The Journal of Systems and Software 226 (2025) 112442

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Improving distributed learning-based vulnerability detection via multi-modal prompt tuning☆

Zilong Ren [a], Xiaolin Ju [a,*], Xiang Chen [a,*], Yubin Qu [b]

[a] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China
[b] Jiangsu College of Engineering and Technology, Nantong, China

ARTICLE INFO

Keywords:
Vulnerability detection
Multiple modalities
Deep learning
Distributed learning

ABSTRACT

Software vulnerabilities pose significant threats to the integrity and reliability of complex systems, making their detection critical. In recent years, a growing body of research has explored deep learning-based approaches for identifying vulnerabilities, which have shown promising results. However, many of these methods ignore privacy and security issues. We utilize distributed learning techniques that enable local models to interact without data sharing. By aggregating these locally trained models, we can update the global model while maintaining data privacy and security. Additionally, existing methods rely on a single source of code semantic information. However, leveraging multiple modalities can capture diverse code representations and features. Specifically, graph-based representations and source code provide structural and syntactic-semantic information that complements traditional code analysis. In this study, we propose a novel function-level vulnerability detection approach MIVDL. It integrates both structured and unstructured features of source code. Then, it further combines the code token sequence with the Code Property Graph (CPG) for enhanced detection accuracy. This hybrid representation leverages the strengths of different modalities to provide a comprehensive understanding of code semantics. Furthermore, our approach employs a pre-trained model applied to distinct parts of each modality before being integrated into a single hybrid representation. This allows a unified analysis framework to utilize each modality's unique features and strengths. Additionally, distributed learning facilitates collaborative learning and knowledge-sharing among participating entities. We evaluate MIVDL on three datasets (Devign, Reveal, and Big-Vul), and the results indicate that MIVDL outperformed eight state-of-the-art baselines by 3.04~70.73% in terms of F1-score. Therefore, combining multi-modal prompt tuning and distributed learning can improve performance in vulnerability detection.

1. Introduction

A software vulnerability is a security flaw or weakness within computer software that attackers can exploit to compromise system security (Johnson et al., 2011; Anon, 2020; Nord, 2017). For instance, the CVE-2023-20198 vulnerability can be exploited by unauthenticated remote attackers to target Cisco software devices. Reports from the Norwegian National Security Authority indicate that more than 41,000 Cisco devices have been targeted and compromised. This highlights the importance of software vulnerability detection. Therefore, developing effective automatic software vulnerability detection methods is crucial for ensuring the security of software systems.

Currently, the existing methods for detecting vulnerabilities can be classified into two categories: software vulnerability detection methods based on machine learning (Le et al., 2022; Dam et al., 2018; Wang et al., 2018) or deep learning (Li et al., 2018; Russell et al., 2018; Li et al., 2021b; Wu et al., 2022; Dam et al., 2017). Software vulnerability detection methods based on traditional machine learning rely on rules manually designed by experts. However, excessive dependence on expert experience in rule formulation often leads to false positives (Wen et al., 2023). On the other hand, software vulnerability detection methods based on deep learning (DL) (Zhou et al., 2019; Chakraborty et al., 2021; Li et al., 2021a, 2018) are capable of detecting vulnerabilities by automatically learning the features of vulnerable code. These methods can achieve higher performance in vulnerability detection without requiring human inspection.

Various detecting vulnerability methods have recently been proposed, benefiting from the advancements in DL techniques. Compared to traditional methods, DL-based methods utilize complex Neural Networks (NNs) to automatically learn features of vulnerability from

☆ Editor: Alexander Serebrenik.

* Corresponding authors.
E-mail addresses: Zilongren23@gmail.com (Z. Ren), Ju.xl@ntu.edu.cn (X. Ju), Xchencs@ntu.edu.cn (X. Chen), Quyubin@hotmail.com (Y. Qu).

https://doi.org/10.1016/j.jss.2025.112442
Received 15 October 2024; Received in revised form 18 February 2025; Accepted 16 March 2025
Available online 27 March 2025

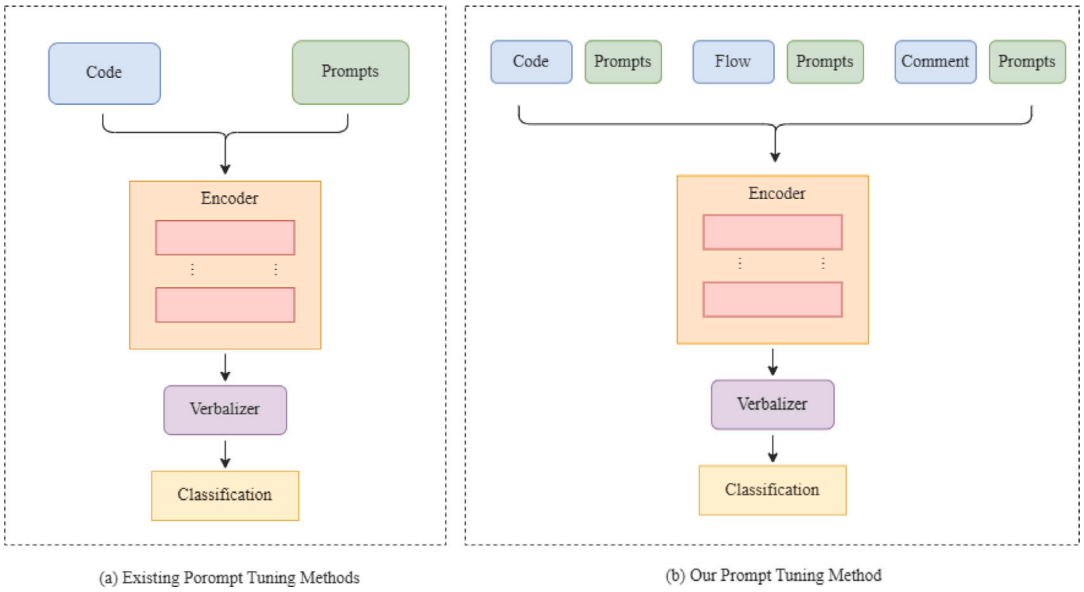

Fig. 1. The prompt tuning processing between MIVDL and existing methods.

known vulnerabilities to detect software vulnerabilities. The DL-based methods typically process the source code into token sequences (Li et al., 2021b; Wu et al., 2022; Hin et al., 2022) or graph structure representations (Zhou et al., 2019; Cao et al., 2022; Wen et al., 2023). For example, VulDeePecker (Li et al., 2018) was based on program slicing, which segments source code according to library or API calls and feeds the segmented code into Recurrent Neural Networks (RNNs) to detect vulnerabilities. Additionally, ReVeal (Chakraborty et al., 2021) utilizes code property graph (CPG) and employs Graph Attention Networks (GAT) to learn graph-based structural properties of code snippets. LineVul (Fu and Tantithamthavorn, 2022) demonstrates better performance using the CodeBERT (Feng et al., 2020).

Though these DL-based methods have demonstrated promising performance, we have identified certain limitations in these methods. **The first limitation is existing works have not fully utilized this modality information.** Specifically, graphs provide structural information, and code offers semantic information. Most existing work only used one of the graphs and code. Therefore, we use graphs, code, and code comments in our approach. Furthermore, previous works have proposed prompt information for LLMs (Wang et al., 2022; Yu et al., 2023; Nie et al., 2022), as shown in Fig. 1 (a). These works only input one type of information and construct context prompts. Our motivation is inspired by the multi-modal framework of GraphCodeBERT, which simultaneously integrates graph structures and other forms of information during the training process. We propose multi-modal prompt tuning to better adapt to downstream tasks (Schick and Schütze, 2021; Wang et al., 2022), as shown in Fig. 1 (b).

The second limitation is that existing vulnerability detection methods (Li et al., 2021b; Cao et al., 2022) **cannot address the limitation of information security.** We simulate a real-world scenario where a company possesses a vulnerability detection tool that needs to be updated and retrained but lacks sufficient training data. To address this limitation, they wish to use source code from other companies to expand their training datasets. However, the source code is commercial confidentiality and cannot be disclosed. This scenario highlights existing limitations in current methods. Peters et al. (2015) proposed LACE2, which reduces the amount of shared data by using multi-party data sharing, demonstrating the effectiveness of this method. However, we observe that privacy concerns cannot be fully alleviated because sharing data among multiple participants is inevitable. Recently, federated learning (FL) has been proposed as a distributed machine learning method. Federated learning is an emerging machine learning paradigm that allows multiple parties to collaboratively train models without sharing the raw data. Unlike centralized machine learning, federated learning distributes the model training process to the participating parties, where each party only needs to train its own local model and then upload the model parameters to a central coordinator. The central coordinator aggregates the collected model parameters, generates a global model, and then distributes it back to the participating parties. This distributed training approach not only protects privacy but also improves the generalization performance of the model. Distributed learning has a wide range of application prospects in fields such as healthcare and finance (Xu et al., 2021), where privacy and security are highly important.

To this end, we propose MIVDL to improve vulnerability detection while protecting user privacy and data security. Firstly, we apply Joern (Yamaguchi et al., 2014) to parse the source code to CPG (see Section 3.1) and then extract graph structure information. Simultaneously, we extract code and comment from the function (see Section 3.2). Next, we combine three modality information and input them into GraphCodeBERT (Guo et al., 2020). To efficiently leverage prompt knowledge in LLMs, we incorporate prompt engineering among information types to enable the model to recognize the input content quickly (see Section 3.3). We then input this into GraphCodeBERT to extract features related to the source code and construct an MLM head for classification. Finally, we constructed a federated learning model that combines features from multi-party data to enhance the performance of the vulnerability detection model (see Section 3.3). Our method utilizes different datasets from various sources to ensure a more comprehensive understanding of vulnerabilities and maintains data privacy by keeping the data decentralized.

