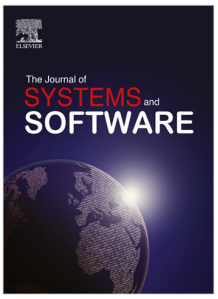
Journal Pre-proof



GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, Zhilong Cai

PII: S0164-1212(24)00074-8
DOI: https://doi.org/10.1016/j.jss.2024.112031
Reference: JSS 112031

To appear in: *The Journal of Systems & Software*

Received date : 18 October 2023
Revised date : 4 February 2024
Accepted date : 18 March 2024

Please cite this article as: G. Lu, X. Ju, X. Chen et al., GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *The Journal of Systems & Software* (2024), doi: https://doi.org/10.1016/j.jss.2024.112031.

GRACE: Empowering LLM-based Software Vulnerability Detection with Graph Structure and In-context Learning

Guilong Lu[a], Xiaolin Ju[a,*], Xiang Chen[a,*], Wenlong Pei[a], Zhilong Cai[a]

[a]*School of Information Science and Technology, Nantong University, Nantong, China*

Abstract

Software vulnerabilities inflict considerable economic and societal harm. Therefore, timely and accurate detection of these flaws has become vital. Large language models (LLMs) have emerged as a promising tool for vulnerability detection in recent studies. However, their effectiveness suffers when limited to plain text source code, which may ignore the syntactic and semantic information of the code. To address this limitation, we propose a novel vulnerability detection approach GRACE that empowers LLM-based software vulnerability detection by incorporating graph structural information in the code and in-context learning. We also design an effective demonstration retrieval approach that identifies highly relevant code examples by considering semantic, lexical, and syntactic similarities for the target code to provide better demonstrations for in-context learning. To evaluate the effectiveness of GRACE, we conducted an empirical study on three vulnerability detection datasets (i.e., Devign, Reveal, and BigVul). The results demonstrate that GRACE outperforms six state-of-the-art vulnerability detection baselines by at least 28.65% in terms of the F1 score across these three datasets. Therefore, our study highlights the effectiveness of integrating graph structural information and in-context learning in LLMs for vulnerability detection. These findings motivate further investigation into tailoring such approaches for specific vulnerability types or adapting them to other security tasks.

Keywords: Vulnerability Detection, Large Language Model, In-context Learning, Source Code Representation, Graph Structure.

1. Introduction

The complexity of software components and structures has led to an increase in software vulnerability exploitation. This has resulted in frequent security incidents (such as hacker attacks, privacy breaches, and data extortion) [1]. For example, Facebook experienced a system vulnerability exposure in September 2018 [2], exploited by hackers to steal the personal information of approximately 50 million users, including susceptible data usernames, passwords, IP addresses, browsing histories, and contact information.

In recent years, the rapid advancement of deep learning (DL) technology has provided a solid technical foundation for performing vulnerability detection [3, 4, 5, 6, 7, 8, 9]. For example, the VulDeePecker [10] divided the code into code gadgets, using them as the granularity, and employed a bidirectional Long Short Term Memory (LSTM) Network to extract features and conduct vulnerability detection. The IVDetect [11] utilized the Feature-Attention Graph Convolutional Network model (FA-GCN) for vulnerability detection. IVDetect considered both the vulnerability statement and its context through data dependency and control dependency, resulting in enhanced detection performance.

However, existing deep learning-based vulnerability detection approaches suffer from two limitations. The first limitation is the significant time required for model training, as demonstrated by IVDetect [11], which takes up to 9 days. The second limitation is that the size of the training dataset constrains the performance of these models.

Recent advancements in LLMs have shown promise in alleviating these two limitations [12]. However, according to the evaluation results, Cheshkov et al. [13] found that using LLMs for vulnerability detection is deemed unsatisfactory. We conducted an in-depth analysis of the underlying reasons and found that the knowledge acquired by the LLM is insufficient. Based on our observation, we resorted to two customizable optimizations. First, we integrate graph structure information with the LLM to enable the model to learn the structural information of the source code, such as data dependency relationships. Second, incorporating domain knowledge through in-context learning has been shown to help improve LLMs' performance [14]. Therefore, we provide high-quality domain knowledge to the LLMs for effective vulnerability detection.

In this study, we propose **GRACE**, a novel vulnerability detection approach that empowers large langua**G**e model with g**RA**ph stru**C**tur**E** and in-context learning. GRACE comprises three main modules: (1) **Demonstration selection module**, which selects the most similar prompt by simultaneously considering semantic similarity, syntactic similarity, and lexical similarity. (2) **Graph structure representation module**, which integrates abstract syntax tree (AST), program dependence graph (PDG),

*Xiaolin Ju and Xiang Chen are corresponding authors.
Email addresses: guil.lu@outlook.com (Guilong Lu), ju.xl@ntu.edu.cn (Xiaolin Ju), xchencs@ntu.edu.cn (Xiang Chen), wl.pei@outlook.com (Wenlong Pei), zhil.cai@outlook.com (Zhilong Cai)

and control flow graph (CFG) to enable LLM to capture more code structure information. (3) **Enhanced Vulnerability Detection Module**, which is designed to improve LLMs' capability in mining vulnerability-domain knowledge. It does so by creating a new basic prompt design that includes domain information and identity information, as well as adding graph structures prompt and in-context learning prompt.

To evaluate the performance of GRACE, we utilize three widely studied benchmark datasets for software vulnerability detection: Reveal [15], FFmpeg+Qemu [16], and Big-Vul [17]. We comprehensively compare six state-of-the-art software vulnerability detection baseline approaches, including three graph-based approaches (i.e., Devign [16], Reveal [15], and IVDetect [11]), and three token-based approaches (i.e., VulDeePecker [10], Russell et al. [18]), SySeVR [19]). The comparison results demonstrate that our approach GRACE outperforms all baseline approaches. In particular, GRACE achieves an average improvement of at least 28.65% in terms of F1 score compared to the baselines across the three datasets.

In summary, the main contributions of our work are as follows:

- We present a novel approach GRACE by introducing graph structure information into LLM for vulnerability detection, which can effectively address the limitations of LLM when only treating the code as plain text.
- To provide better demonstrations for in-context learning, we design an effective demonstration retrieval approach that identifies the most similar code by simultaneously considering semantic, lexical, and syntactic similarity.
- We perform a large-scale evaluation of GRACE on three public benchmark datasets, and the results demonstrate GRACE can achieve better performance compared to state-of-the-art DL-based models for vulnerability detection.

Open science. Our source code and experimental data are available at: *https://github.com/P-E-Vul/GRACE*.

Paper Organization. Section 2 introduces the background and research motivation. Section 3 presents the overall architecture of GRACE and the three modules in detail, including the Demonstration selection module, Graph construction information generation module, and Enhanced vulnerability detection module. Section 4 describes the experimental setup, including datasets, baselines, metrics, and experimental settings. Section 5 introduces the experimental results and analysis. Section 6 discusses the effectiveness of GRACE for vulnerability type detection and the threats to validity. Section 7 analyzes related work, and Section 8 concludes this paper.

2. Background and Motivation

2.1. Vulnerability Detection

Software security is a critical subject in the field of software engineering [20]. Within software security, vulnerability detection holds significant importance [21]. A vulnerability detection task involves analyzing software systems or code to identify vulnerabilities. Its primary objective is to assist in discovering and repairing vulnerabilities within software systems, thereby enhancing system security and mitigating potential security risks. In previous studies, researchers proposed program analysis (PA)-based approaches to detect vulnerabilities [22]. For example, Cppcheck[1] used unique code analysis and focused on detecting undefined behavior and dangerous code structures [23]. Unfortunately, PA-based approaches rely on predefined patterns to identify vulnerabilities [24], and the predefined patterns need to be manually created by security experts, which can be time-consuming.

In recent years, there has been a surge in the utilization of deep learning (DL) for the automated detection of software vulnerabilities [25, 26, 27]. This approach involves leveraging learned patterns from existing vulnerability data to enhance the accuracy and efficiency of the detection process. A DL-based vulnerability detector generally comprises three key components [28].

❶ **Learn representation for the code snippet.** Every code snippet extracted from the target program must be tokenized into a sequence of tokens and then encoded into a vector representation.

❷ **Train the DL model.** A neural network model, such as LSTM, CNN, or self-attention, is specifically designed and trained on a dataset of vectorized code snippets. Based on their vector representations, the model learns to analyze the patterns in vulnerable and non-vulnerable codes.

❸ **Perform vulnerability detection.** Once trained, the model can process new code snippets, converting them into vectors and classifying them as vulnerable or non-vulnerable based on the patterns it learned during training. The accuracy of the model's predictions is evaluated using a test dataset.

