structural information. Compared to *Devign*, *LCVD* crops the graph representation to highlight loop structure and combines information about natural code sequences into the initial encoding of nodes. In contrast to *ReGVD*, *LCVD* adds structure information from AST, CFG, and DFG to the graph representation, enriching the syntax and semantics information.
- **Deep neural network.** *LCVD* employs GNN as the deep neural network model because it uses graph representations for the source code. In the phase of node embedding, *LCVD* adds a self-attention sub-layer and residual connections to GGNN compared to *Devign* which uses GGNN to update node features. Compared with *ReGVD* using GNNs with residual connection, *LCVD* adds a self-attention layer. In the phase of graph embedding, compare to other approaches, *LCVD* considers both the original node features and the node features obtained by the graph neural network. These improvements enable *LCVD* to focus on nodes closely related to loop-oriented vulnerabilities.

Answer to RQ1. *LCVD* outperforms three static analysis-based and four state-of-the-art DL-based VD approaches in all metrics, indicating that *LCVD* can capture the features of loop-oriented vulnerabilities better.

Table 3
Comparison with different graph representations for source code.

Structure	Node features			*LCVD(GCN)*				*LCVD(GGNN)*			
	Type	*Text*	*Position*	A(%)	P(%)	R(%)	F1(%)	A(%)	P(%)	R(%)	F1(%)
AST	✓			61.04%	64.71%	58.56%	55.10%	**90.26%**	90.47%	**90.66%**	**90.26%**
	✓	✓		60.06%	59.84%	59.85%	59.84%	61.04%	60.95%	61.00%	60.93%
		✓	✓	61.04%	60.69%	60.29%	60.23%	62.01%	62.04%	60.79%	60.35%
	✓	✓	✓	87.99%	88.59%	88.55%	87.99%	87.99%	89.66%	88.86%	87.97%
CFG	✓			85.71%	86.37%	86.29%	85.71%	89.29%	90.05%	89.91%	89.28%
	✓	✓		56.49%	57.77%	53.68%	48.08%	89.61%	89.61%	89.85%	89.59%
		✓	✓	60.06%	59.69%	59.18%	59.03%	88.96%	89.96%	89.66%	88.96%
	✓	✓	✓	87.01%	88.56%	87.85%	86.99%	57.14%	58.41%	58.05%	56.92%
AST+CFG+DFG	✓			61.04%	60.73%	60.14%	59.98%	88.64%	90.11%	89.46%	88.62%
	✓	✓		87.01%	88.78%	87.90%	86.99%	89.29%	90.21%	89.96%	89.28%
		✓	✓	62.01%	61.77%	61.76%	61.76%	56.82%	57.34%	57.29%	56.81%
	✓	✓	✓	62.01%	61.77%	61.76%	61.76%	88.96%	90.34%	89.76%	88.95%
LFAST	✓			85.71%	86.37%	86.29%	85.71%	88.64%	89.55%	89.31%	88.63%
	✓	✓		56.49%	57.77%	53.68%	48.08%	64.61%	65.26%	63.25%	62.74%
		✓	✓	60.06%	59.69%	59.18%	59.03%	63.31%	63.08%	62.55%	62.51%
	✓	✓	✓	**88.64%**	**89.91%**	**89.41%**	**88.63%**	89.29%	**90.57%**	90.06%	89.28%