To evaluate the effectiveness of MIVDL, we employed three widely used datasets for vulnerability detection: Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021), and Big-Vul (Fan et al., 2020). We conducted a comparative experiment between MIVDL and eight existing software vulnerability detection methods, namely SySeVR (Li et al., 2021b), VulDeePecker (Li et al., 2018), Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021), CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), LineVul (Fu and Tantithamthavorn, 2022), RoBERTa-M (Do et al., 2024) The experimental results on three datasets indicate that MIVDL can improve F1 scores by 5.56%, 5.83%, and 3.04%, respectively. In summary, the main contributions of this paper are as follows.

- We propose MIVDL, which combines multiple information modalities with prompt tuning and uses federated learning to protect data privacy and enhance vulnerability detection.
- We implement three information modalities, namely graphic structure, code semantics, and annotations, along with the method of prompt tuning.
- From the perspective of data privacy protection, we propose a vulnerability detection method based on horizontal joint learning.
- For the convenience of reproduction, we publicly share the code.[1]

2. Background and related work

2.1. Vulnerability detection

Automated software vulnerability detection is crucial for ensuring software security by identifying vulnerabilities and weaknesses in software products. Currently, mainstream vulnerability detection methods rely not on program analysis methods (Cherem et al., 2007; Fan et al., 2019; Kroening and Tautschnig, 2014; Heine and Lam, 2006) but on learning based on source code representations. These methods can be classified into two types: sequence-based methods (Fu and Tantithamthavorn, 2022; Li et al., 2021b; Kamiya et al., 2002) and graph-based methods (Hin et al., 2022; Wen et al., 2023).

Many program analysis methods or traditional vulnerability detection systems are developed based on machine learning or similarity-based approaches (Li et al., 2021b, 2018). These methods require humans to formulate detection rules (e.g., data flow analysis, abstract interpretation, and taint analysis) (Cheng et al., 2024). Although these methods have proven effective in discovering vulnerabilities in software, they rely on rules formulated by humans. If the individuals formulating the rules lack sufficient experience, the rules may struggle to cover various vulnerabilities (Ren et al., 2024).

Addressing the above issues, learning based on source code representations offers a practical approach. Sequence-based vulnerability detection methods convert source code into token sequences. For example, SySeVR (Li et al., 2021b) uses a Bi-LSTM network to learn traversed AST node information. Similarly, VulDeePecker (Li et al., 2018) employs a bidirectional Bi-LSTM network for fine-grained vulnerability detection. LineVul (Fu and Tantithamthavorn, 2022) leverages pre-trained models to evaluate the impact of each input token on detection results, enabling line-level vulnerability detection.

Graph-based methods for detecting vulnerabilities in source code represent the code as graphs. For instance, Devign (Zhou et al., 2019) transforms source code into Code Property Graphs (CPGs) and utilizes Graph Neural Networks (GNNs) to learn features for classification tasks. Similarly, Reveal (Chakraborty et al., 2021) also uses CPGs and applies GGNNs (Groh et al., 2022) to learn features for vulnerability detection. Another approach, AMPLE (Wen et al., 2023) constructs a code structure graph, simplifies it, and uses Graph Convolutional Networks (GCNs) to learn features for vulnerability detection tasks.

In this study, we examined three types of information: graphs that offer structural details, source code that provides semantic context, and comments that offer supplementary information. Our experimental results confirmed the efficacy of our approach.

2.2. Prompt tuning

Prompt tuning was first introduced for adapting large pre-trained language models in NLP (Jia et al., 2021; Lester et al., 2021). Since various NLP tasks can be framed as "text-to-text" problems, specialized prompts guide the language model to answer specific questions (Wang et al., 2022). However, manually creating prompts is challenging and often not optimal. Recently, automatic prompt generation (Li et al., 2023; Li and Liang, 2021)has become a promising method for effectively adapting language models.

In software engineering, the pioneering work uses manually designed contexts to enable models to utilize downstream tasks better or employ prompt tuning to generate appropriate prompts more aligned with task contexts, thereby improving model performance. For example, Li et al. (2023) combined vulnerability code descriptions with just-in-time adjustments for vulnerability assessment while evaluating different types of contexts. Yu et al. (2023) proposed a smart contract slicing method to reduce irrelevant code and combined sliced code with just-in-time tuning. Ruan et al. (2023)utilized prompt tuning to modify original English description inputs to generate the required vulnerability exploits automatically. However, these studies combined a single information modality with prompt tuning. Our work identified multiple information modalities relevant to vulnerability detection, so we integrated them with prompt tuning.

2.3. Distributed learning

Distributed learning aims to train high-performance, centralized models while preserving the privacy of the distributed training data (Konecnỳ et al., 2016). Distributed learning can address issues such as data leakage and data island through coordination among multiple clients under a global aggregator. The two main concepts of distributed learning are local computation and transmission model (Yin et al., 2021). Local computation reduces some of the systemic privacy risks and costs associated with traditional centralized machine learning methods. Model transmission addresses data leakage and data silo issues by training confidential data stored on localized devices locally and then uploading the model to a global aggregator. The global aggregator distributes the updated model back to the localized devices, allowing learning from global data and ultimately achieving learning objectives without exchanging raw data.

Distributed learning integrates privacy protection mechanisms that prevent privacy leakage (Zhang et al., 2021). The emergence of distributed learning has opened up new directions for current research. For example, CPDP (Yamamoto et al., 2023) uses logistic regression and distributed learning for software defect prediction, demonstrating its effectiveness with 25 projects. VFBFL (Zhang et al., 2024) combines vulnerability information with distributed learning for software vulnerability detection, addressing code island problems.

In our study, we propose a novel distributed learning-based vulnerability detection method where we integrate source code, graphs, comments, and prompt information to extract more comprehensive vulnerability feature information. Compared to VFBFL, we employed more advanced techniques and evaluated MIVDL using a more comprehensive dataset (Fan et al., 2020), resulting in improved performance.

2.4. Large language models

The large language models (LLMs) have succeeded significantly in the NLP. LLMs are models trained on massive corpora with billions of parameters. These models exhibit exceptional comprehension abilities, capture knowledge from diverse domains, and are easily applicable to downstream tasks. Deployable LLMs are typically based on the Transformer architecture and come in three main types: encoder-only models, decoder-only models, and models with both encoder and decoder components. Different types of LLMs employ distinct training methods, influencing their suitability for different downstream tasks. For example, encoder-only models (CodeBERT Feng et al., 2020, GraphCodeBERT Guo et al., 2020) are more suitable for classification tasks, decoder-only models (GPT Liu et al., 2023) excel in generation tasks, and models with both encoder and decoder components (CodeT5 Wang et al., 2021) are often used for generation tasks. After training on large-scale data to establish a general model, these LLMs are fine-tuned or prompted to tailor them for downstream tasks, thereby achieving specific objectives.

[1] https://github.com/ntu-juking/MIVDL

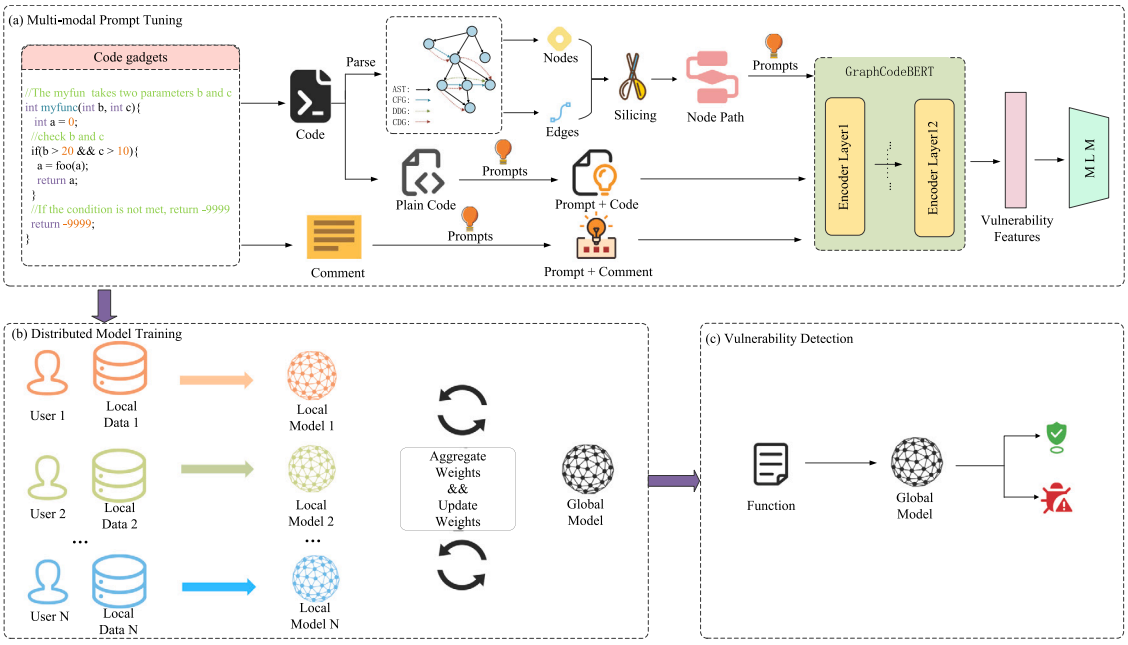

Fig. 2. The processing framework of MIVDL.