2.2. Code Representation

Code representation, which converts source code into appropriate formats, is a vital technique extensively studied in vulnerability detection [29]. By accurately representing the semantics and structures of code, hidden vulnerability patterns can be effectively extracted and analyzed. Many existing works strive to understand program behaviors by customizing effective techniques for different tasks, which has achieved promising results [30, 18, 3]. In general, we can classify these works into four main types of code representations: feature-based, sequence-based, tree-based, and graph-based [31].

Feature-based approaches require extracting explicit program features from source code and feeding them into machine learning-based (ML-based) models for vulnerability analysis and detection. For example, Bhel et al. [32] proposed using TF-IDF and Naive Bayes algorithms [33] to mine security vulnerabilities from bug reports. Sequence-based approaches consider code as token sequences and convert them into numeric vectors using distributed representations [34] for vulnerability detection. Tree-based approaches deal with highly structured data, and therefore, many encoding techniques aim to extract

[1] `http://cppcheck.net/`

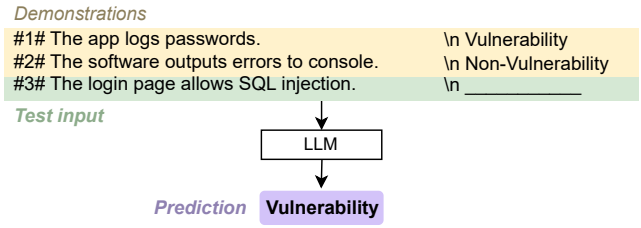


Figure 1: An overview of in-context learning. Two examples are provided here (i.e., #1# and #2#)

structural information, such as Abstract Syntax Trees (ASTs), to capture semantics [35, 36]. Graph-based approaches attempt to incorporate more structural information into graphs to represent program semantics, such as Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), Call Graphs (CGs), and Program Dependence Graphs (PDGs) on vulnerability detection [37]. Among them, CFGs model execution flows and are commonly used for control-flow-related vulnerabilities; DFGs capture data dependencies useful for data flow analysis; CGs and PDGs incorporate control and data flow information to capture complex dependencies.

2.3. In-context learning

LLMs have demonstrated strong performance on various downstream tasks [38, 39]. Fine-tuning these models has been a popular method for transferring them to new tasks [40, 29]. However, fine-tuning an LLM can be impractical due to the high computational resources it requires [41]. To mitigate this issue, Brown et al. [42] proposed in-context learning for new tasks by concatenating the training data as demonstrations without needing any gradient updates. LLMs can learn and infer new tasks through given examples without fine-tuning, as illustrated in Figure 1. Due to the vast corpora, they are trained on, LLMs have acquired extensive domain knowledge, enabling them to generalize to unseen tasks through in-context learning without the requirement for fine-tuning.

2.4. Research Motivation

DL-based approaches have dominated vulnerability detection in recent years. However, these approaches face two significant challenges: (1) Model performance relies on training data scale. Li et al. [11] conducted experiments on the Big-Vul dataset using five different proportional splits: 40:30:30, 50:25:25, 60:20:20, 70:15:15, and 80:10:10. The results demonstrated a progressive increase in the model's AUC metric, increasing from a minimum of 69% to 90%. (2) Training models are incredibly time-consuming. For example, Wu et al. [43] reported that their VulCNN model required a total training time of 2054.89 minutes. Similarly, Li et al. [11] spent 9 days and 23 hours processing data and conducting training on the Big-Vul dataset.

The rise of LLMs overcomes these limitations and has achieved impressive results on tasks like code comment generation [12] and program repair [44, 45]. However, in current research related to using LLMs for vulnerability detection, previous works have only considered code as plain text, and their effectiveness has been limited [13]. Therefore, we further consider combining in-context learning and graph structural information to enhance LLMs' capability for vulnerability detection, which can provide a promising direction for future vulnerability detection studies.

3. Approach

The framework of our approach is illustrated in Figure 2, which comprises three modules: the **Demonstration Selection Module**, the **Graph Structure Information Generation Module**, and the **Enhanced Vulnerability Detection Module**. In the subsequent sections, we provide a detailed explanation of each of these modules.

3.1. Demonstration selection module

Prior studies have demonstrated that incorporating demonstrations can assist models in better capturing developers' intents, thereby enhancing the accuracy of predictions [46, 47, 48]. To obtain high-quality demonstrations, we design a novel retrieval approach that identifies the most similar demonstrations by jointly modeling semantic similarity, lexical similarity, and syntactic similarity between the context and source code examples. Different from previous methods, such as LSI [49], NNGen [50], and CloCom [51], which only focused on lexical and syntactic information, semantic similarity was not taken into account, resulting in situations where code snippets with high lexical and syntactic similarity were selected, but with low semantic similarity. To alleviate this limitation, we first compare the semantic similarity and filter out code snippets with low semantic similarity. Subsequently, we consider the lexical and syntactic information. Next, we will provide a detailed description of our approach.

We first leverage the codeT5 [52] model to extract code semantic features and reduce feature dimensionality via T-SNE [53]. Then, we measure the $L2$ distance between the target code and the training set to compute semantic similarity. Based on this similarity, we retrieve the top K code snippets from the training set that are most similar to the target code. Finally, we incorporate lexical and syntactic similarities to identify the most similar instance among top K candidates. Specifically, regarding lexical similarity, we treat code snippets as token sets and quantify the similarity between set_a and set_b using statistical methods.

$$\text{lexical\_similarity}(A, B) = \frac{|\ set\_a \cap set\_b\ |}{|\ set\_a \cup set\_b\ |} \quad (1)$$

where set_a represents the token sets of the target code, while set_b represents the token sets of a candidate code. As for syntactic similarity, we consider the AST information of the source code. Specifically, we first use Joern [2] to parse the code into corresponding ASTs. Then, since tree matching algorithms (e.g., tree edit distance [54]) are computationally expensive for

[2] https://joern.io/

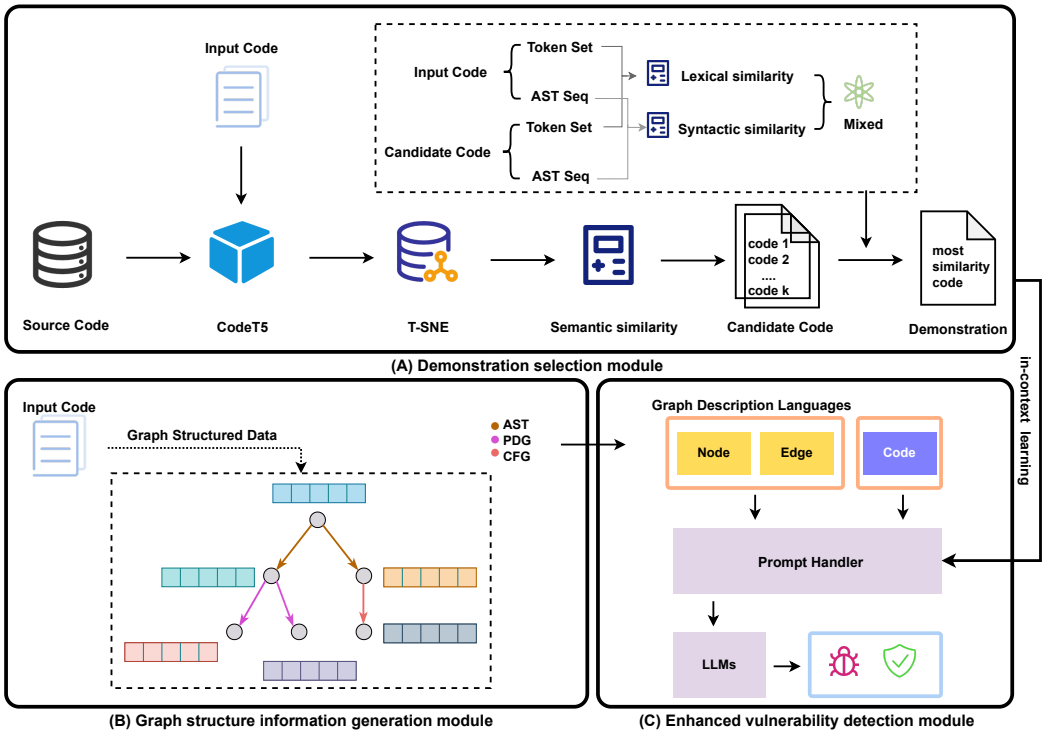


Figure 2: The architecture of GRACE, which mainly contains three components: (A) Demonstration selection module; (B) Graph structure information generation module; (C) Enhanced vulnerability detection module.

similarity evaluation, we adopt SimSBT [55] to traverse the AST type nodes and generate sequences that better capture the tree structure. For two code snippets A and B, we obtain the ordered sequences a and b via SimSBT and then calculate the edit distance between a and b to quantify the sequential similarity.

$$\text{syntactic\_similarity}(A,B) = \frac{\text{sum}(\text{len}(a),\text{len}(b)) - \text{lev}}{\text{sum}(\text{len}(a),\text{len}(b))} \quad (2)$$

Where *lev* is the Levenshtein distance. Finally, we propose a mixed score to retrieve the most similar instances from candidates. For two code snippets A and B, mixed-score is defined as:

$$\begin{aligned}\text{mixed\_score}(A,B) = \omega \times \text{lexical\_similarity}(A,B) \\ +(1-\omega) \times \text{syntactic\_similarity}(A,B)\end{aligned} \quad (3)$$

where ω is a parameter that can adjust the weights between different similarities.