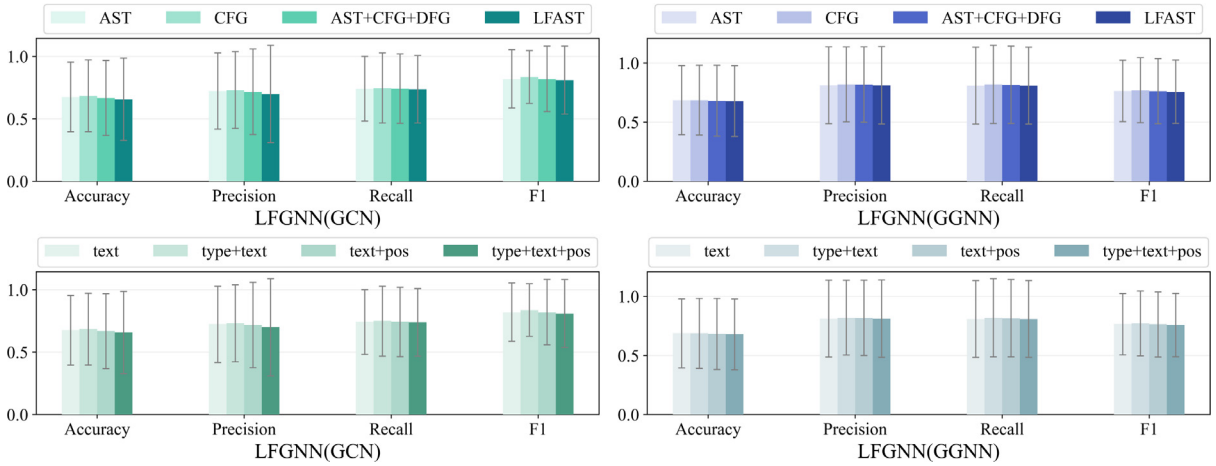

Fig. 5. Min–max bars show the performance of vulnerability detection across different code graph structures and initial node features.

5.2. Comparison with different graph representations for source code (RQ2)

For the graph representation of the source code, we will focus on two components: graph structures and initial node features. For the graph structures of the source code, we parse the source code as AST, CFG, combined AST+CFG+DFG, and LFAST. We did not use separate DFG graphs here because for the loop-oriented vulnerability dataset we constructed, much of the source code samples contain little or no information after being parsed into DFG by Joern. For initial node features, we used pure text, (type+text), (text+pos), and (type+text+pos) as the initial features for encoding to obtain different vectors respectively. We conducted experiments on LFGNN(GCN) and LFGNN(GGNN). The graph pooling method used the proposed hybrid of max-pooling and sum-pooling with the Multiplication operator.

Result. Table 3 shows the impact of different graph structures and initial features of nodes on the performance of vulnerability detection. For *LCVD(GCN)*, it achieves the best performance when using LFAST/(type+text+pos), which means using LFAST as the graph structure and using type+text+position as the initial node feature, with the 88.64% in Accuracy, 89.91% in Precision, 89.41% in Recall, and 88.63% in F1 score. For *LCVD* (GGNN), AST/text achieves the best Accuracy of 90.26%, Recall of 90.66%, and F1 score of 90.26%. Our proposed LFAST/(type+text+pos) outperforms AST/text for *LCVD(GGNN)* by up to 0.1% in Precision but is slightly below it in terms of other metrics.

Analysis. We have plotted additional charts for Table 3 for ease of analysis. Fig. 5 show the performance of vulnerability detection by the min–max bars across the different graph structures and initial node features. For *LCVD(GCN)*, the highest performance was achieved by LFAST/(type+text+pos). We believe that since GCN loses much information when processing the graph structure constructed from source code (Siow et al., 2022), the input graph nodes contain more initial features of the source code can help GCN capture the vulnerability features. Besides, the average accuracy is 87.5% when using our proposed (type+text+pos) as the initial features of nodes on four different graph structures, indicating that the combination of type, text, and position information can help *LCVD(GCN)* improve the performance of vulnerability detection.

But for *LCVD* (GGNN), the highest performance was achieved by AST/text. As shown in Fig. 6, the accuracy of our proposed LFAST/(type+text+pos) is 0.97% lower than AST/text, but the precision is 0.10% higher than AST/text. In the process of training the model, we found that for GGNN, using more information as the initial features of the nodes sometimes led to premature overfitting during training, which resulted in a decrease in the effect after adding node type or node location information. We consider that is because GGNN is recognized as more suitable for code compared to GCN (Scarselli et al., 2009; Siow et al., 2022), where the former is far better at capturing code structure information, and inputting too much feature information may restrict the model to local details.