3. Approach

This section introduces the phases of MIVDL, which integrates distributed learning with vulnerability detection and provides guidance on implementing multi-modal prompting. Fig. 2 illustrates the overall workflow of MIVDL, and it contains three main phases: The first phase is graph structure information extraction, where we slice nodes and extract paths to remove useless information and obtain effective paths. Additionally, separate the code and comments for extraction and combine prompt information for the path, code, and comments. Input these pieces of information into the model for classification. The second phase involves continually updating the model through distributed learning. The third phase uses a trained model for vulnerability detection. The following subsections detail the specifics of MIVDL.

3.1. Multi-modal prompt tuning

To effectively extract features for the vulnerability detection task, we explored the potential of multi-modal prompts. Previous methods focused on inputting only one type of information and corresponding prompts. Our approach emphasizes the importance of multi-modal prompt methods. As shown in Fig. 1 (b), we introduce three information modalities: (1) source code, (2) paths extracted from the source code, and (3) comments related to the source code.

3.1.1. Graph structure information extraction

The goal of this phase is to obtain effective paths. First, we use Joern to parse the source code to a CPG, which is a data structure integrating AST (Zhang et al., 2019), CFG (Allen, 1970), and PDG (Ferrante et al., 1987). Each type of graph contributes unique informational content to the CPG (Yamaguchi et al., 2014). The AST provides property code representations for each node, reflecting structured information with assigned attribute orders. The CFG provides the property values STMT and PRED for each node and adds label information to each edge to provide control flow information. The PDG establishes control dependencies, including control and data dependencies. Each control dependency in the PDG is assigned an attribute condition indicating the truth value of the original predicate.

During the source code parsing process, we identify variables that have been defined but remain unused. As shown in Fig. 3, we used Joern to parse the source code, generating node and edge information. The variables marked in red on the left side of the Figure are not designed but are used within the function. For this variable, we observed on the right side of the Figure that the generated node information includes only three relevant nodes: parameter, parameter definition, and identifier. The edge information contains only two edges: 19 to 20 and 19 to 50, which correspond to keys in the node information. These three nodes form two essentially meaningless paths. Therefore, we removed these nodes and paths.

The CPG is formally represented as $G = (V, E)$, where V represents node information and E represents edge information. We extract multiple paths based on the captured edge and node information. Fig. 4 illustrates an example of a vulnerable code that was triggered, in which the vulnerability occurs due to accessing an array address beyond its designed maximum capacity. Initially, we convert the source code into a graph by mapping statements to AST nodes. Next, we construct paths based on the control and dependency information between nodes. The resulting paths are depicted in the right half of Fig. 4. The obtained paths are represented by $G' = (V, E')$, where V is the set of nodes, and $E' = \{e_1, e_1, \ldots, e_k\}$ represents the sequence of nodes in the path.

3.1.2. Code semantic and comments extraction

Although path information provides structural details of the source code, they may lack programming logic (Wang et al., 2023). To address this limitation, we adopted a multi-modal approach. In addition to the structural information of the code, the semantic information is equally important. Therefore, we leveraged positional encoding from pre-trained models to extract semantic information from the source code. Furthermore, during our analysis of the dataset, we found that there were a few comments in the code. We hypothesized that comment information might impact the performance of the vulnerability detection task, so we included comments as another information modality and investigated their effectiveness in our research question (see Section 5.3). Fig. 2 illustrates how code and comments were integrated within the dataset. We used regular expressions to separate them, obtaining distinct files for the source code and comments. In the dataset, some source code contains detailed contextual comments that explain the overall function and provide in-depth explanations of specific statements or code blocks, aiding the understanding of complex implementation logic. However, a significant portion of the source code

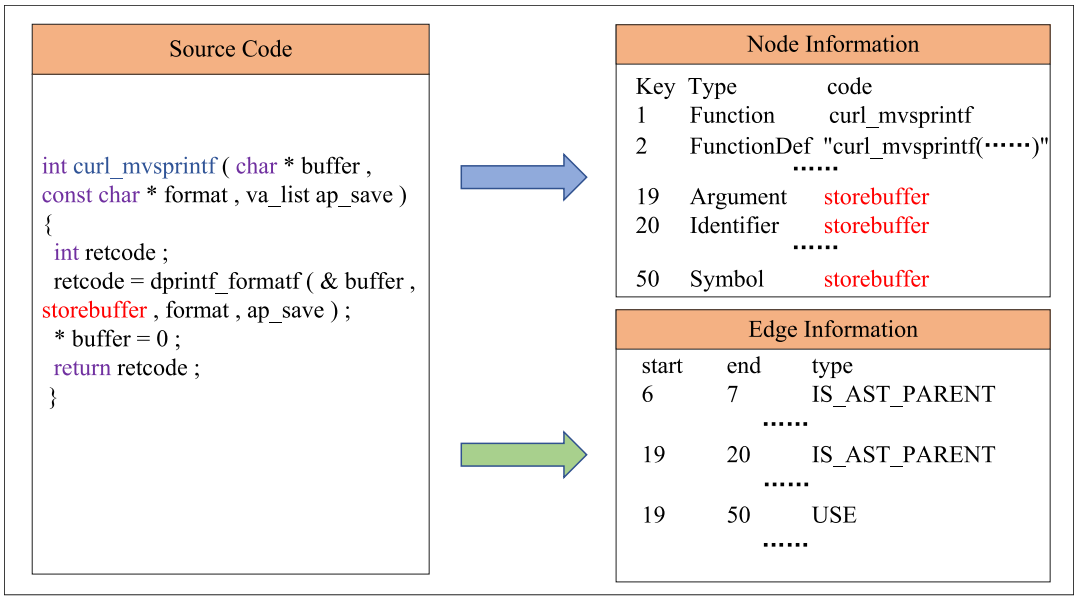

Fig. 3. Illustrating the source code parsing process using an example.

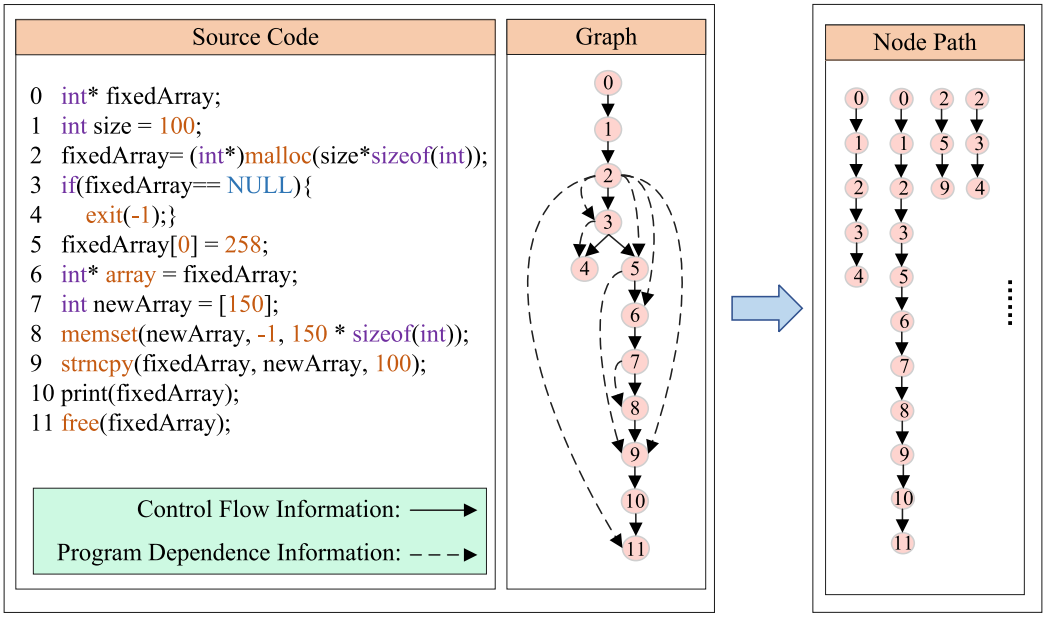

Fig. 4. Illustrating the path extraction process using an example.

lacks detailed explanations of the internal statements, with comments limited to simple descriptions of the functions and missing a deeper analysis of the implementation process. Due to the substantial amount of source code lacking detailed comments, handling the source code and comments separately is necessary.

3.1.3. Prompt template construction

According to the types of prompt tokens, prompt templates can be classified into three types: hard prompt, soft prompt, and hybrid prompt.

Hard prompt indicate tasks by constructing task-related prompts. The expertise of domain specialists informs the design of these prompts. Consequently, the hard prompts tailored for the vulnerability detection task are structured as follows.

$$f_{hard} = \text{Does this code snippet [X] contain a vulnerability?[Z]} \quad (1)$$

where, [X] will be filled with the code snippet, and [Z] is the predicted answer.

Soft prompt contains prompts that are not human-readable and are continuously adjusted during the training process of downstream tasks. Therefore, the soft prompts for the vulnerability detection task are designed as follows.

$$f_{soft} = \text{[SOFT] [X] [SOFT] [Z]} \quad (2)$$

where, [SOFT] represent prompts that are not human-readable.