3.2. Graph structure information generation module

In this module, we leverage the Joern tool to generate a Code Property Graph (CPG) representation of the source code. CPG combines CFG, PDG, and AST. As depicted in Figure 4, a C function (Figure 3 shown) is transformed into a graph consisting of nodes and edges of various types. The nodes and black edges in the graph represent the AST, capturing the function's syntactic information. However, the AST alone does not provide sufficient information for the model to reason about

```
01. void foo()
02. {
03.     int a = 43;
04.     char arr[55];
05.     if (a<55)
06.     {
07.         a=63;
08.     }
09.     arr[a] = 'X'
10. }
```

Figure 3: A snippet of C code

the function's semantics. Therefore, CPG augments the AST with semantic information, such as data flow and control flow. Specifically, in Figure 4, the blue edges with the "D_a" label indicate data dependencies between nodes, signifying that the value defined in the variable "a" is utilized in the node's subtree. Similarly, the green edges represent execution order, such as conditional branches. By incorporating both syntax and semantics, we enable the model to better comprehend the information conveyed by the function. Figure 5 presents the node and edge list output generated by Joern.

In general, the AST depicts the syntactic structure of the program code. The PDG models the data and control dependencies within the code. The CFG represents all paths that might be traversed during the program execution. By combining these graphs, the CPG provides a comprehensive representation of the structural and semantic information of the code. The set of edges for the CPGs in our target datasets consists of 12 edge

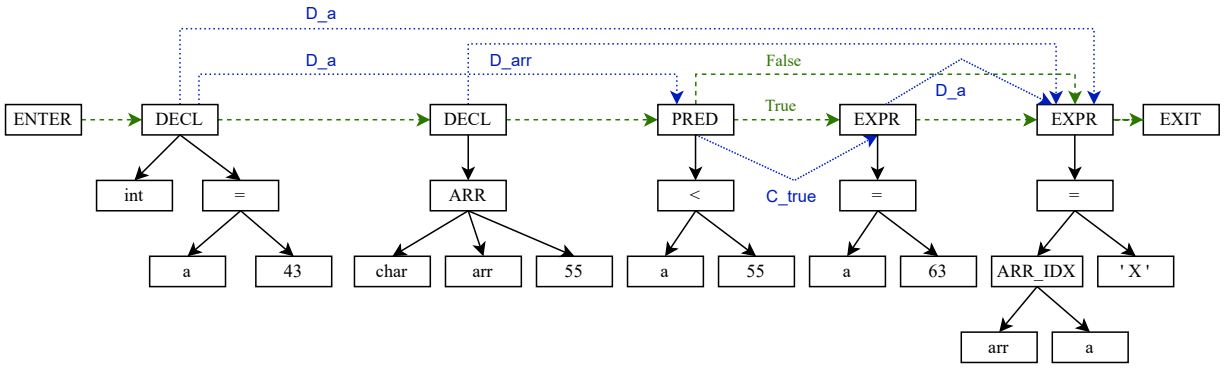


Figure 4: An example of a CPG derived from the functions in Figure 3. The solid black lines represent AST, the dashed green lines represent CFG, and the dashed blue lines represent PDG. Here, the complexity has been simplified based on [56], and the actual number of edges and nodes generated by Joern is approximately twice as shown here.

NODES

Node	Type	Code
3	FunctionDef	foo()
5	IdentifierDeclStmt	int a=43
6	IdentifierDecl	a=43
7	IdentifierDeclType	int
...	...	...
17	IfStatement	if(a<55)
...	...	...
39	Symbol	a

EDGES

Node1	Node2	EdgeType
6	7	IS_AST_PARENT
5	6	IS_AST_PARENT
...	...	...
36	5	FLOWS TO
5	18	...
18	39	USE
...	...	...
5	39	DEF

Figure 5: Output generated by Joern for the function shown in Figure 3. Shown is a list of CPG nodes and edges, indexed by node IDs.

types. We show the details in Table 1.

3.3. Vulnerability detection module

After curating the demonstrations for in-context learning and customizing the graph structure prompts, we designed a set of prompting schemes for vulnerability detection. Our prompts consist of two components: ❶ basic prompt and ❷ auxiliary information.

Basic prompt. First, to get results when using an LLM as a detector, a basic prompt is required. In this paper, we use the following basic prompt and ask the LLM to output a fixed answer format (vulnerable or not vulnerable) to express the result:

Table 1: Edge Types and Descriptions in CPG

Egde types	Description (the relationship between nodes)
IS CLASS OF	A class and its subclass.
IS FUNCTION OF AST	A function and its AST representation
IS FUNCTION OF CFG	A function and its CFG representation
IS AST PARENT	A node in the AST and its parent node
USE	A node in the AST and the variables it uses
DEF	A node in the AST and the variables it defines
DOM	A node in the AST and the dominator tree
POST DOM	A node in the AST and the post-dominator tree
CONTROLS	A node in the AST and the control flow graph
DECLARES	A node in the AST and the declared variables
FLOWS TO	A node in the AST and the nodes that it flows to
REACHES	A node in the AST and the nodes that it reaches

> Basic prompt (P_b): "[Code snippet] In the above code snippet, check for potential security vulnerabilities and output either 'Vulnerable' or 'Non-vulnerable'."

Where [Code snippet] indicates the test function. The fixed answer format of 'Vulnerable' or 'Not Vulnerable' gives the model clear guidelines for communicating its assessment. This converts the open-ended vulnerability detection task into a binary classification that can be more easily interpreted.

Considering that the basic prompt alone is not enough [57], we improve on the basic prompt by adding identity information. Specifically, we use identity information in the prompt to explicitly specify the role of the LLM as a software engineering system:

> Enhance basic prompt by identity information (P_i): "You are now an excellent programmer."

Adding this clear statement of identity primes the model to adopt the correct mindset for the task. It activates the model's capabilities related to security analysis and vulnerability finding.

After obtaining the identity information, LLM is aware of the role it is supposed to play. However, we believe that this

alone is insufficient. Therefore, we introduce domain-specific information to further enhance the basic prompt. We design a domain prompt as follows:

> Enhance basic prompt by domain information (P_d): "You are conducting a function vulnerability detection task for C/C++ language."

By enhancing basic prompts with identity information and domain information, we have developed a more comprehensive and personalized vulnerability detection prompt framework. This framework guides the LLM to analyze vulnerabilities within specific identities and domains, resulting in more accurate vulnerability detection outcomes.

Auxiliary information. In this paper, the auxiliary information consists of two parts: in-context learning demonstrations and graph structure information. The in-context learning demonstrations are obtained through Module A, which retrieves the most similar functions to the target code. The graph structure information is generated by Module B and includes the nodes and edges corresponding to the functions.

The in-context learning demonstrations are examples of code that are similar to the target code. These demonstrations help the LLM understand the specific context and patterns associated with the target code. By learning from and analyzing functions that resemble the target code, LLM can enhance its vulnerability detection capabilities.

The graph structure information provides a CPG representation of the code. The nodes represent the functions, and the edges represent the connections between them. By incorporating node information and edge information, LLM gains a more comprehensive understanding of the code's structure, facilitating a more holistic analysis of vulnerabilities.

For a more intuitive description of our method, we provide an example in Figure 6 that illustrates all the information fed to the model. The block named [code snippet] represents the code that needs to be detected, and its specific style can be referred to in Figure 3. The blocks named [edge information] and [node information] represent the graph structure information generated in our Section 3.2, and their styles are displayed in Figure 5, respectively. Additionally, we have the block [demonstration], which is a code snippet generated in our Section 3.1 for LLMS learning. We will input the corresponding label for this demonstration in [label], which can be either `vulnerable` or `non-vulnerable`.

In the auxiliary information, it merits noting that the prompt length needs to conform to the in-context window limitation of LLMs. Specifically, the context window of GPT4 used in the paper is limited to 8192 tokens. When our content exceeds 8192 tokens, the remaining content will be truncated.

4. Experimental Setup

This section describes our experimental setup to address our research questions.