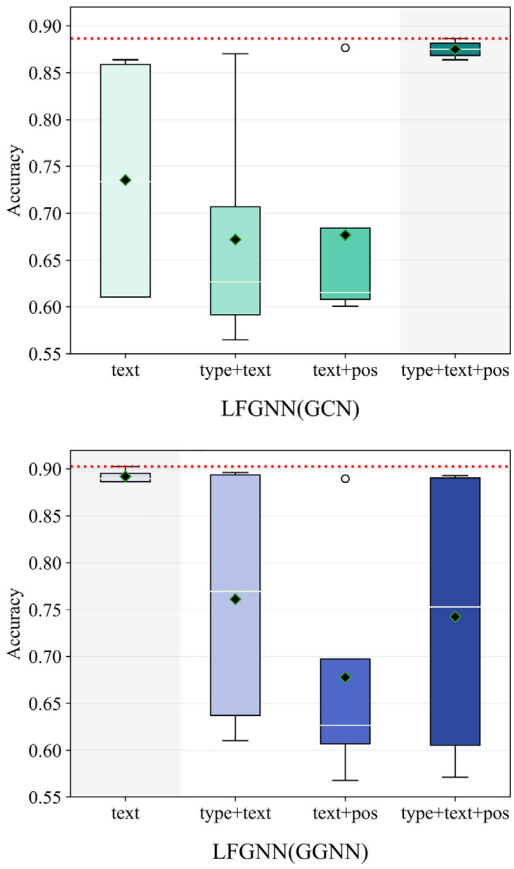

Fig. 6. Comparing the accuracy of vulnerability detection with different initial node features.

Table 4
Compare with other popular GNNs.

Model	A(%)	P(%)	R(%)	F1(%)
GAT	86.36%	87.17%	86.99%	86.36%
GGNN	87.34%	87.25%	87.44%	87.30%
GCN	63.64%	64.18%	62.24%	61.66%
LFGNN(GCN)	88.64%	89.91%	89.41%	88.63%
LFGNN(GGNN)	**89.29%**	**90.57%**	**90.06%**	**89.28%**

Answer to RQ2. Using LFAST/(type+text+pos) can help *LCVD (GCN)* achieve better detection performance, while for *LCVD(GG NN)*, AST /text works better.

5.3. Effectiveness of LFGNN components (RQ3)

We decompose the problem into two sub-problems, focusing on the effectiveness of the node-embedding module and the graph-pooling module respectively.

RQ3.1 Does the node-embedding module in LFGNN outperform some existing GNNs in loop-oriented vulnerability detection?

Table 4 shows the experimental results of the node-embedding module of LFGNN in comparison with other popular GNNs. We observe that both LFGNN(GCN) and LFGNN(GGNN) outperform the baselines, improving by 1.30%–2.66% and 1.95%–3.32% respectively compared to the best-performing baseline GGNN. In particular, it can be seen from the experimental results that the performance using only GCN is poor, but the LFGNN (GCN) shows a significant improvement compared to it, with 25.00%–27.16% improvement in all metrics. There are two main reasons. First, compared to GAT, LFGNN adds multi-layers GNN to expand the perceptual field while focusing on first-order neighbors through a self-attentive mechanism. Secondly, compared to GCN and GGNN, the node-embedding module of LFGNN combines the self-attention mechanism and residual connectivity on top of the graph neural network to focus more on the loop-oriented vulnerability features.

RQ3.2 Does the graph-pooling module outperform some existing graph-pooling methods in loop-oriented vulnerability detection?

We propose a hybrid graph-pooling module that uses a linear mixture of max-pooling and sum-pooling. For this question, we will compare it with the 1D-CNN Conv pooling module proposed by *Devign* (Zhou et al., 2019), and the mixture pooling module proposed by *ReGVD* (Nguyen et al., 2022). In addition, we compare with some common graph pooling methods, including max-pooling, sum-pooling and average-pooling.