Hybrid prompt combines hard prompts and soft prompts. Specifically, the prompts in hard prompts typically consist of important keywords directly related to the task and are not allowed to be changed during training. The prompts in soft prompts, however, are continuously modified during training and are not as critical. Therefore, the hybrid prompts for the vulnerability detection task are designed as

follows.

$$f_{hybrid} = \text{[SOFT] code snippet [X] contain a vulnerability [SOFT] [Z]} \tag{3}$$

Existing research has demonstrated that hybrid prompt perform better than the other two types of prompt (Li et al., 2023; Ren et al., 2024). Therefore, we directly adopted the construction form of hybrid templates in our approach. We input the three types of information and their corresponding prompts into GraphCodeBERT. First, in Section 3.1, we obtain the processed paths G', for which we construct prompt information $P(G) = \{pg_1, pg_2, \ldots, pg_l\}$. Next, in Section 3.2, we extract the code and comments and flatten them to obtain code $C = \{c_1, c_2, \ldots, c_n\}$ and comments $M = \{m_1, m_2, \ldots, m_j\}$. The corresponding prompt information for the code is $P(C) = \{pc_1, pc_2, \ldots, pc_m\}$, and for the comments $P(M) = \{pm_1, pm_2, \ldots, pm_z\}$. To ensure compatibility and consistency with the model input, we chose to use the tokenizer of PLM. We concatenate the prompt information and the three information modalities into a sequence input $I = \{[CLS], P(C), [SEP], C, [SEP], P(G), [SEP], V, [SEP], P(M), [SEP], M\}$, where [CLS] is a special symbol indicating the start of the sequence, and [SEP] is a special symbol used to separate different data segments. GraphCodeBERT takes the sequence I as input and converts it into vectors. For each token, its input vector is constructed by summing the corresponding token and position embeddings. The model applies n Transformer layers to the input vectors to generate contextual representations. Each Transformer layer contains the same architecture with multi-head self-attention operations. Finally, through the model's input, we obtain the feature vectors.

3.2. Distributed model training

From Fig. 2, we observe the model setup for distributed learning, which involves multiple users with their local data. We train a personalized model by combining multiple users. In our approach, we utilize horizontal distributed learning, where the model consists of multiple local models and one global model. The network architecture of both the global and local models is identical. We employ feature vectors to learn, with the final layer used for classifying and detecting vulnerabilities. To ensure data privacy protection, we employ data leakage through differential privacy techniques (El Ouadrhiri and Abdelhadi, 2022). Specifically, no raw data is shared between participants; instead, only the model weights are shared with the central server during the model update phase. Furthermore, differential privacy is applied by adding noise during the aggregation phase. This approach safeguards against the inference of individual data from model updates and prevents the model from memorizing sensitive personal information during training. Next, we will describe how distributed learning and vulnerability detection are integrated.

Algorithm 1 A model's local training procedure at round t

Require: D_i, θ^{t-1}_{global}, γ, Epoch E, local learning rate lr, Batch b
Ensure: Local model's weights θ^t
  Initialize $\theta^t \leftarrow \theta^{t-1}_{global}$
  for each $e \in [1, E]$ **do**
    if $e == 1$ **then**
      Initialize θ^0
    else
      $\theta^t \leftarrow \theta^{t-1}_{global}$
    end if
    for $b \in D_{local}$ **do**
      $\theta^t \leftarrow \theta^t - lr \cdot \nabla \gamma(\theta^t, b)$
    end for
  end for
  return θ^t

First, we construct both local models and a global model. Local models are responsible for training on local data, while the global model interacts with these local models. Second, when a local model has not received additional data from the global model, it initially trains its own model locally and updates its weights. Third, local models interact with the global model by sharing updates to refine the global model's weights. During the process of updating model weights, we employed differential privacy, as outlined in Wei et al. (2020). Specifically, noise was introduced during the gradient updates phase, which effectively mitigates the risk of privacy leakage during data-sharing operations. Fourth, after the global model is updated, it distributes its parameters to each local model, updating their weights. Fifth, after completing one round of updates through steps two to four, the entire distributed learning process iterates a specified number of times to complete learning of the global model. Finally, the trained global model is used to predict vulnerabilities. Additionally, local models can also perform predictions locally.

Specifically, we train four local models using the datasets Reveal, Devign, and two subsets of Big-Vul. The Big-Vul dataset, due to its large size, is divided into two subsets through a random uniform sampling method, ensuring that the original distribution of vulnerabilities is preserved in both subsets. This guarantees an equal proportion of vulnerable and non-vulnerable samples across the two subsets, maintaining consistency with the original dataset. In Step 1, the linear layer weights in all models are initialized to 0. Step 2 involves local models starting training until each local model's loss converges, resulting in weights W_1^0, W_2^0, W_3^0, W_4^0, and feature vectors n_1, n_2, n_3, n_4. Step 3 involves the central model receiving the feature vectors n_1, n_2, n_3, n_4, calculating and aggregating weights W_1^0, W_2^0, W_3^0, W_4^0 to obtain W^1. Step 4 distributes the W^1 weights to each client, which then accepts the weights. This process constitutes one communication round, which repeats steps 2, 3, and 4 until all communication rounds are completed.

Formally, n local databases correspond to n local models, undergoing m communication rounds to contribute to the training. Each local model i has a corresponding local dataset D_i. Each model is assigned initial parameters θ in the initial phase. After that, at the start of each communication round, the global model distributes the previous round's parameters θ^{t-1}_{global} to the local models. Upon receiving these parameters, the local models update their parameters with θ^{t-1}_{global} and continue training on D_i, producing local model parameters θ^t_i, as described in Algorithm 1. Subsequently, each local model sends the difference $(\theta^t_i - \theta^{t-1}_{global})$ back to the global model. The server aggregates these differences from all local models using the FedAvg algorithm (McMahan et al., 2017) to update the global model parameters to θ^t_{global} as follows:

$$\theta^t_{global} = \theta^{t-1}_{global} + \sum_{i=1}^{m} \frac{n_i}{n_t}(\theta^t_i - \theta^{t-1}_{global}) \tag{4}$$

where n_i is the number of samples from the local model i, and n_t is the total number of samples from the selected clients in round t. In our approach, all models participate in the parameter updates, and all models are selected each time, so the formula is as follows:

$$\theta^t_{global} = \theta^{t-1}_{global} + \sum_{i=1}^{m}(\theta^t_i - \theta^{t-1}_{global}) \tag{5}$$

3.3. Vulnerability detection

During the detection phase, we apply the model trained during the distributed learning phase to detect potential vulnerabilities in the code. Specifically, in the initial phase, the graphical representation of the source code, the semantic representation of the source code itself, and accompanying comments are combined to capture latent information through LLM. To enhance the model's ability to comprehend input content effectively, we designed distinct prompt structures tailored to the characteristics of various input types. Next, the LLM encoder embeds all information into low-dimensional vectors, and the MLM head learns and predicts information. Subsequently, we create a

local model for each dataset and simultaneously create a global one. The global model and local models update each other, resulting in a trained model. Finally, all information is fed into the trained detection model for vulnerability detection.

4. Experimental setup

4.1. Research questions

To evaluate MIVDL, we aim to answer the following three research questions:

RQ1: How does MIVDL compare to state-of-the-art function-level vulnerability detection methods?

To answer this question, we want to compare MIVDL with state-of-the-art function-level vulnerability detection methods (such as SySeVR Li et al., 2021b, VulDeePecker Li et al., 2018, Devign Zhou et al., 2019, Reveal Chakraborty et al., 2021, CodeBERT Feng et al., 2020, CodeT5 Wang et al., 2021, LineVul Fu and Tantithamthavorn, 2022), to evaluate its performance in terms of accuracy, efficiency, and applicability across various types of vulnerabilities.

RQ2: How does the impact of different information modalities for MIVDL?

To answer this question, we want to investigate how various information modalities, such as code semantics, graph, and code comment, influence the performance of MIVFL in detecting vulnerabilities. By analyzing the impact of each modality, we aim to understand which aspects of the information contribute most to the accuracy and effectiveness of MIVFL, ultimately guiding the optimization of the model for better vulnerability detection.

RQ3: How does the impact of distributed learning for MIVDL?

To answer this question, we want to explore the impact of distributed learning on MIVDL by evaluating how decentralized training across multiple data sources affects its ability to detect vulnerabilities.

4.2. Datasets

Our study evaluates MIVDL on three widely used datasets: Devign (Zhou et al., 2019), ReVeal (Chakraborty et al., 2021), and Big-Vul (Fan et al., 2020). We show the details of these three datasets as follows.

- **Devign** (Zhou et al., 2019). The Devign dataset consists of functions collected by FFMPeg+Qemu, comprising approximately 10k vulnerable functions and about 12k non-vulnerable functions. Devign is a balanced dataset.
- **ReVeal** (Fan et al., 2020). ReVeal collected from Linux Debian Kernel and Chromium, contains 1.6k vulnerable functions and approximately 16k non-vulnerable functions. ReVeal is an imbalanced dataset.
- **Big-Vul** (Chakraborty et al., 2022). Big-Vul, collected by Fan et al. comprises 10k vulnerable functions and about 168k non-vulnerable functions. Big-Vul is also an imbalanced dataset.

Table 1 presents the details of these three datasets, including the total number of samples, the number of vulnerable samples (#Vul), the number of non-vulnerable samples (#Non-vul), and the ratio of vulnerabilities (Vul Ratio).

Table 1
Statistics of the datasets.

Dataset	#Samples	#Vul	#Non-vul	Vul ratio(%)
Devign	22,361	10,067	12,294	45.02
ReVeal	18,169	1,664	16,505	9.16
Big-Vul	179,299	10,547	168,752	5.88

4.3. Performance metrics

To evaluate the effectiveness of our method, we employed the following four widely used evaluation metrics to evaluate MIVDL:

TP: True Positive (TP) denotes the count of instances where the model accurately predicts samples of the positive class. In the context of vulnerability detection, TP represents the cases where the model successfully identifies code with vulnerabilities.