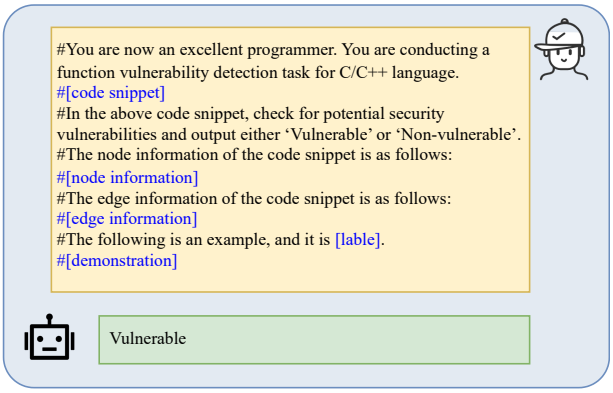


Figure 6: A specific example of inputting all the information to LLMs.

4.1. Research Questions

To evaluate the effectiveness of our proposed GRACE, we want to answer the following four research questions (RQs):

RQ1: How effective is GRACE in vulnerability detection?

Motivation: In this RQ, we aim to investigate whether GRACE can enhance the vulnerability detection capability of LLMs by incorporating graph structural information and in-context learning. We compare GRACE with six state-of-the-art baselines, VulDeePecker [10], Russell et al. [18], SySeVR [19], Devign [16], Reveal [15], and IVDetect [11], to examine if GRACE outperforms these baselines.

RQ2: How does graph structural information contribute to the performance of GRACE?

Motivation: In this RQ, we aim to know whether the graph structural information customized for vulnerability detection can improve the performance of GRACE. We conduct an ablation study, comparing GRACE with and without the graph structural information.

RQ3: How does our in-context learning contribute to the performance of GRACE?

Motivation: In this RQ, we aim to know the effectiveness of the demonstrations retrieved through our proposed approach. We compare three cases: no demonstration, a randomly selected demonstration, and a demonstration retrieved through our proposed approach.

RQ4: How effective is our proposed enhanced basic prompt?

Motivation: In this RQ, we aim to investigate the effectiveness of our proposed enhanced prompt. We compare versions with and without domain knowledge and versions with and without identity type in the basic prompt.

4.2. Datasets

In our experiment, we use three popular datasets called FFmpeg+Qemu [16], Big-Vul [17], and Reveal [15]. These datasets have been widely used in vulnerability studies, such as AMPLE [58] and IVDetect [11]. We show the details of these three datasets as follows.

- **FFmpeg+Qemu** [16]. The FFmpeg+Qemu dataset was manually labeled and sourced from two open-source C

projects: FFmpeg and Qemu. Specifically, it contains 10,067 functions labeled as vulnerable and 12,294 functions labeled as non-vulnerable.

- **Big-Vul** [17]. The Big-Vul dataset was sourced from over 300 open-source C/C++ projects on GitHub, covering 91 distinct vulnerability types listed in the Common Vulnerabilities and Exposures (CVE) database from 2002 to 2019. The dataset comprises 10,547 vulnerable functions and 179,299 non-vulnerable functions.
- **Reveal** [15]. The Reveal dataset was sourced from two open-source projects: Linux Debian Kernel and Chromium. It comprises 1,664 vulnerable functions and 16,505 non-vulnerable functions.

Table 2 presents the details of these three datasets, including the total number of samples, the number of vulnerable samples, the number of non-vulnerable samples, and the ratio of vulnerabilities.

Table 2: Statistics of the dataset used in our vulnerability detection study

Dataset	Samples	Vul	Non-vul	Vul Ratio (%)
FFmpeg+Qemu [16]	22,361	10,067	12,294	45.02
Big-Vul [17]	179,299	10,547	168,752	5.88
Reveal [15]	18,169	1,664	16,505	9.16

4.3. Baseline Methods

To evaluate the effectiveness of GRACE, we compare it with three sequence-based methods (i.e., VulDeePecker [10], Russell et al. [18], SySeVR [19]) and three graph-based methods (i.e., Devign [16], Reveal [15], IVDetect [11]). These baselines are widely recognized and highly regarded in the field of vulnerability analysis. Another advantage is that these methods are open-source. We conducted a simple statistical analysis, and as of January 10, 2024, these baselines have been cited as follows: 856, 556, 467, 623, 300, and 122 times, respectively. Moreover, their performance also ranks among the top in the field of vulnerability detection. We show the details of these six baselines as follows.

- **VulDeePecker** [10]. VulDeePecker divides code into code gadgets and uses a bidirectional LSTM network to extract features and perform binary classification.
- **Russell et al.** [18]. Russell et al. first utilize static analysis tools to label the dataset, then employ convolutional neural networks and other techniques to detect vulnerabilities.
- **SySeVR** [19]. SySeVR extracts code representations that incorporate both syntactic and semantic code features. It then utilizes a bidirectional gated recurrent unit (BGRU) network to perform vulnerability detection.
- **Devign** [16]. Devign builds code structure representations by extracting graph information from functions, then feeds this graph information into a Gated Graph Neural Network (GGNN) to categorize the code.
- **Reveal** [15]. Reveal constructed multiple directed graphs from source code and then utilized a GGNN to analyze these graphical representations for detecting vulnerabilities.
- **IVDetect** [11]. IVDetect constructs Program Dependency Graphs (PDGs) and employs a feature attention graph convolutional network to learn representations of these graphs for vulnerability detection.

We reproduce the results of these baselines by replicating their open-source code. The exception is Devign [16], as the original Devign is not open-sourced, so we use Chakraborty et al. [15] proposed third-party open-source code to reproduce it. Additionally, IVDetect [11] requires an excessively long training time, so we report the reproduction results of Wang et al. [58] in our paper.

4.4. Performance Metrics

To assess the effectiveness of GRACE, we employ four commonly used metrics in vulnerability detection: accuracy, precision, recall, and F1 score. These metrics are widely recognized in the software engineering community and have been utilized in numerous studies on vulnerability detection, such as LineVul [59], ReGVD [60], PLBART [61], and the baseline models mentioned in the article.

We define the performance metrics of our model in terms of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). TP represents cases where the model correctly identifies a vulnerability, and TN refers to non-vulnerable code appropriately classified as such. On the other hand, FP denotes instances where benign code is mistakenly categorized as vulnerable. FN corresponds to cases where existing vulnerabilities are missed and not detected by the model. Specific details about the four metrics are as follows.

Accuracy: $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$. Accuracy examines the overall proportion of correct predictions made by a model. It reflects the model's combined capability of detecting vulnerable and non-vulnerable code.

Precision: $Precision = \frac{TP}{TP+FP}$. Precision measures the proportion of TP among the code samples detected as vulnerable by the model. It reflects the model's accuracy when classifying code samples as "vulnerable".

Recall: $Recall = \frac{TP}{TP+FN}$. Recall reflects the proportion of TP samples (i.e., vulnerable code) correctly detected by the model. It measures the model's ability to identify all vulnerable samples. A higher recall means the model can detect more vulnerable samples with fewer missed detections.

F1 score: $F1score = 2 * \frac{Precision*Recall}{Precision+Recall}$. In vulnerability detection, the F1 score comprehensively evaluates a model's precision and recall. It balances both precision and recall by calculating their harmonic mean. A higher F1 score indicates

better overall model performance in terms of both precision and recall.

Using these four metrics, we can evaluate the performance of the vulnerability detection model from various perspectives, providing a comprehensive understanding of its strengths and weaknesses.

4.5. Implementation Details

We utilize GPT-4, the most advanced model for our experiments. We split the dataset into train, validation, and test sets with a ratio of 8:1:1. Notably, GPT-4 does not require the train and validation portions, only the test set. Our method utilizes a test set that is consistent with all baseline test sets. Specifically, in RQ1, we use 100% of the available test data for evaluation. For the ablation experiments conducted in our study (RQ2-RQ4), we adopted a random sampling approach where 30% of the samples from the test set were selected for testing purposes. This decision was made to optimize costs while still maintaining a representative subset of data. To ensure the fairness and consistency of the experiments, the same 30% of data was used for each ablation experiment when comparing different methods. All experiments are performed on an NVIDIA GeForce RTX 4090 GPU server.

5. Experimental Results

5.1. RQ1: How effective is GRACE in vulnerability detection?

Approach. To answer RQ1, we assess the model's performance using the four metrics outlined in Section 4.4. Then, we select six popular vulnerability detection approaches as baselines, including three sequence-based methods (i.e., VulDeePecker [10], Russell etal. [18], SySeVR [19]) and three graph-based methods (i.e., Devign [16], Reveal [15], IVDetect [11]). Experiments are conducted on three commonly used vulnerability datasets, including Devign [16], Big-Vul [17], and Reveal [15]. To guarantee a fair comparison, we will ensure that all training sets for our proposed approach and baselines remain consistent.

Result. As shown in Table 3, our proposed GRACE outperforms all baselines in F1 score across the three datasets. Specifically, GRACE achieves average relative improvements of 28.65%, 62.36%, and 75.71% in F1 score compared to six baseline models on the FFmpeg+Qemu, Reveal, and Big-Vul datasets. Regarding accuracy, GRACE obtains the best accuracy on Reveal while ranking second on FFmpeg+Qemu and Big-Vul. Then, to check whether the performance difference between GRACE and baselines is significant, we conduct Wilcoxon signed-rank tests [62] at a confidence level of 95%. The p-value is less than 0.05, indicating that the performance improvement of GRACE compared to the baselines is statistically significant.