Table 5 shows the result of using different graph pooling methods. LinearMix represents our proposed hybrid graph pooling method, while LinearMix(MUL) indicates the use of the multiplication operator as a hybrid method and LinearMix(SUM) indicates the use of the addition operator. For LFGNN(GCN), the best performance is obtained by Linear Mix(MUL) which improves 4.86%–5.63% in metrics over the best-performing baseline average-pooling. For LFGNN(GGNN), LinearMix(MUL) also outperformed the other baselines, improving over the best-performing baseline sum-pooling ranging of 1.62%–1.93%.

In contrast to the graph pooling methods proposed by *Devign* or *ReGVD*, our proposed hybrid graph pooling method combines sum-pooling and max-pooling, taking into account the structural information of the graph. When computing the graph embedding, the initial features of the graph nodes are added in addition to the node features updated by GNNs, avoiding the loss of the original information due to computation over multiple layers of the deep network.

Answer to RQ3. The node-embedding module and the graph-pooling module play important roles in LFGNN, both of which have improved the performance of *LCVD* for vulnerability detection.

6. Threats to validity

Internal validity. The main internal threat to our study comes from the hyperparameters we set for our baseline model. Different hyperparameter settings can affect the baseline experimental results. We trained the baseline model with the default parameters from the original paper or tried our best to re-implement it as described in the original paper, but some deviations may still lead to uncertainty in the experimental results.

External validity. The external threats to our study are mainly from the loop-oriented vulnerability dataset we use. Since loop-oriented vulnerabilities are less marked in real projects, the majority of loop-oriented vulnerabilities in our collected dataset are synthetic academic vulnerabilities rather than real-world ones. Another threat is that we currently filter loop-oriented vulnerabilities manually due to the lack of filters to identify loop-oriented vulnerabilities, which may introduce certain errors. Furthermore, we have only experimented with samples in the C/C++ programming language so far. Though our approach is general, we still need further work to validate the applicability of our model to other programming languages. We also found that using only AST is even more effective than our proposed LFAST in some cases. We still need to continue to investigate code graph representations that are more applicable to loop structures in the future.

Table 5
Compare with different graph pooling methods.

Pooling	LFGNN(GCN)				LFGNN(GGNN)			
	A(%)	P(%)	R(%)	F1(%)	A(%)	P(%)	R(%)	F1(%)
Devign	65.26%	65.37%	64.26%	64.12%	62.01%	61.73%	61.20%	61.11%
ReGVD	65.91%	67.04%	64.45%	63.88%	53.90%	26.95%	50.00%	35.02%
max-pooling	79.55%	79.48%	79.65%	79.50%	87.01%	88.56%	87.85%	86.99%
sum-pooling	82.47%	83.54%	83.17%	82.46%	87.66%	88.65%	88.35%	87.66%
average-pooling	83.77%	84.28%	84.28%	83.77%	87.66%	87.57%	87.64%	87.60%
LinearMix(SUM)	73.05%	72.89%	72.91%	72.90%	84.74%	85.32%	85.28%	84.74%
LinearMix(MUL)	**88.64%**	**89.91%**	**89.41%**	**88.63%**	**89.29%**	**90.57%**	**90.06%**	**89.28%**

7. Related work

This paper aims to investigate the representation and detection of loop-oriented vulnerabilities based on DL, broadly belonging to the task of program comprehension. We have divided the related work into two parts: source code representation and vulnerability detection with DL. The former is the basis for many DL-based program comprehension tasks, while the latter is the purpose of this paper.

7.1. Source code representation

The considerable success of source code representation techniques has been achieved in code classification (Lu et al., 2019; Mehrotra et al., 2022), code completion (Li et al., 2018a; Wang and Li, 2021), clone detection (Yu et al., 2019; Wang et al., 2020), vulnerability detection (Li et al., 2022; Zou et al., 2021; Li et al., 2018b; Zhou et al., 2019; Cao et al., 2021), and other programming comprehension tasks. Source code is usually represented as an intermediate form (e.g., AST) based on programming language rules, and then learned the implicit semantic information via the DL model. Several works (Zhang et al., 2019; Mou et al., 2014; Li et al., 2018a) represent the source code as AST and learn the structural information by traversing tree-based structures. Li et al. (2018a) construct sequences by a depth-first traversal of AST according to in-order, and then learn the contextual information in the source code by RNN. Wang and Li (2021) flatten AST and then model them as an AST graph, and use the graph attention module to capture different dependencies in the AST graph. What is more, some works (Yamaguchi et al., 2014; Allamanis et al., 2018; Zhou et al., 2019; Cao et al., 2021) combine information from multiple program graphs, such as the code property graph proposed by Fabian et al. (Yamaguchi et al., 2014), which combines properties from AST, CFG and PDG in a joint data structure to detect the code vulnerabilities through traversal of the graph. Allamanis et al. (2018) extended AST to program graphs containing multiple syntactic and semantic relations by adding different dependencies as edges to the AST.