TN: True Negative (TN) represents the count of instances where the model correctly identifies samples as belonging to the negative class. In vulnerability detection, TN refers to the cases where the model accurately assesses that the code is free from vulnerabilities.

FN: False Negative (FN) represents when the model mistakenly classifies positive samples as negative. In the context of vulnerability detection, FN represents instances where the model fails to identify genuine vulnerabilities.

FP: False Positive (FP) represents when the model erroneously classifies negative samples as positive. In vulnerability detection, FP represents situations where the model incorrectly identifies code without vulnerabilities as having vulnerabilities.

Accuracy: Accuracy refers to the proportion of correctly predicted or identified vulnerabilities relative to the total number of vulnerabilities present. It is calculated as $\frac{TP+TN}{TP+TN+FN+FP}$.

Precision: Precision assesses the proportion of relevant vulnerabilities among those retrieved. It is calculated as $\frac{TP}{TP+FP}$.

Recall: Recall evaluates the proportion of retrieved relevant vulnerabilities. It is calculated as $\frac{TP}{TP+FN}$.

F1 Score: The F1 score, representing a balance between precision and recall, is the harmonic mean of precision and recall. It is calculated as $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision}+\text{Recall}}$.

4.4. Baseline methods

We compared eight baseline methods, including token-based, graph-based, and LLM-based vulnerability detection methods, as shown below:

- SySeVR (Li et al., 2021b): SySeVR employs a bidirectional recursive neural network within its vulnerability framework. It extracts both syntax and semantic features from the code to enhance vulnerability detection.
- VulDeePecker (Li et al., 2018): VulDeePecker converts code into an intermediary form containing semantic information like data and control dependencies. This intermediate representation is converted into vectors for input into a bidirectional LSTM-based neural network for detecting vulnerabilities.
- Devign (Zhou et al., 2019): Devign is a graph-based method that encodes function source code into a unified graph structure with comprehensive program semantics using graph embedding layers. It then utilizes Gated Recurrent Unit (GRU) layers to learn features of nodes within the graph, followed by a Convolutional (Conv) module to extract node representations for graph-level predictions.
- Reveal (Chakraborty et al., 2021): ReVeal employs Graph Attention Networks (GAT) to learn the structural properties of code snippets. GAT utilizes gated graph neural networks, resampling techniques, and triplet loss to learn the structural properties of code snippets.
- CodeBERT (Feng et al., 2020): CodeBERT is an LLM based on BERT and was developed for six programming languages: Java, Python, JavaScript, PHP, Ruby, and Go. This model utilized a masked language approach and incorporated token replacement detection objectives.

Table 2
Comparison results between MIVDL and baselines on the three datasets in vulnerability detection. The best results for each metric are highlighted in bold.

Metrics(%) \ Dataset	Devign				Reveal				Big-Vul			
Baseline	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
SySeVR	48.59	47.08	60.02	52.77	73.21	43.56	27.84	33.97	90.10	30.91	14.08	19.34
VulDeePecker	50.12	47.89	33.34	39.31	78.51	20.63	14.59	17.09	81.19	38.44	12.75	19.15
Devign	57.19	52.41	58.11	55.11	86.38	28.98	34.73	31.60	56.25	50.84	70.79	59.24
Reveal	62.73	53.94	71.22	61.39	85.25	29.73	**64.96**	40.79	53.45	48.01	70.07	56.98
CodeBERT	59.77	57.98	56.59	57.28	91.09	**59.38**	31.67	41.30	64.25	56.74	54.58	55.64
CodeT5	46.59	58.83	54.96	56.83	90.49	45.57	35.14	39.68	62.14	60.15	59.39	59.77
LineVul	62.75	**63.98**	50.51	56.45	92.43	48.86	41.35	44.79	95.82	90.68	83.31	86.84
RoBERTa-M	54.43	43.81	77.93	56.09	91.78	43.43	39.74	41.50	88.14	55.45	70.24	61.97
MIVDL	**64.21**	55.70	**83.55**	**66.84**	**92.91**	57.86	45.00	**50.62**	**98.78**	**91.34**	**88.47**	**89.88**

- CodeT5 (Wang et al., 2021): CodeT5 is a pre-trained language model designed for code generation and understanding, aiming to improve code-related tasks such as code generation, code completion, code translation, etc.
- LineVul (Fu and Tantithamthavorn, 2022): LineVul utilizes a transformer architecture to address three limitations in the IVDetect method and performs vulnerability detection at both the line and function levels.
- RoBERTa-M (Do et al., 2024): RoBERTa-M uses the RoBERTa model to extract features from source code. These features are then processed using supervised machine learning algorithms for classification tasks.

To ensure accuracy and fairness in our experiments, we adhered to the hyperparameters and dataset split specified in the original baseline papers. For Devign, since the code was unavailable, we replicated the experiments by following the methodology considered by ReVeal.

4.5. Experiment settings

To ensure a fair comparison, we follow the experimental settings of the baselines, such as Devign, Reveal, and Big-Vul. We used the same hyperparameters as detailed in the referenced work. Consistent with prior research (Wen et al., 2023; Ren et al., 2024; Wu et al., 2022), we employed stratified sampling to partition the dataset into disjoint training, validation, and test sets, adhering to a split ratio of 8:1:1. All baselines and our model used the same dataset partition. For Devign, where code was not provided, we replicated the experiments based on the methods described in ReVeal.

Our experimental setup employed GraphCodeBERT as our pre-trained model with a maximum input length of 514, a learning rate of 2e-5, and a batch size of 16. For graph extraction, we used the parsing tool Joern. During the distributed learning phase, we used seven communication rounds. We trained the model using an NVIDIA GeForce RTX 4090, with a maximum of 100 iterations and early stopping patience set to 50 epochs.

5. Experimental results

5.1. RQ1. Effectiveness of MIVDL

To address this research problem, we compared eight methods: graph-based, sequence-based, and LLM-based. As illustrated in Table 2, MIVDL demonstrates superior performance in terms of F1 scores and accuracy across three datasets when compared to the other models. Specifically, MIVDL improves the F1 scores by 5.45%,5.83%, and 3.04% compared to the current best baseline models, respectively. The accuracy scores increased by 1.46%, 0.48%, and 2.96%, respectively.

In imbalanced datasets, the internal class distribution imbalance renders accuracy an insufficient evaluation metric. As a more comprehensive measure, F1 score provides a more meaningful evaluation. For Devign, all metrics appear consistent, whereas for Reveal, metrics other than accuracy also remain stable. However, the remarkable performance observed on the Big-Vul dataset suggests a potential explanation: Big-Vul originates from real-world vulnerabilities, which may overlap with the corpus used for pre-training the model. This overlap could contribute to the observed performance improvements.

Previous methods leveraging LLM-based approaches typically treat source code as natural language sequences, enabling the model to learn only semantic and syntactic information. In contrast, our approach captures semantic and syntactic information and integrates graph attributes and annotation data, thereby providing a more comprehensive representation. The results demonstrate that our approach outperforms methods solely on semantic and syntactic information, such as LineVul.

Furthermore, even the use of foundational LLMs (e.g., CodeBERT, CodeT5) has achieved high F1 scores, and the performance of LLM-based detection methods (e.g., RoBERTa-M, LineVul) indicates that superior results can be obtained. Notably, LineVul and RoBERTa-M have already outperformed all existing program analysis-based deep learning methods without employing program analysis. This highlights the potential of LLM-based vulnerability detection methods to achieve state-of-the-art performance. Overall, by integrating multiple types of information, our approach achieves better performance compared to purely LLM-based models. In other words, the MIVDL framework surpasses existing graph-based, token-based, and LLM-based methods in vulnerability detection.

Integrating multiple modalities with rapid fine-tuning introduces certain complexities. To mitigate these challenges, we minimized task-irrelevant factors as much as possible, as outlined in Sections 3.1.1 and 3.1.2. Furthermore, given the extensive number of parameters in the pre-trained model, we froze a subset of them to reduce overall complexity. Therefore, the multi-modal information prompting method can further enhance the performance of LLMs. Additionally, we discuss the impact of information modalities on MIVDL in Section 5.3.

> **Summary for RQ1:** The F1 score of MIVFL outperformed all baselines on three datasets. Specifically, the MIVDL achieves improvements of 5.45%, 5.83%, and 3.04% in F1 score. The results demonstrate the effectiveness of our method.

5.2. RQ2: Effectiveness of prompt information

To address this research question, we are dedicated to exploring the contribution of information modalities to the MIVDL approach and examining the effectiveness of different information modalities.

Table 3
The impact of different information modalities on the performance of MIVDL.

Method	Devign		Reveal		Big-Vul	
	Recall	F1	Recall	F1	Recall	F1
MIVDL +C	74.33	62.63	41.63	43.79	85.12	86.55
MIVDL +P	81.14	59.76	38.24	40.39	83.75	84.53
MIVDL +C+M	82.66	62.94	41.81	45.54	85.23	86.72
MIVDL +P+M	81.37	61.42	37.27	42.27	82.37	85.14
MIVDL +C+P	83.45	65.93	43.72	49.98	86.74	88.92
MIVDL +C+P+M	**83.55**	**66.84**	**45.00**	**50.62**	**88.47**	**89.88**

Our study uses three information modalities: (1) source code, (2) paths, and (3) comments. We investigate the impact of these three types of information on MIVDL. We denote source code as C, paths as P, and comments as M. For example, MIVDL +C indicates that we only use the source code as the information modality in the model.