Figure 7 presents a code example that contains a CWE-193 vulnerability. This example is taken from a real-world scenario. Unlike VulDeePecker and Devign, GRACE, based on ChatGPT, incorporates a wider range of software engineering-related data, including vulnerability detection, during training.

```
01. #define PATH_SIZE 60
02.
03. char filename[PATH_SIZE];
04.
05. for (int i = 0; i < PATH_SIZE; i++) {
06.     char c = getc(stdin);
07.     if (c == EOF) {
08.         filename[i] = '\0';
09.         break;
10.     }
11.     filename[i] = c;
12. }
```

Figure 7: An example from the official CWE (Common Weakness Enumeration) website: CWE-193 (Off-by-one Error). While VulDeePecker and Devign fail to detect this vulnerability, GRACE can detect it successfully.

By leveraging the extensive code patterns learned through domain knowledge and identity information, GRACE can effectively identify this vulnerability.

> **Summary for RQ1:** GRACE outperforms the six state-of-the-art baselines in terms of F1 score. Specifically, GRACE achieves average relative improvements of 28.65%, 62.36%, and 75.71% in F1 score, demonstrating the effectiveness of our proposed approach.

5.2. RQ2: How does graph structural information contribute to the performance of GRACE?

Approach. To evaluate the contribution of graph structural information to our proposed GRACE, we conduct controlled experiments comparing its performance with and without this information. For RQ2 experiments, we utilize fixed prompt templates and in-context learning demonstrations.

Result. As shown in Table 4, we can observe that the accuracy and F1 score decrease substantially without the graph structure information. Specifically, the F1 score dropped by 11.47%, 19.79%, and 42.56% on FFmpeg+Qemu, Reveal, and Big-Vul, respectively, compared to using the entire model with graph structure. The accuracy also declined by 15.32%, 30.38%, and 32.47% on the three datasets.

The noticeable decrease in performance demonstrates the significance of integrating graph structure information into our approach. By incorporating graph structure information, our approach gains valuable insights into the structural knowledge and relationships among program entities. This enables the model to make more precise vulnerability predictions.

> **Summary for RQ2:** Incorporating graph structure information into GRACE has a significant impact, resulting in an improvement of 14.82%, 24.64%, and 73.8% in F1 score across the three datasets. This demonstrates the ability of LLMs to understand graph structure information effectively.

Table 3: Comparison results between GRACE and the baselines on the three datasets in vulnerability detection. "-" means that the baseline does not apply to the dataset in this scenario. The best result for each metric is highlighted in bold. The cells shaded in grey represent the performance of the top-3 best approaches for each metric, with darker shading indicating better performance.

Metrics(%) \ Dataset	FFmpeg+Qemu [16]				Reveal [15]				Big-Vul [17]			
Baseline	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
VulDeePecker	49.91	46.05	32.55	38.14	76.37	21.13	13.10	16.17	81.19	**38.44**	12.75	19.15
Russell et al.	57.60	54.76	40.72	46.71	68.51	16.21	52.68	24.79	86.85	14.86	26.97	19.17
SySeVR	47.85	46.06	58.81	51.66	74.33	**40.07**	24.94	30.74	90.10	30.91	14.08	19.34
Devign	56.89	52.50	64.67	57.95	87.49	31.55	36.65	33.91	**92.78**	30.61	15.96	20.98
Reveal	**61.07**	**55.50**	70.70	62.19	81.77	31.55	61.14	41.62	87.14	17.22	34.04	22.87
IVDetect	57.26	52.37	57.55	54.84	-	-	-	-	-	-	-	-
GRACE	59.78	53.94	**82.13**	**65.11**	**89.73**	33.21	**61.53**	**43.13**	90.73	32.52	**39.08**	**35.50**

Table 4: Results of the Ablation Study Without Graph Structure Information. The best result for each metric is highlighted in bold.

Metrics(%) \ Dataset	FFmpeg+Qemu [16]				Reveal [15]				Big-Vul [17]			
Baseline	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
w/o graph	50.93	45.25	80.19	57.82	61.29	27.88	43.26	33.91	61.94	26.91	16.77	20.66
GRACE	**60.13**	**54.58**	**84.68**	**66.38**	**88.12**	**32.05**	**62.01**	**42.26**	**91.69**	**31.85**	**41.21**	**35.93**

5.3. *RQ3: How does our in-context learning contribute to the performance of GRACE?*

Approach. We experiment with different demonstration strategies to investigate the effectiveness of the demonstration selection approach. Three demonstration selection strategies are evaluated. First is zero demonstration, where no examples are provided during learning. Second is random demonstration selection, which randomly selects a function and its label from the dataset as the demonstration. We executed the random strategy twice on each test set and reported the average results. Third is our proposed retrieval method to identify the most relevant demonstration. For the RQ3 experiment, for each test sample, we provide different demonstrations while using fixed prompt information and graph structure information.

Result. Table 5 presents our results. When no demonstrative examples were provided, the LLM's performance in vulnerability detection was limited, emphasizing the importance of in-context learning. Interestingly, when using random demonstrations, the detection performance of LLMs can deteriorate compared to not using any demonstrations. This could be due to the fact that randomly selected examples do not offer meaningful guidance to the models. There are two potential reasons for this: ❶ The random demonstrations lack representativeness and diversity, failing to cover the scope of real-world vulnerabilities. This results in limited knowledge transfer to the models. ❷ Random demonstrations introduce noise that misleads model judgment. Without careful screening, they may include invalid or even erroneous examples. However, the most notable enhancement was observed when providing the most contextually similar example to the model. In this condition, the model showcased a significant boost in its detection capabilities, surpassing both zero and random demonstrations. Specifically, including the most contextually similar example resulted in an average increase of 31.82% in the F1 score across the three vulnerability detection datasets.

Table 5: Results of Different Demonstration Strategies. The best result for each metric is highlighted in bold.

Dataset	Module	Accuracy	Precision	Recall	F1 score
FFmpeg+Qemu [16]	Zero	50.61	47.61	84.50	60.91
	Random	52.23	48.06	**84.93**	61.38
	GRACE	**60.13**	**54.58**	84.68	**66.38**
Reveal [15]	Zero	81.80	17.54	**83.33**	28.98
	Random	76.22	20.23	41.86	27.28
	GRACE	**88.12**	**32.05**	62.01	**42.26**
Big-Vul [17]	Zero	63.32	17.21	**50.44**	25.66
	Random	73.13	21.01	40.51	27.67
	GRACE	**91.69**	**31.85**	41.21	**35.93**

Furthermore, it is worth noting that our method excels in terms of accuracy, precision, and F1 score, with the trade-off of a lower recall. After conducting a detailed analysis, we have identified the underlying reason for this decrease. Our method may be more cautious in identifying vulnerability modules, resulting in a lower recall, but a better precision. At the same time, there is a certain conflict between the recall and the precision, so the F1 score can better integrate the values of these two indicators. In our work, we have found that the F1 score will become better. Although the recall rate has decreased, our method has shown the best performance in accuracy, precision, and F1 score.

These results highlight the importance of incorporating vulnerability contextual cues during detection, as they significantly improve the LLM's ability to detect vulnerabilities accurately. By leveraging relevant demonstrative examples, our approach equips the model to make more informed decisions, improving its overall performance in vulnerability detection tasks.

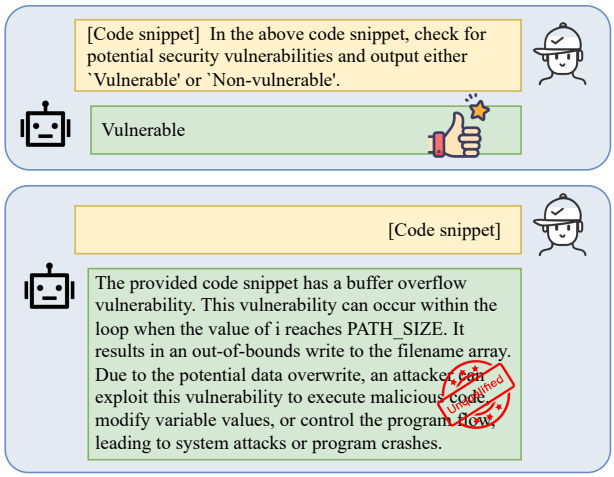


Figure 8: Examples of model outputs with and without basic prompt provided to LLMs

Summary for RQ3: The demonstration retrieval method for obtaining the most similar demonstration has proven effective in enhancing the model's performance in vulnerability detection. On average, incorporating our carefully selected demonstrative examples across the three datasets resulted in a 31.82% model performance improvement in terms of F1 score.