In contrast to existing source code representation techniques, we have designed a graph representation for loops called LFAST. Unlike existing approaches, LFAST focuses on the interleaving relationships of multi-paths in the loop structure.

7.2. Vulnerability detection with DL

DL-based vulnerability detection approaches can be divided into two categories from the perspective of code representation: sequence-based and graph-based. Several works (Li et al., 2018b, 2022; Zou et al., 2021) have attempted to apply natural language processing techniques to vulnerability detection, which consider code as a sequence of tokens and use RNNs to learn semantic information about the code. Li et al. (2018b) proposed *Vuldeepecker* based on DL, which uses a program slicing technique based on heuristics to process vulnerable programs as code gadgets and train a Bi-LSTM model for vulnerability detection. However, there are still limitations in the form of text sequences in terms of representation logic and structure. Therefore, recent works (Zhou et al., 2019; Cao et al., 2021; Nguyen et al., 2022; Cao et al., 2022) focus on graph-based vulnerability detection approaches. For example, Zhou et al. (2019) proposed *Devign*, which represents the vulnerable source code as a joint graph while using GGNN to extract vulnerability features. Cao et al. (2021) proposed *BGNN4VD* to extract syntactic and semantic information from source code through a variety of code graph structures and constructed a bi-directional graph neural network (BGNN) based vulnerability detection framework. Nguyen et al. (2022) proposed *ReGVD*, which flattens the sequence of tokens into a graph structure and uses GNN with residual connections and a graph-level mixture graph pooling based on soft attention mechanisms for graph classification.

Our approach differs from existing DL-based vulnerability detection approaches by designing a dedicated LFGNN for focusing on loop structures and extracting loop-oriented vulnerability features.

8. Conclusion

We introduce *LCVD*, a loop-oriented vulnerability detection approach. *LCVD* represents the source code as a heterogeneous graph structure named LFAST to focus on loop structures. Then a novel LFGNN is proposed to learn the implicit features of loop-oriented vulnerabilities and detect the vulnerable functions. To evaluate the effectiveness of *LCVD*, we construct a loop-oriented vulnerability dataset containing 4488 samples. *LCVD* was compared with three static analysis-based and four state-of-the-art DL-based VD approaches. The experimental results show that *LCVD* is more effective in dealing with loop-oriented vulnerabilities, with an improvement of 36.36%–50.74% compared to the best-performing static analysis-based baseline and 7.13%–8.48% compared to the best-performing DL-based baseline in the term of each evaluation metric.

In future work, we aim to expand the loop-oriented vulnerability dataset to cover more vulnerabilities that are closely related to loops, as our current work still has limitations in this regard. Additionally, we plan to continue exploring effective representations that are more suitable for loops on larger datasets. Besides, we will consider approaches that are more fine-grained and interpretable for loop-oriented vulnerability detection.

CRediT authorship contribution statement

Mingke Wang: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Visualization, Writing – original draft, Writing – review & editing. **Chuanqi Tao:** Conceptualization, Validation, Investigation, Resources, Writing – review & editing, Supervision, Project administration. **Hongjing Guo:** Conceptualization, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chuanqi Tao reports financial support was provided by National Key Research and Development Program of China Stem Cell and Translational Research. Chuanqi Tao reports a relationship with National Key Research and Development Program of China (2018YFB1003900) that includes: funding grants. Chuanqi Tao reports a relationship with Open Fund of the State Key Laboratory for Novel Software Technology (KFKT2021B32) that includes: funding grants. Chuanqi Tao reports a relationship with Research Funds for the Central Universities (NO.NS2019058) that includes: funding grants. Chuanqi Tao reports a relationship with China Postdoctoral Science Foundation Funded Project (No.2019M651825) that includes: funding grants.