According to Table 3, we observe that different information modalities affect the performance of MIVDL differently. Firstly, comparing MIVDL +P and MIVDL +C, we see that MIVDL +C performs the best. Additionally, comparing MIVDL +P+M and MIVDL +C+M, MIVDL +C+M shows better performance. This indicates that the source code has the most significant impact on the performance of MIVDL. This also explains why previous studies (Fu and Tantithamthavorn, 2022) used source code as the information modality. Comparing MIVDL +C+P+M and MIVDL +C+P, we find that comments have a slight impact on the performance of MIVDL. This is because not every function contains comment information, and not all comments are relevant to vulnerabilities. Note Table 3 does not include the case of MIVDL +M because we observed that not all code contains comments. Therefore, comments were not evaluated as a standalone information modality but rather as an adjunct to other modalities. Next, we compared three methods: MIVFL+C, MIVFL+C+P, and MIVFL+C+P+M. The results indicate that as the number of information modalities increases, the model performance improves. This is because different information modalities provide distinct content: source code offers syntax information, path provides structural information, and comments further explain the source code.

Summary for RQ2: The source code has the most significant impact on MIVDL, followed by path, while the influence of comments is relatively minimal. MIVDL, which utilizes three types of information, improved the F1 score by 4.21%, 6.83%, and 3.33% across the three datasets compared to the method that only used source code. Furthermore, recall can be increased by 9.22%, 3.37%, and 3.35%.

5.3. RQ3: Effectiveness of distributed model training

To address this research question, we are dedicated to exploring the contribution of distributed learning to the MIVDL approach. Fig. 5 illustrates the performance of MIVDL with and without using distributed learning. The red bars represent the performance with distributed learning, while the orange bars represent the performance without distributed learning. For methods that do not use distributed learning, we use all data not included in the test setting or validation set as the training set to ensure fairness in data utilization. We evaluated the method on three datasets using four evaluation metrics, and the results show that MIVDL with distributed learning outperforms the method without distributed learning. Specifically, MIVDL with distributed learning achieved similar performance by four metrics (see Section 4.3). Additionally, the use of distributed learning ensures data security.

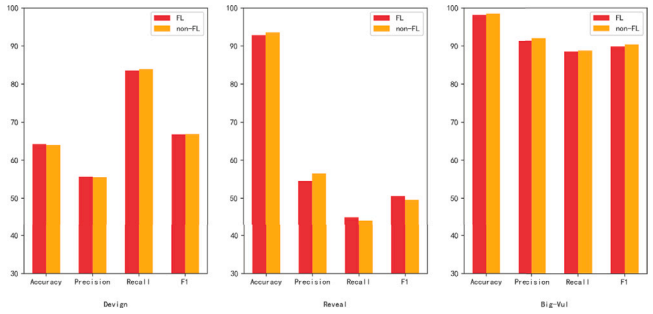

Fig. 5. The impact of distributed learning for MIVDL.

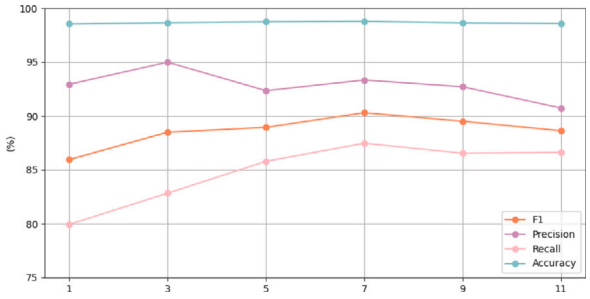

Fig. 6. The impact of different communication rounds on MIVDL performance.

Distributed learning is influenced by various factors, among which uneven data distribution can result in minimal performance improvement. Table 1 shows the three datasets exhibit an uneven distribution of vulnerabilities. Model updates on devices may conflict with each other, leading to unstable convergence of the global model. However, using distributed learning allows each local model to be trained according to its own data, producing a model that better meets its specific needs. When these local models are aggregated, they generate a robust and adaptable global model. Consequently, such a model is well-suited to handle data heterogeneity across different devices and can show better performance. The results indicate that vulnerability detection can be performed with a slight performance improvement.

Summary for RQ3: The performance of using distributed learning is better than that of non-distributed approaches, while also ensuring data privacy.

6. Discussion

6.1. Performance impact of different communication rounds

Our empirical study finds that the number of communications in federated learning might affect the performance of MIVDL. Therefore, we investigated the impact of different communication rounds on model performance. Fig. 6 shows the results of different communication rounds on the validation set. We explored communication rounds 1, 3, 5, 7, 9, and 11. Using 7 communication rounds, all metrics except for the precision score were higher than in other rounds. The precision score did not show a consistent trend across different communication rounds. We speculate that this may be related to the class imbalance issue. The results indicate that F1, Accuracy, and Recall performed best in the seventh communication round. Thus, we selected 7 rounds for MIVDL.

Table 4
The impact of different prompt templates on the performance of MIVDL.

Template		Devign		Reveal		Big-Vul	
		Recall	F1	Recall	F1	Recall	F1
H1	[SOFT] code snippet [X], graph [G], comment [C] contain a vulnerability [SOFT] [Z]	83.55	66.84	45.00	50.62	88.47	89.88
H2	[SOFT] code snippet [X], path [G], comment [C] contain a vulnerability [SOFT] [Z]	82.73	65.20	42.27	49.27	83.75	87.53
H3	[SOFT] code gadget [X], graph [G], comment [C] contain a vulnerability [SOFT] [Z]	83.16	66.58	43.72	48.98	85.23	87.72
H4	[SOFT] code snippet [X], graph [G], annotate [C] contain a vulnerability [SOFT] [Z]	83.21	65.41	43.65	49.56	87.16	88.64
H5	[SOFT] code snippet [X], graph [G], comment [C] contain a bug [SOFT] [Z]	82.28	65.44	43.72	47.12	86.43	87.92
N1	[SOFT] code snippet [X], graph [G], comment [C] is classified into two categories [SOFT] [Z]	71.37	61.42	46.63	44.16	82.37	82.14
N2	[SOFT] text [X], image [G], picture [C] contain a vulnerability [SOFT] [Z]	76.34	62.93	38.62	45.79	83.74	83.59
N3	[SOFT] text [X], image [G] picture [C] is classified into two categories [SOFT] [Z]	64.63	35.38	35.89	43.16	82.95	71.02

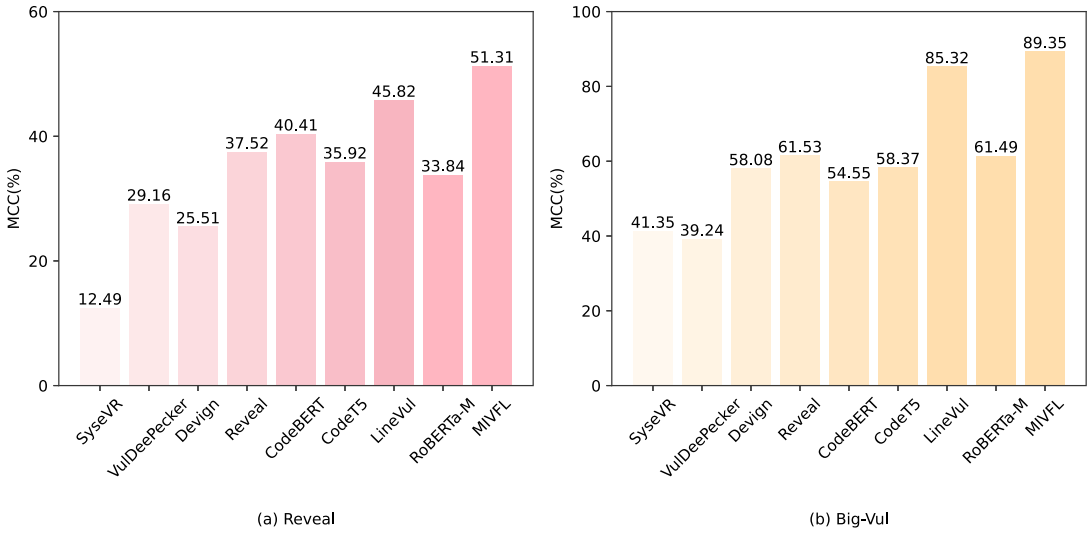

Fig. 7. The comparison results between MIVDL and seven baselines in MCC.

6.2. Performance comparison in terms of MCC

In our evaluation datasets, we discovered an issue of class imbalance (see Section 4.2). Therefore, we used the Matthews Correlation Coefficient (MCC) as a metric further to evaluate the performance of our model (Tanha et al., 2020). MCC is a metric for evaluating the performance of binary classifiers. It accounts for true positives, true negatives, and false positives. MCC ranges from −1 to 1, where 1 indicates perfect prediction, 0 indicates random prediction, and −1 indicates complete mismatch. The specific formula is as follows:

$$\mathrm{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{6}$$

Fig. 7 shows the performance of different methods on two imbalanced datasets. MIVDL achieved the highest MCC scores on two datasets. Specifically, the score on the Big-Vul dataset is close to 1, indicating excellent model predictions. Additionally, the score on the Reveal dataset is 5.49% higher than the best previous score, demonstrating that MIVDL outperforms the baseline.

6.3. Performance impact of different prompt templates

In this subsection, we mainly discuss the impact of keyword semantics in prompt templates. Section 3.1.3 introduces hybrid prompts in the MIVDL, where the templates combine keywords from hard prompts. We explore the influence of different keywords on MIVDL. We did not study the soft prompts in the hybrid templates because this part is learned and explored by the model during downstream tasks.