5.4. RQ4: How effective is our proposed enhanced basic prompt?

Approach. To answer RQ4, we attempt to provide different basic prompts to the LLM. Specifically, we devise three different strategies. The first strategy is no domain information (i.e., P_b + P_i), the second is no identity information (i.e., P_b + P_d), and the third is both (i.e., P_b + P_d + P_i). It is worth noting that we will not compare scenarios without a basic prompt. Without the constraint of a basic prompt, the model may produce a significant amount of redundant information rather than the specific answers we need regarding "vulnerable" or "non-vulnerable" aspects. Here, we choose the function in Figure 7 as an example and present the output of the LLM in Figure 8. For the RQ4 experiment, for each test sample, we provide different prompts while keeping the input of fixed graph structure information and in-context demonstrations consistent.

Result. Table 6 presents the results obtained from our experiments. The results demonstrate the significant contributions of the two enhanced basic prompt components designed in our approach.

Using the FFmpeg+Qemu dataset as an example, we compare the model's performance with and without P_d and P_i. When P_d are not provided, the model's accuracy decreases by 7.25% and its F1 score decreases by 12.17% relative to our proposed GRACE. Similarly, when P_i is not included, the model experiences relative decreases of 4.98% in accuracy and 10.17% in F1 score compared to GRACE.

These performance declines underscore the importance of incorporating P_d and P_i in LLMs. They highlight the value of leveraging domain knowledge to improve the model's ability to capture and understand vulnerability Knowledge. Moreover, our findings suggest that P_d has a more pronounced impact on performance enhancement compared to P_i.

Table 6: Results of Different Prompting Methods. The best result for each metric is highlighted in bold.

Dataset	Module	Accuracy	Precision	Recall	F1 score
FFmpeg+Qemu	w/o P_d	55.77	47.23	76.14	58.29
	w/o P_i	57.13	48.23	78.11	59.63
	GRACE	**60.13**	**54.58**	**84.68**	**66.38**
Reveal	w/o P_d	76.26	21.34	60.29	31.52
	w/o P_i	80.41	28.94	**67.22**	40.46
	GRACE	**88.12**	**32.05**	62.01	**42.26**
Big-Vul	w/o P_d	84.35	25.02	30.86	27.63
	w/o P_i	87.21	28.99	33.46	31.06
	GRACE	**91.69**	**31.85**	**41.21**	**35.93**

Summary for RQ4: Our carefully crafted enhanced basic prompt has proven to be effective, and domain-specific knowledge has a more significant influence in the performance of GRACE than identity information. In particular, incorporating domain-specific information into three vulnerability detection datasets resulted in an average improvement of 26.33% in terms of F1 score.

6. Discussion

6.1. How effective is GRACE in vulnerability type detection?

We conduct further investigations to comprehend the capability of GRACE in vulnerability type classification, which is a multi-class classification task, different from vulnerability detection [63]. Accurately identifying vulnerability types can facilitate subsequent tasks like vulnerability repair [64]. We select Devign and Reveal as baselines for comparison. We aim to compare the performance of GRACE with these well-established methods and evaluate its effectiveness in vulnerability type classification.

To obtain labels for vulnerability types, we extract and curate a new dataset from Fan et al. [17], spanning different vulnerability categories from 2002 to 2019. This dataset encompasses functions for 91 distinct vulnerability types, with details tabulated in Table 7. Notably, for any vulnerability type with a total sample of less than 20, we consolidate its label as remain for the analysis.

The categories for multi-class classification in the dataset are imbalanced, with the proportions shown in Table 7, it does not imply that less represented vulnerability types are less important. Therefore, for multi-class classification, we use accuracy and weighted F1 score as the evaluation metric. We will provide a detailed introduction to the weighted F1 score.

Weighted F1 score. *Weighted F1 score* is calculated by taking a weighted average of the F1 score for each class, where

Table 7: The specific vulnerability types and their corresponding proportion and group of vulnerabilities in this paper. Classes with a sample size of less than 20 are grouped in the Remain class. "None type" means the vulnerability is not classified into any class. As these vulnerabilities exist in the real world as well, they are also considered to be a vulnerability type.

Types	Ratio	Nums	Types	Ratio	Nums
CWE-119	19.94%	2127	CWE-415	0.76%	81
None Type	19.85%	2117	CWE-732	0.62%	66
CWE-20	10.71%	1142	CWE-404	0.58%	62
CWE-399	6.90%	736	CWE-79	0.52%	55
CWE-125	5.86%	625	CWE-19	0.52%	55
CWE-264	4.76%	508	CWE-59	0.49%	52
CWE-200	4.72%	503	CWE-17	0.48%	51
CWE-189	3.16%	337	CWE-400	0.45%	48
CWE-416	3.09%	330	CWE-772	0.43%	46
CWE-190	2.88%	307	CWE-269	0.36%	38
CWE-362	2.61%	278	CWE-22	0.33%	35
CWE-476	2.02%	215	CWE-369	0.32%	34
CWE-787	1.86%	198	CWE-18	0.32%	34
CWE-284	1.66%	177	CWE-835	0.32%	34
CWE-254	1.15%	123	Remain	1.57%	167
CWE-310	0.88%	94	Total	100%	10680

the weights are based on the relative frequency of each class in the dataset. The formula is defined as follows:

$$Weighted\ F1\ score = \sum_{i=0}^{m} w_i * (F1\ score)_i \quad (4)$$

where m is the number of classes, w_i is the weight for class i (calculated as the ratio of the number of samples in class i to the total number of samples).

Our comparison results are showcased in Table 8, revealing the superiority of our approach over the baselines in terms of accuracy and weighted F1 score. Specifically, our method demonstrates a substantial average absolute enhancement of 15.59% in accuracy compared to the two baselines. Additionally, our approach achieves an average absolute improvement of 5.86% in Weight F1 score.

Table 8: Comparison results between GRACE and baseline on vulnerability type detection.

Method	Accuracy	Weight F1 score
Devign	19.69	46.71
Reveal	28.36	49.22
GRACE	**39.62**	**53.83**

To better understand GRACE's performance in vulnerability type detection, we conducted an in-depth analysis of different types of vulnerabilities. Our findings are illustrated in Figure 9, where the bars with navy-blue color indicate the accuracy of our GRACE method in detecting different types of vulnerabilities. In contrast, the lines with blue color represent the accuracy of Devign and the lines with gray color represent the accuracy of Reveal. Taking a broader perspective, the curves of the GRACE method consistently outperform those of Devign and Reveal, highlighting the effectiveness of GRACE. Particularly noteworthy is its exceptional performance in detecting CWE-415 and CWE-79 vulnerabilities, surpassing Devign and Reveal by a significant margin. We conduct a thorough analysis to elucidate the underlying factors. For instance, when detecting CWE-79 (Cross-Site Scripting), graph structures aid LLM in comprehending the function's input points, output points, and data transmission paths, thereby facilitating vulnerability analysis. Moreover, the extensive training of the LLM on a vast dataset empowers it to capture a broader range of vulnerability patterns, potentially enhancing its generalization across CWE-79. In comparison, Devign and Reveal rely on static analysis information and self-supervised learning methods, which may encounter generalization limitations due to the training data's quality and diversity. Furthermore, GRACE demonstrates an accuracy exceeding 50% for over 50% of the vulnerability types, underscoring its proficiency in detecting multiple vulnerability types.

We also examine vulnerabilities with an accuracy below 0.5. For example, CWE-254 (Security Features) demonstrates an accuracy rate of only around 10%. This vulnerability typically occurs due to incomplete SSL/TLS verification or misconfigurations in security settings. We conjecture that CWE-254 involves intricate security concerns related to random number generation. It is an inherently complex and technical vulnerability. It requires a solid understanding of randomness, entropy, seeds, algorithms, and the correct implementation and usage of random number generators. The LLM may encounter certain limitations when dealing with such intricacies.

6.2. Threats to Validity

In this subsection, we primarily outline the potential threats to the validity of our research.

Internal threat. GPT-4 was trained on a large amount of open-source data, so there is a possibility of data leakage, meaning GPT-4 may have learned answers to test sets during pre-training. However, our primary goal is to enhance the vulnerability detection capabilities of LLMs by graph structure information and in-context learning. We have conducted ablation studies to demonstrate that graph structure information and our proposed retrieval method can improve the ability of LLMs to detect vulnerabilities.

External threat. External threat is related to the ChatGPT Version. Since the training parameters differ across ChatGPT versions, we cannot guarantee that our conclusions are reproducible across all versions. To mitigate this issue, please use GPT-4 to reproduce the results.

Construct threat. Recent work in prompt engineering has shown that different prompts can lead large models to produce different results. The same applies in the vulnerability domain. To mitigate this issue, we designed a new prompting template that contains the necessary context and auxiliary information. Experiments demonstrate that our newly designed prompting template is effective.