Data availability

Data will be made available on request

Acknowledgments

This work is supported by National Key R&D Program of China (2018YFB1003900), Open Fund of the State Key Laboratory for Novel Software Technology (KFKT2021B32), Fundamental Research Funds for the Central Universities (NO.NS2019058), and China Postdoctoral Science Foundation Funded Project (No. 2019M651825)

References

Allamanis, M., Brockschmidt, M., Khademi, M., 2018. Learning to represent programs with graphs. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net.

Anon, 2022a. Common weakness enumeration. https://cwe.mitre.org/. (Accessed 10 December 2022).

Anon, 2022b. Cppchecker. https://cppcheck.sourceforge.io/. (Accessed 10 December 2022).

Anon, 2022c. Flawfinder. http://www.dwheeler.com/flawfinder. (Accessed 10 December 2022).

Anon, 2022d. Joern. https://joern.io/. (Accessed 10 December 2022).

Anon, 2022e. National Institute of Standards and Technology. https://www.nist.gov/. (Accessed 10 December 2022).

Anon, 2022f. National vulnerability database. https://nvd.nist.gov/. (Accessed 10 December 2022).

Anon, 2022g. NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD/. (Accessed 10 December 2022).

Anon, 2022h. PyTorch. https://pytorch.org/. (Accessed 10 December 2022).

Anon, 2022i. Rough audit tool for security. https://code.google.com/archive/p/rough-auditing-tool-for-security/. (Accessed 10 December 2022).

Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y., 2009. Bounded model checking. In: Handbook of Satisfiability, Vol. 185, No. 99. pp. 457–481.

Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. *BGNN4VD*: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136, 106576. http://dx.doi.org/10.1016/j.infsof.2021.106576.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: 44th IEEE/ACM 44th International Conference on Software Engineering. ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, ACM, pp. 1456–1468. http://dx.doi.org/10.1145/3510003.3510219.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Moschitti, A., Pang, B., Daelemans, W. (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP 2014, October 25-29, 2014, Doha, Qatar, a Meeting of SIGDAT, a Special Interest Group of the ACL, ACL, pp. 1724–1734. http://dx.doi.org/10.3115/v1/d14-1179.

Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2020. Online Event, 16-20 November 2020, In: Findings of ACL, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.

Fey, M., Lenssen, J.E., 2019. Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds.

Godefroid, P., Luchaup, D., 2011. Automatic partial loop summarization in dynamic test generation. In: Dwyer, M.B., Tip, F. (Eds.), Proceedings of the 20th International Symposium on Software Testing and Analysis. ISSTA 2011, Toronto, on, Canada, July 17-21, 2011, ACM, pp. 23–33. http://dx.doi.org/10.1145/2001420.2001424.

Hamilton, W.L., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017. December 4-9, 2017, Long Beach, CA, USA, pp. 1024–1034.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, IEEE Computer Society, pp. 770–778. http://dx.doi.org/10.1109/CVPR.2016.90.

Kingma, D.P., Ba, J., 2015. Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net.

Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M., 2013. Loop summarization using state and transition invariants. Form. Methods Syst. Des. 42 (3), 221–261.

Li, J., Wang, Y., Lyu, M.R., King, I., 2018a. Code completion with neural attention and pointer networks. In: Lang, J. (Ed.), Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, ijcai.org, pp. 4159–4165. http://dx.doi.org/10.24963/ijcai.2018/578.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2022. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secur. Comput. 19 (4), 2244–2258. http://dx.doi.org/10.1109/TDSC.2021.3051525.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018b. VulDeePecker: A deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium. NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S., 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In: Vanschoren, J., Yeung, S. (Eds.), Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021. December 2021, Virtual.