To investigate the impact of keyword semantics on MIVDL and the specific manifestation of this impact, we designed a set of keywords, as shown in Table 4. First, we set up task-related keywords and input-related prompt words, designing five template s (H1, H2, H3, H4, H5). Next, we designed two more sets of templates: one with keywords unrelated to the task but related to the input, and the other with keywords related to the task but irrelevant to the input (N1, N2). Finally, we designed a set of templates with keywords unrelated to the task and the input (N3).

By comparing task-related keywords and input-related prompt words, we find that setting irrelevant prompts significantly affects the method's performance. Specifically, the performance was the worst in N3, where all keywords were irrelevant. Additionally, comparing N1 and N2 reveals that templates unrelated to the task have a more significant impact. This may be due to the restriction on the model's understanding and reasoning capabilities. Furthermore, we explored the impact of synonymous keywords on the templates. H1 demonstrated the best performance.

We used the GraphCodeBERT model as the base for our training. The common separators are as follows: [SEP], [CLS], [MASK], [PAD], and [UNK]. The specific embedding vectors for these separators do not significantly affect the model's performance. Furthermore, in the [SOFT] section, the inference phase may encounter delimiters that are unknown to us. To mitigate potential performance degradation due to these unknown separators, we conducted multiple repeated experiments.

Table 5
Wilcoxon test between MIVDL and seven baselines on F1 metrics.

Baselines	Devign	Cliff's Delta	Reveal	Cliff's Delta	Big-Vul	Cliff's Delta
SySeVR	**	large	**	large	**	large
VulDeePecker	**	large	**	large	**	large
Devign	**	large	**	large	**	large
Reveal	**	large	**	large	**	large
AMPLE	**	large	**	large	**	large
LineVul	**	large	**	large	**	large
RoBERTa-M	**	large	**	large	**	large

Notes: *** means p-value <0.001, ** means p-value <0.01, * means p-value <0.05.

Table 6
Wilcoxon test across different modalities based on their F1 metrics.

Dataset / Wilcoxon	Devign			Reveal			Big-Vul		
Method	MIVDL +C	MIVDL +P	MIVDL +C+P	MIVDL +C	MIVDL +P	MIVDL +C+P	MIVDL +C	MIVDL +P	MIVDL +C+P
MIVDL +C+M	*	–	–	**	–	–	*	–	–
MIVDL +C+P	**	**	–	**	**	–	**	**	–
MIVDL +P+M	–	**	–	–	**	–	–	*	–
MIVDL +C+P+M	–	–	**	–	–	**	–	–	*

Notes: *** means p-value <0.001, ** means p-value <0.01, * means p-value <0.05, (-) indicates no comparison.

6.4. Statistical analysis on the performance of MIVDL

The Wilcoxon test is often used to determine whether the difference between two means is statistically significant (i.e. when the computed p-value is less than 0.05). We employ the Wilcoxon rank-sum test (Wilcoxon, 1992) to assess whether the superior performance of MIVDL is attributable to chance. The Cliff's Delta value spans from $[-1, 1]$. A value of 0 indicates no difference between the two datasets. The metric reaches a value of 1 when all elements in one dataset exceed those in the other, and -1 when the opposite is true, with all elements in the second dataset exceeding those in the first. When $0.148 \leq ||d|| < 0.33$, the effect size is deemed small; when $0.33 \leq ||d|| < 0.474$, it is considered medium; and when $||d|| \geq 0.474$, it is classified as large. Specifically, we compare the F1 scores of MIVDL with those of the baseline methods to assess the presence of significant differences. Additionally, we investigated whether the integration of different modalities, as presented in Table 3, exhibits statistically significant differences. The comparison results with the baseline are presented in Table 5.

We evaluated and compared the performance of MIVDL against the baseline methods using the same test set. The results indicate that the p-values between MIVDL and all baseline methods are less than 0.01, and the effect size of Cliff's Delta was greater than 0.474 in all instances. This signifies a large effect size, highlighting a statistically significant difference in performance. Furthermore, a similar comparison was conducted to assess the integration of different modalities, with the results presented in Table 6.

Overall, the integration of different modalities also shows significant differences, further validating the effectiveness of our method. However, when performing multiple significance tests on MIVDL with added comments on the Big-Vul dataset, not all tests yielded significant results, suggesting a certain degree of randomness. Upon further analysis, we found that the Big-Vul dataset contains very few comments, which do not substantially enhance performance. In contrast, incorporating comments into MIVDL proved to be effective for the other two datasets.

6.5. Qualitative analysis

We extend our analysis by evaluating MIVDL on real-world datasets. Furthermore, we perform a comparative analysis with LineVul, which leverages large language models (LLMs) but relies solely on the semantic information extracted from source code. For this evaluation, we selected the PreciseBug dataset (He et al., 2023). Rather than employing the entire dataset, we specifically focused on C/C++ samples. We curated 1000 vulnerability-containing samples from this subset based on the 2024 CWE high-risk vulnerability list (MITRE, 2024). The selected types are shown in Table 7.

Table 7
Statistics of the datasets.

Type	Description	LineVul	MIVDL
CWE-787	Out-of-bounds Write	39	45
CWE-20	Improper Input Validation	35	34
CWE-416	Use After Free	32	50
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	22	34
CWE-190	Integer Overflow or Wraparound	42	49
CWE-287	Improper Authentication	45	39
CWE-119	Buffer Overflow	40	45
CWE-200	Exposure of Sensitive	55	51
CWE-476	NULL Pointer Dereference	19	35
CWE-125	Out-of-bounds Read	40	43

For statistical analysis, we selected 10 distinct types of vulnerabilities and assigned 100 vulnerability-containing functions in each kind. This selection enabled a systematic comparison of the performance of MIVDL and LineVul. The final two columns indicate the number of vulnerabilities identified by each method across 100 distinct vulnerable functions. The comparison results are shown in Table 7. The results indicate that MIVDL outperforms LineVul in detection accuracy for multiple vulnerability types. Notably, many of the most critical vulnerabilities, as outlined in the list of the most dangerous software weaknesses (MITRE, 2024), rely on complex conditional statements, loop structures, and other intricate control flow behaviors. Examples of such vulnerabilities include Use-After-Free, Integer Overflow or Wraparound, and Buffer Overflow. Our approach leverages graph structures to analyze complex control flow patterns effectively, achieving superior performance. However, the performance improvement was less pronounced for vulnerabilities such as Improper Authentication and Exposure of Sensitive Information. This may be attributed to the model's inability to fully capture relevant features during the communication process, resulting in some loss of critical information. Nevertheless, the performance of our method on these two vulnerability types is only marginally lower than that of LineVul.

6.6. Training time

We evaluate the time efficiency of various methods, explicitly examining the time required for model convergence during training. To ensure a fair comparison, we standardized key factors, including the

Table 8
The cost of training time in different methods.

Method \ Dataset	Devign		Reveal	
	Train time	Round	Train time	Round
fine-tuning	17358 (s)	100	16485 (s)	100
prompt tuning	3400 (s)	11	3807 (s)	15

training data size, learning rate, batch size, and loss-stopping threshold. The comparison results, which encompass both prompt tuning and regular fine-tuning, are presented in Table 8.

The results presented in Table 8 indicate that prompt tuning requires significantly less time, achieving convergence by the 11th round. In contrast, with fine-tuning, the model fails to reach the convergence threshold even after completing the maximum number of training rounds (i.e., 100 rounds). A comparison of training times highlights that prompt tuning substantially reduces the overall training duration. In summary, prompt learning saves considerable time and underscores our proposed method's efficiency and effectiveness.

6.7. Threats to validity

External threats. The external threats we studied mainly originate from the datasets. The three datasets evaluated by MIVDL are widely used in previous research (Wen et al., 2023; Fu and Tantithamthavorn, 2022). However, these datasets only include C/C++, excluding other programming languages such as Java, PHP, or Python. In the future, we will expand the programming languages in the datasets to evaluate MIVDL.

Internal threats. The internal threats we studied mainly: 1. Modular Design: Rather than fully integrating modalities at all levels, we propose a modular approach wherein each modality is managed independently within specialized modules, thereby preserving distinct boundaries among them. This strategy may uphold interpretability while benefiting from the complementary information each modality provides. 2. Model Simplification: To reduce the complexity introduced by multiple modalities, we employ techniques such as pruning, which can simplify the model without significantly compromising performance, thus improving both scalability and interpretability. 3. Visualization Tools: We may utilize existing visualization tools that facilitate the tracking of information flow across the modalities, thereby empowering users to better understand how the model processes and integrates various types of data. Additionally, MIVDL is controlled by multiple parameters, such as learning rate and optimizer, which may affect the effectiveness of our approach. As the scale of the dataset increases, finding the optimal parameter settings becomes more challenging. However, our research is not aimed at finding the optimal parameter settings. By comparing the performance of MIVDL with the baseline, MIVDL already outperforms the baseline without seeking optimal parameters. Therefore, the performance presented in this paper can be considered a lower bound of the method, with the potential for further improvement through parameter tuning.

Construct threats. The construct threats we studied mainly stem from the selected performance metrics. We used four commonly used performance metrics to evaluate the performance of MIVDL. However, due to the class imbalance in the dataset, we also considered using MCC to ensure the study's rigor.