Conclusion threat. In our study, we only tested vulnerabilities in C/C++ code. In principle, GRACE can be applied to other programming languages, as LLMs have been trained on

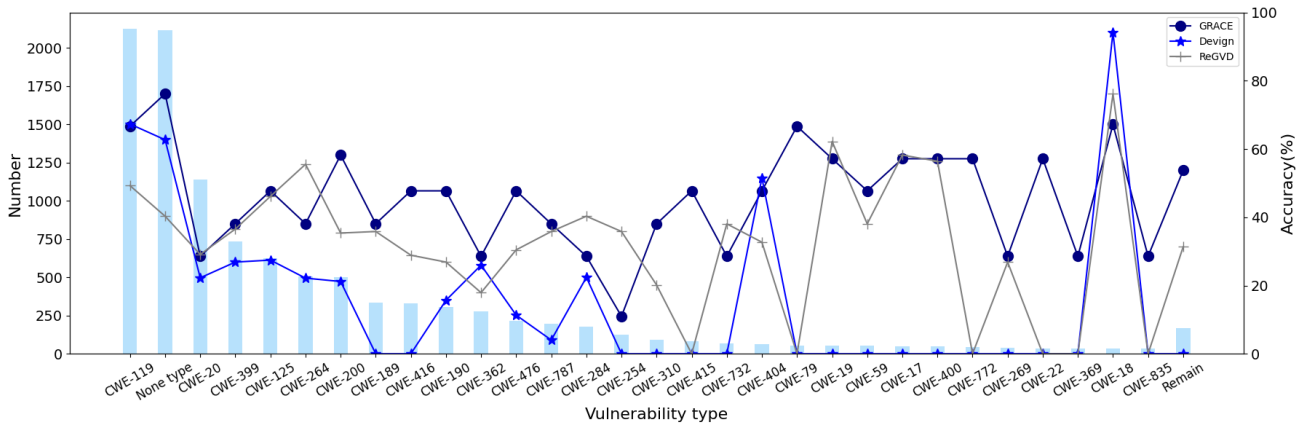


Figure 9: The accuracy of each vulnerability type in GRACE. The X-axis denotes the vulnerability type. The Y-axis on the left and right indicate the number of vulnerabilities and accuracy, respectively. The lines in navy, blue, and gray represent the accuracy of GRACE, Devign, and Reveal, respectively.

massive datasets and have acquired domain knowledge about other languages. However, more research is needed to confirm the effectiveness of other languages, which remains a direction for future work.

7. Related Work

7.1. Deep learning-based Vulnerability Detection

Traditional ML-based approaches for vulnerability detection rely on software metrics as features, such as lines of code, cyclomatic complexity, etc [65, 66]. Unfortunately, collecting these metric features manually is laborious and time-consuming. To alleviate this issue, DL-based vulnerability detection approaches have been proposed, which can automatically learn vulnerability patterns from the data itself.

Sequence-based approaches refer to treating source code as a sequence of bytes or tokens and then feeding it into sequential DL models such as RNNs, LSTMs, CNNs, etc. This allows the model to directly learn representations of vulnerabilities extracted from the code sequence [10, 43, 19, 67]. For example, VulDeePecker [10] employed code gadgets as an intermediate representation, converting code gadgets into fixed symbolic representations. For each code gadget, lexical analysis was performed to construct a vocabulary. Word2vec then transformed these into fixed-length vector representations, which were input to a bidirectional LSTM to predict whether the function contains a vulnerability. However, sequence-based approaches do not take into account the graph structure of source code, which may lead to inaccurate predictions. Therefore, graph-based approaches have been proposed, which can better model the structural information of code.

Graph-based approaches refer to detecting vulnerabilities in source code by constructing a graph representation of the code, such as ASTs or CFGs, and then analyzing it using GNNs [16, 11, 58, 4, 68, 69, 70]. For example, Wen et al. [58] utilized an Edge-aware Graph Convolutional Network (EA-GCN) model for vulnerability detection. They first simplified the graph structure and then applied the GCN model to the preprocessed graph. This approach effectively improves detection performance.

However, training DL-based models for these tasks often requires a substantial amount of time and resources and necessitates tailoring the model to the specific dataset. Therefore, we utilize a pre-trained LLM as the detector, avoiding the issue of lengthy training times. In addition, we improve the vulnerability detection performance of the LLM by leveraging graph structural information and in-context learning.

7.2. In-Context Learning for LLM4SE

LLM has been widely applied in various tasks in the natural language processing field [71, 72, 73]. Its emerging characteristics have also attracted widespread attention in software engineering, including coding, requirements, debugging, refactoring, etc.

In-context learning is a new approach for LLMs to learn tasks. Recently, in software engineering, a series of studies have emerged, such as the empirical study on comment generation by Geng et al., exploring the feasibility of utilizing LLMs to address ambiguous comment generation [12]. Nashid et al. [74] proposed CEDAR, probing two downstream tasks through few-shot learning, i.e., test assertion generation and program repair. However, to our best knowledge, we have not found related studies to employ LLM in-context learning for vulnerability detection. To fill this gap, we design a new demonstration retrieval approach to provide better demonstrations for in-context learning, and our empirical results also confirm the effectiveness of our customized in-context learning approach.

8. Conclusion

In current research related to using LLMs for vulnerability detection, code is typically treated as plain text directly input to the model [13]. This lacks the structural information of code and domain knowledge relevant to vulnerabilities. To mitigate these issues, we propose GRACE, a vulnerability detection approach based on LLM that emphasizes incorporating graph structure information and in-context learning. Our approach GRACE employs a novel demonstration selection strategy that

helps LLMs acquire relevant domain knowledge by selecting high-quality demonstrations. Additionally, enhanced prompting templates allow models to learn more pertinent knowledge. Our experimental results on three C/C++ datasets demonstrate that GRACE outperforms the state-of-the-art of six vulnerability detection baselines, especially improving the F1 score by at least 28.65%. Furthermore, our three ablation studies also validate the effectiveness of graph structure information, the novel demonstration selection strategy, and the customized prompting templates, respectively.

In the future, we first want to enhance GRACE's performance by designing more effective prompting templates and demonstration selection strategies. Secondly, we want to conduct empirical studies on other programming languages to examine whether GRACE could be generalized to additional languages. Finally, we want to investigate other challenges in the vulnerability domain in conjunction with LLMs and graph structure information, such as automatic vulnerability repair [64, 75, 76], which remains an open and worthwhile direction.

Declaration of Competing Interests

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.

CRediT Authorship Contribution Statement

Guilong Lu: Data curation, Software, Validation, Conceptualization, Methodology, Writing -review & editing. **Xiaolin Ju:** Conceptualization, Methodology, Writing -review & editing, Supervision. **Xiang Chen:** Conceptualization, Methodology, Writing -review & editing, Supervision. **Wenlong Pei**: Conceptualization, Data curation, Software. **Zhilong Cai**: Conceptualization, Data curation, Software.

References

[1] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, E. Akin, A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions, Electronics 12 (6) (2023) 1333.

[2] C. Cadwalladr, E. Graham-Harrison, Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach, The guardian 17 (1) (2018) 22.

[3] S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection, Information and Software Technology 136 (2021) 106576.

[4] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, C. Tao, MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks, CoRR abs/2203.02660 (2022).

[5] H. Guo, S. Chen, Z. Xing, X. Li, Y. Bai, J. Sun, Detecting and augmenting missing key aspects in vulnerability descriptions, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (3) (2022) 1–27.

[6] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, Y. Liu, Software composition analysis for vulnerability detection: An empirical study on java projects, in: Proceedings of the 2023 31th acm sigsoft international symposium on foundations of software engineering, 2023.

[7] S. Pan, L. Bao, X. Xia, D. Lo, S. Li, Fine-grained commit-level vulnerability type prediction by cwe tree structure, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 957–969.

[8] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, T. Peng, Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1029–1040.

[9] P. K. Kudjo, J. Chen, A cost-effective strategy for software vulnerability prediction based on bellwether analysis, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 424–427.

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

[11] Y. Li, S. Wang, T. N. Nguyen, Vulnerability detection with fine-grained interpretations, in: ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, ACM, 2021.

[12] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, arXiv preprint arXiv:2304.11384 (2024).

[13] A. Cheshkov, P. Zadorozhny, R. Levichev, Evaluation of chatgpt model for vulnerability detection, arXiv preprint arXiv:2304.07232 (2023).

[14] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, Z. Sui, A survey for in-context learning, arXiv preprint arXiv:2301.00234 (2022).

[15] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48 (09) (2022) 3280–3296.

[16] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Advances in neural information processing systems 32 (2019).

[17] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A C/C++ code vulnerability dataset with code changes and CVE summaries, in: MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, ACM, 2020.

[18] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, M. W. McConley, Automated vulnerability detection in source code using deep representation learning, in: M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, E. Lughofer (Eds.), 17th IEEE International Conference on Machine Learning and Applications, Orlando, FL, USA, December 17-20, IEEE, 2018.