Lu, M., Tan, D., Xiong, N., Chen, Z., Li, H., 2019. Program classification using gated graph attention neural network for online programming service, CoRR abs/1903.03804. arXiv:1903.03804.

Mehrotra, N., Agarwal, N., Gupta, P., Anand, S., Lo, D., Purandare, R., 2022. Modeling functional similarity in source code with graph-based siamese networks. IEEE Trans. Softw. Eng. 48 (10), 3771–3789. http://dx.doi.org/10.1109/TSE.2021.3105556.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. In: Bengio, Y., LeCun, Y. (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.

Mou, L., Li, G., Jin, Z., Zhang, L., Wang, T., 2014. TBCNN: A tree-based convolutional neural network for programming language processing, CoRR abs/1409.5718. arXiv:1409.5718.

Nguyen, V., Nguyen, D.Q., Nguyen, V., Le, T., Tran, Q.H., Phung, D., 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In: 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings. ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022, ACM/IEEE, pp. 178–182. http://dx.doi.org/10.1145/3510454.3516865.

Russell, R.L., Kim, L.Y., Hamilton, L.H., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P.M., McConley, M.W., 2018. Automated vulnerability detection in source code using deep representation learning. In: Wani, M.A., Kantardzic, M.M., Mouchaweh, M.S., ao Gama, J., Lughofer, E. (Eds.), 17th IEEE International Conference on Machine Learning and Applications. ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, IEEE, pp. 757–762. http://dx.doi.org/10.1109/ICMLA.2018.00120.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2009. The graph neural network model. IEEE Trans. Neural Netw. 20 (1), 61–80. http://dx.doi.org/10.1109/TNN.2008.2005605.

Sharma, R., Dillig, I., Dillig, T., Aiken, A., 2011. Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. In: Lecture Notes in Computer Science, 6806, Springer, pp. 703–719. http://dx.doi.org/10.1007/978-3-642-22110-1_57.

Siow, J.K., Liu, S., Xie, X., Meng, G., Liu, Y., 2022. Learning program semantics with code representations: An empirical study. In: IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2022, Honolulu, HI, USA, March 15-18, 2022, IEEE, pp. 554–565. http://dx.doi.org/10.1109/SANER53432.2022.00073.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y., 2018. Graph attention networks. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30–May 3, 2018, Conference Track Proceedings. OpenReview.net.

Wang, Y., Li, H., 2021. Code completion by modeling flattened abstract syntax trees as graphs. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021. Virtual Event, February 2-9, 2021, AAAI Press, pp. 14015–14023.

Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2020, London, on, Canada, February 18-21, 2020, IEEE, pp. 261–271. http://dx.doi.org/10.1109/SANER48275.2020.9054857.

Xiao, X., Li, S., Xie, T., Tillmann, N., 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 246–256.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. SP 2014, Berkeley, CA, USA, May 18-21, 2014, IEEE Computer Society, pp. 590–604. http://dx.doi.org/10.1109/SP.2014.44.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: Guéhéneuc, Y., Khomh, F., Sarro, F. (Eds.), Proceedings of the 27th International Conference on Program Comprehension. ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp. 70–80. http://dx.doi.org/10.1109/ICPC.2019.00021.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering. ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhou, Y., Liu, S., Siow, J.K., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019. NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 10197–10207.

Zou, D., Wang, S., Xu, S., Li, Z., Jin, H., 2021. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secur. Comput. 18 (5), 2224–2236. http://dx.doi.org/10.1109/TDSC.2019.2942930.

Mingke Wang received a B.Eng. degree in software engineering from the Zhejiang University of Technology, Hangzhou, China, in 2021. She is pursuing an M.E. degree with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. Her research interests include vulnerability detection and artificial intelligence in software engineering.

Chuanqi Tao received a Ph.D. degree from Southeast University, Nanjing, China, in 2013. He is currently an Associate Professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. His research interests include intelligent software development and testing.

Hongjing Guo is currently working toward a Ph.D. degree with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. Her research interests include AI software testing and software repository mining.