7. Conclusion

In this work, we proposed a novel vulnerability detector capable of detecting various vulnerabilities. MIVDL first identifies the source code, separating comments from the source code, and then extracts the paths of the code segments. These three types of information are then augmented with prompt information and input into the LLM, leveraging the pre-trained knowledge of the LLM to generate features. Finally, we use federated learning to interact with the features, enhancing the data and ensuring data security. Evaluation results indicate that MIVDL outperforms state-of-the-art detectors across three datasets.

In the future, we first want to consider vulnerabilities in other programming languages (such as Java and Python). We second want to consider better modal fusion methods. We third want to consider other new distributed learning methods.

CRediT authorship contribution statement

Zilong Ren: Writing – review & editing, Writing – original draft, Software, Methodology, Data curation. **Xiaolin Ju:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Xiang Chen:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Yubin Qu:** Writing – review & editing, Supervision, Software, Investigation.

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.

Data availability

The data that has been used is confidential.

References

Allen, F.E., 1970. Control flow analysis. ACM Sigplan Not. 5 (7), 1–19.

Anon, 2020. The exactis breach: 5 things you need to know. https://blog.infoarmor.com/individuals-and-families/the-exactis-breach-5-things-you-need-to-know.

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: Proceedings of the 44th International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 1456–1468.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng. 48 (9), 3280–3296.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2022. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (09), 3280–3296.

Cheng, B., Zhao, M., Wang, K., Wang, M., Bai, G., Feng, R., Guo, Y., Ma, L., Wang, H., 2024. Beyond fidelity: Explaining vulnerability localization of learning-based detectors. ACM Trans. Softw. Eng. Methodol.

Cherem, S., Princehouse, L., Rugina, R., 2007. Practical memory leak detection using guarded value-flow analysis. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, pp. 480–491.

Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2017. Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368.

Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2018. Automatic feature learning for predicting vulnerable software components. IEEE Trans. Softw. Eng. 47 (1), 67–85.

Do, C.X., Luu, N.T., Nguyen, P.T.L., 2024. Optimizing software vulnerability detection using RoBERTa and machine learning. Autom. Softw. Eng. 31 (2), 40.

El Ouadrhiri, A., Abdelhadi, A., 2022. Differential privacy for deep and federated learning: A survey. IEEE Access 10, 22359–22380.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512.

Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J., Zhang, C., 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, Montreal, QC, Canada, pp. 72–82.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, Online, pp. 1536–1547.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. IEEE, pp. 608–620.

Groh, F., Ruppert, L., Wieschollek, P., Lensch, H.P., 2022. Ggnn: Graph-based gpu nearest neighbor search. IEEE Trans. Big Data 9 (1), 267–279.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. In: International Conference on Learning Representations.

He, Y., Chen, Z., Goues, C.L., 2023. PreciseBugCollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1899–1910.

Heine, D.L., Lam, M.S., 2006. Static detection of leaks in polymorphic containers. In: Proceedings of the 28th International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 252–261.

Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories. MSR, IEEE, pp. 596–607.

Jia, C., Yang, Y., Xia, Y., Chen, Y.-T., Parekh, Z., Pham, H., Le, Q., Sung, Y.-H., Li, Z., Duerig, T., 2021. Scaling up visual and vision-language representation learning with noisy text supervision. In: International Conference on Machine Learning. PMLR, pp. 4904–4916.

Johnson, A., Dempsey, K., Ross, R., Gupta, S., Bailey, D., et al., 2011. Guide for security-focused configuration management of information systems. NIST Spec. Publ. 800 (128), 16.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Konecnỳ, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D., 2016. Federated learning: Strategies for improving communication efficiency. 8, arXiv preprint arXiv:1610.05492.

Kroening, D., Tautschnig, M., 2014. CBMC–C bounded model checker: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings 20. Springer, Berlin, Heidelberg, pp. 389–391.

Le, T.H., Chen, H., Babar, M.A., 2022. A survey on data-driven software vulnerability assessment and prioritization. ACM Comput. Surv. 55 (5), 1–39.

Lester, B., Al-Rfou, R., Constant, N., 2021. The power of scale for parameter-efficient prompt tuning. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, pp. 3045–3059.

Li, X.L., Liang, P., 2021. Prefix-tuning: Optimizing continuous prompts for generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Association for Computational Linguistics, pp. 4582–4597, (Online).

Li, X., Ren, X., Xue, Y., Xing, Z., Sun, J., 2023. Prediction of vulnerability characteristics based on vulnerability description and prompt learning. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, Taipa, Macao, pp. 604–615.

Li, Y., Wang, S., Nguyen, T.N., 2021a. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 292–303.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secur. Comput. 19 (4), 2244–2258.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., Tang, J., 2023. GPT understands, too. AI Open.

McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A., 2017. Communication-efficient learning of deep networks from decentralized data. In: Artificial Intelligence and Statistics. PMLR, pp. 1273–1282.

MITRE, 2024. 2024 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html/, (Online; accessed 2024).

Nie, E., Liang, S., Schmid, H., Schütze, H., 2022. Cross-lingual retrieval augmented prompt for low-resource languages. arXiv e-prints arXiv–2212.

Nord, R.L., 2017. Software vulnerabilities, defects, and design flaws: A technical debt perspective. In: Fourteenth Annual Acquisition Research Symposium. Acquisition Research Program, Boston, USA, p. 451.

Peters, F., Menzies, T., Layman, L., 2015. LACE2: Better privacy-preserving data sharing for cross project defect prediction. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, IEEE, pp. 801–811.

Ren, Z., Ju, X., Chen, X., Shen, H., 2024. ProRLearn: boosting prompt tuning-based vulnerability detection by reinforcement learning. Autom. Softw. Eng. 31 (2), 38.

Ruan, X., Yu, Y., Ma, W., Cai, B., 2023. Prompt learning for developing software exploits. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware. pp. 154–164.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, Orlando, FL, USA, pp. 757–762.

Schick, T., Schütze, H., 2021. Exploiting cloze-questions for few-shot text classification and natural language inference. In: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume. Association for Computational Linguistics, Online, pp. 255–269.

Tanha, J., Abdi, Y., Samadi, N., Razzaghi, N., Asadpour, M., 2020. Boosting methods for multi-class imbalanced data classification: an experimental review. J. Big Data 7 (1), 1–47.

Wang, S., Liu, T., Nam, J., Tan, L., 2018. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46 (12), 1267–1293.

Wang, M., Tao, C., Guo, H., 2023. LCVD: Loop-oriented code vulnerability detection via graph neural network. J. Syst. Softw. 202, 111706.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., Lyu, M.R., 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 382–394.

Wei, K., Li, J., Ding, M., Ma, C., Yang, H.H., Farokhi, F., Jin, S., Quek, T.Q., Poor, H.V., 2020. Federated learning with differential privacy: Algorithms and performance analysis. IEEE Trans. Inf. Forensics Secur. 15, 3454–3469.

Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. arXiv preprint arXiv:2302.04675.

Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.

Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H., 2022. VulCNN: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering. Association for Computing Machinery, Pittsburgh, Pennsylvania, pp. 2365–2376.

Xu, J., Glicksberg, B.S., Su, C., Walker, P., Bian, J., Wang, F., 2021. Federated learning for healthcare informatics. J. Heal. Informatics Res. 5, 1–19.

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. IEEE, pp. 590–604.

Yamamoto, H., Wang, D., Rajbahadur, G.K., Kondo, M., Kamei, Y., Ubayashi, N., 2023. Towards privacy preserving cross project defect prediction with federated learning. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 485–496.

Yin, L., Feng, J., Xun, H., Sun, Z., Cheng, X., 2021. A privacy-preserving federated learning for multiparty data sharing in social IoTs. IEEE Trans. Netw. Sci. Eng. 8 (3), 2706–2718.

Yu, L., Lu, J., Liu, X., Yang, L., Zhang, F., Ma, J., 2023. PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 556–567.

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: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794.

Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., Gao, Y., 2021. A survey on federated learning. Knowl.-Based Syst. 216, 106775.

Zhang, C., Yu, T., Liu, B., Xin, Y., 2024. Vulnerability detection based on federated learning. Inf. Softw. Technol. 167, 107371.

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

Zilong Ren is pursuing a Master's degree at the School of Artificial Intelligence and Computer Science, Nantong University. His research interests include vulnerability detection.

Xiaolin Ju (Member, IEEE) was born in April 1976. He received a B.S. in information science from Wuhan University in 1998, an M.Sc. degree in computer science from Southeast University in 2004, and a Ph.D. in computer science from the Chinese University of Mining Technology in 2014. He is currently an Associate Professor at the School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China. His research interests include software testing, such as collective intelligence, deep learning testing and optimization, and software defects analysis.

Xiang Chen received a B.Sc. in the school of management from Xi'an Jiaotong University, China, in 2002. Then, he received his M.Sc. and Ph.D. in computer software

and theory from Nanjing University, China, in 2008 and 2011, respectively. He is currently an Associate Professor at the School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software - Practice and Experience, Automated Software Engineering, Journal of Computer Science and Technology, IET Software, Software Quality Journal, Knowledge-based Systems, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER). His research interests include software engineering, particularly software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information about him can be found at: https://xchencs.github.io

Yubin Qu was born in Nanyang, China in 1981. He received the B.S. and M.S. degrees in Computer Science and Technology from Henan Polytechnic University in China in 2004 and 2008. He is currently pursuing a doctoral degree at Army Engineering University of PLA. Since 2022, he has been an associate professor with Information Engineering Institute, Jiangsu College of Engineering and Technology. He is the author of more than 10 articles. His research interests include software maintenance, software testing, and machine learning.