[19] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A framework for using deep learning to detect software vulnerabilities, IEEE Trans. Dependable Secur. Comput. 19 (4) (2022) 2244–2258.

[20] G. McGraw, Software security, IEEE Security & Privacy 2 (2) (2004) 80–83.

[21] B. Liu, L. Shi, Z. Cai, M. Li, Software vulnerability discovery techniques: A survey, in: 2012 fourth international conference on multimedia information networking and security, IEEE, 2012, pp. 152–156.

[22] P. Li, B. Cui, A comparative study on software vulnerability static analysis techniques and tools, in: IEEE international conference on information theory and information security, IEEE, 2010.

[23] A. Kaur, R. Nayyar, A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code, Procedia Computer Science 171 (2020) 2023–2029.

[24] Z. Shen, S. Chen, A survey of automatic software vulnerability detection, program repair, and defect prediction techniques, Security and Communication Networks (2020) 1–16.

[25] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: Statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 596–607.

[26] X. Cheng, G. Zhang, H. Wang, Y. Sui, Path-sensitive code embedding via contrastive learning for software vulnerability detection, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 519–531.

[27] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, Y. Wang, Assbert: Active and semi-supervised bert for smart contract vulnerability detection, Journal of Information Security and Applications 73 (2023) 103423.

[28] Y. Nong, R. Sharma, A. Hamou-Lhadj, X. Luo, H. Cai, Open science in software engineering: A study on deep learning-based vulnerability detection, IEEE Transactions on Software Engineering 49 (4) (2022) 1983–2005.

[29] H. Hanif, S. Maffeis, Vulberta: Simplified source code pre-training for vulnerability detection, in: 2022 International joint conference on neural networks (IJCNN), IEEE, 2022, pp. 1–8.

[30] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, J. Hu, Vulpecker: an automated vulnerability detection system based on code similarity analysis, in: Proceedings of the 32nd annual conference on computer security applications, 2016, pp. 201–213.

[31] J. K. Siow, S. Liu, X. Xie, G. Meng, Y. Liu, Learning program semantics with code representations: An empirical study, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022.

[32] D. Behl, S. Handa, A. Arora, A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf, in: International Conference on Reliability Optimization and Information Technology (ICROIT), IEEE, 2014.

[33] G. I. Webb, E. Keogh, R. Miikkulainen, Naïve bayes., Encyclopedia of machine learning 15 (1) (2010) 713–714.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Advances in neural information processing systems 26 (2013).

[35] R. Ma, Z. Jian, G. Chen, K. Ma, Y. Chen, Rejection: A ast-based reentrancy vulnerability detection method, in: Trusted Computing and Information Security: 13th Chinese Conference, Shanghai, China, October 24–27,, Springer, 2020, pp. 58–71.

[36] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, Z. Shi, Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2021, pp. 224–236.

[37] Y. Wu, J. Lu, Y. Zhang, S. Jin, Vulnerability detection in c/c++ source code with graph representation learning, in: 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2021, pp. 1519–1524.

[38] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, Q. Wang, Software testing with large language model: Survey, landscape, and vision, arXiv preprint arXiv:2307.07221 (2023).

[39] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[40] C. Yu, G. Yang, X. Chen, K. Liu, Y. Zhou, Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022, pp. 82–93.

[41] Y. Gu, X. Han, Z. Liu, M. Huang, Ppt: Pre-trained prompt tuning for few-shot learning, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022, pp. 8410–8423.

[42] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[43] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, H. Jin, Vulcnn: An image-inspired scalable vulnerability detection system, in: Proceedings of the 44th International Conference on Software Engineering(ICSE), 2022, pp. 2365–2376.

[44] C. S. Xia, L. Zhang, Conversational automated program repair, arXiv preprint arXiv:2301.13246 (2023).

[45] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023.

[46] P. Ma, R. Ding, S. Wang, S. Han, D. Zhang, Demonstration of insightpilot: An llm-empowered automated data exploration system, arXiv preprint arXiv:2304.00477 (2023).

[47] M. Luo, X. Xu, Z. Dai, P. Pasupat, M. Kazemi, C. Baral, V. Imbrasaite, V. Y. Zhao, Dr. icl: Demonstration-retrieved in-context learning, arXiv preprint arXiv:2305.14128 (2023).

[48] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al., A survey of large language models, arXiv preprint arXiv:2303.18223 (2023).

[49] S. Haiduc, J. Aponte, L. Moreno, A. Marcus, On the use of automated text summarization techniques for summarizing source code, in: 2010 17th Working conference on reverse engineering, IEEE, 2010, pp. 35–44.

[50] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, X. Wang, Neural-machine-translation-based commit message generation: how far are we?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 373–384.

[51] E. Wong, T. Liu, L. Tan, Clocom: Mining existing source code for automatic comment generation, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 380–389.

[52] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[53] A. C. Belkina, C. O. Ciccolella, R. Anno, R. Halpert, J. Spidlen, J. E. Snyder-Cappione, Automated optimized parameters for t-distributed stochastic neighbor embedding improve visualization and analysis of large datasets, Nature communications 10 (1) (2019) 5415.

[54] P. Bille, A survey on tree edit distance and related problems, Theoretical Computer Science 1 (337) (2005) 217–239.

[55] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, K. Liu, Comformer: Code comment generation via transformer and fusion method-based hybrid code representation, in: 2021 8th International Conference on Dependable Systems and Their Applications (DSA), IEEE, 2021, pp. 30–41.

[56] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, SP 2014, IEEE Computer Society, 2014, pp. 590–604.

[57] OpenAI, Gpt-4 technical report (2023). arXiv:2303.08774.

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

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

[60] V. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, D. Q. Phung, Regvd: Revisiting graph neural networks for vulnerability detection, CoRR abs/2110.07317 (2021).

[61] W. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2655–2668.

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

[63] V. Yosifova, A. Tasheva, R. Trifonov, Predicting vulnerability type in common vulnerabilities and exposures (cve) database with machine learning classifiers, in: 2021 12th National Conference with International Participation (ELECTRONICA), IEEE, 2021, pp. 1–6.

[64] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, D. Phung, Vulrepair: a t5-based automated software vulnerability repair, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 935–947.

[65] A. Hovsepyan, R. Scandariato, W. Joosen, J. Walden, Software vulnerability prediction using text analysis techniques, in: Proceedings of the 4th international workshop on Security measurements and metrics, 2012, pp. 7–10.

[66] Y. Pang, X. Xue, A. S. Namin, Predicting vulnerable software components through n-gram analysis and statistical feature selection, in: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), IEEE, 2015, pp. 543–548.

[67] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, H. Jin, VulDeeLocator: A deep learning-based fine-grained vulnerability detector, IEEE Transactions on Dependable and Secure Computing 19 (4) (2022) 2821–2837.

[68] M. Allamanis, M. Brockschmidt, M. Khademi, 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, 2018.

[69] W. Ma, M. Zhao, E. O. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis,

M. Cordy, X. Xie, Y. L. Traon, Graphcode2vec: Generic code embedding via lexical and program dependence analyses, in: 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, ACM, pp. 524–536.

[70] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network, ACM Transactions on Software Engineering and Methodology 30 (3) (2021) 38:1–38:33.

[71] S. Min, M. Lewis, L. Zettlemoyer, H. Hajishirzi, Metaicl: Learning to learn in context, in: Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2022, pp. 2791–2809.

[72] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, L. Zettlemoyer, Rethinking the role of demonstrations: What makes in-context learning work?, in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 11048–11064.

[73] O. Rubin, J. Herzig, J. Berant, Learning to retrieve prompts for in-context learning, in: Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2022, pp. 2655–2671.

[74] N. Nashid, M. Sintaha, A. Mesbah, Retrieval-based prompt selection for code-related few-shot learning, in: Proceedings of the 45th International Conference on Software Engineering (ICSE'23), 2023.

[75] S. Ma, F. Thung, D. Lo, C. Sun, R. H. Deng, Vurle: Automatic vulnerability detection and repair by learning from examples, in: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22, Springer, 2017, pp. 229–246.

[76] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, Z. Chen, Pre-trained model-based automated software vulnerability repair: How far are we?, IEEE Transactions on Dependable and Secure Computing (2023).

Guilong Lu is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include vulnerability detection and LLM4SE.

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

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the Department of Information Science and Technology, 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, in particular 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://smartse.github.io/index.html.

Wenlong Pei is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include vulnerability detection and repair.

Zhilong Cai is currently pursuing the Master degree at the School of Information Science and Technology at Nantong University, and his main research focus is on cross-project software vulnerability detection.

Improved vulnerability detection capability of LLM using graph structure information.

In-context learning method to improve the vulnerability detection capability of LLM.

Evaluation on three real-world datasets.

Declaration of interests

☒ 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.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: