Exploration On Prompting LLM With Code-Specific Information For Vulnerability Detection

Zhihong Liu[a], Zezhou Yang[a], Qing Liao[a, b]
[a] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
[b] Peng Cheng Laboratory, Shenzhen, China
22s151090@stu.hit.edu.cn, yangzezhou@stu.hit.edu.cn, liaoqing@hit.edu.cn

***Abstract*—Software vulnerability detection is a software security analysis technique that aims to recognize possible code vulnerabilities and weaknesses. The majority of previous research has primarily concentrated on deep learning models.**

Recently, with the rapid development of large language models, researchers are exploring the use of GPT in the field of vulnerability detection. However, these works inadequately account for the features of vulnerability detection, lacking specific prompts and code-specific information. This paper primarily investigates the impact of prompts for GPT. We design the following prompts step by step: basic prompts, prompts with code-specific information, and chain-of-thought (CoT) prompts.

Firstly, we explore the performance using basic prompts in the field of vulnerability detection. Subsequently, we incorporate code-specific information into basic prompts, focusing primarily on similar code and data flow graph (DFG). The similar code is obtained through our designed code search algorithm, which takes into account both code semantic and structural information. Finally, we further introduce CoT prompts to inspire GPT's ability for gradually detecting vulnerability. We also conduct experiments to explore the impact of the position of code-specific information in the prompts and the suitable temperature value for the vulnerability detection task. Our experiment results demonstrate that, combined with code-specific information and CoT, GPT can detect vulnerabilities more effectively.

***Index Terms*—software vulnerability detection, large language models, prompt design, code search, CoT**

I. Introduction

Detecting code vulnerabilities is an essential technique in software security analysis. Its primary goal is to pinpoint potential weaknesses and vulnerabilities within software code, thereby minimizing the risk of malicious attacks and system failures. As software systems continue to expand in size and complexity, the importance of identifying vulnerabilities and security risks has become increasingly evident [1], [2]. However, developers face the formidable task of scrutinizing extensive codebases to uncover vulnerabilities. This process requires a high level of expertise and is both tedious and time-consuming due to the numerous subtle and elusive vulnerabilities present [3]–[7]. Consequently, the investigation of automated vulnerability detection using learning-based approaches is of great significance and holds considerable promise.

Previous research extensively utilizes deep learning models, such as convolutional neural networks (CNN) [8], [9] and recurrent neural networks (RNN) [10], [11], to extract vulnerability features [12]–[16]. Methods for extracting code features for vulnerability detection can be divided into graph-based and sequence-based representation techniques. Graph-based code representation involves parsing code into graph structures like abstract syntax trees (AST), control flow graphs (CFG), or data flow graphs (DFG) [17], which are then input into network models [18]. Sequence-based models, such as VulDeepecker [12] and SyseVR [14], examine code processes through system execution traces, function call sequences, and statement execution flow sequences. These models detect vulnerabilities by leveraging the patterns identified within these sequences.

Recently, the rapid advancement of large language models (LLMs) has led to significant breakthroughs across various fields. LLMs are trained on extensive datasets by auto-regressive objective using powerful computational resources. They typically consist of billions of model parameters, which allow them to capture complex patterns and exhibit general intelligence in a wide range of tasks. Some recent works also consider using GPT in the field of vulnerability detection [19], [20]. However, these works inadequately account for the features of vulnerability detection, lacking specific prompts and code-specific information [21]. The advantage of LLM is that it can analyze the code from different perspectives and solve complicated tasks by multiple reasoning steps [22]–[25]. This capability is often overlooked in existing research on vulnerability detection.

To comprehensively explore the potential of LLMs in the field of vulnerability detection, this paper mainly designs the following prompts step by step: basic prompts, prompts with code-specific information and chain-of-thought (CoT) prompts [26]. Firstly, we investigate the following basic prompts: basic prompts, role-based basic prompts, and role-based reverse-question prompts. Then, we analyse the DFG of code and similar code. The similar code refers to code obtained through a code search algorithm that closely resembles the code that needs to be tested. We integrate DFG and similar codes as code-specific information into prompts to form prompts with code-specific information. Finally, leveraging GPT's capability to memorize multiple rounds of conversations [25], this paper conducts experiments by integrating code-specific information into CoT prompts to detect vulnerabilities. Additionally, we explore the influence of the position of code-specific information in the prompts and the temperature parameter in GPT on vulnerability detection. We conduct a large number of experiments on Devign dataset [18] and Reveal dataset [13]. Our main contributions are outlined as follows:

2024 IEEE International Conference on Software Services Engineering (SSE) | 979-8-3503-6851-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/SSE62657.2024.00049

979-8-3503-6851-2/24/$31.00 ©2024 IEEE
DOI 10.1109/SSE62657.2024.00049

- We validate the effectiveness of GPT in the field of vulnerability detection and integrate code-specific information (i.e., DFG and similar codes) into basic prompts and CoT prompts to further improve the performance.
- We examine the impact of hyperparameter settings (i.e., temperature) in GPT and the position of code-specific information within the prompts. We also design a novel code search algorithm considering the semantic and structural information of code to select similar code as demonstrations.

The rest of this paper is organized as follows: Section 2 describes the background and related works. Section 3 introduces data pre-processing for our experiments. Section 4 outlines our prompt designs in the field of vulnerability detection. Section 5 describes the experimental setup, including experimental settings, datasets, baselines and evaluation metrics. Section 6 presents the experimental results and discussions. Section 7 describes the potential threats to validity of this work. Finally, in Section 8, we provide a summary of our work.

II. BACKGROUND AND RELATED WORKS

A. Large Language Models (LLMs)

Recently, large language models (LLMs) have demonstrated impressive effectiveness across a diverse range of tasks [25]. Large language models (LLMs) utilize the transformer architecture and are trained autoregressively, predicting the subsequent word based on the preceding context. These models process input sequences through several layers of self-attention mechanisms and feed-forward neural networks, generating a probability distribution for the next word. There are a number of LLMs that can be employed for a wide range of tasks, represented by OpenAI's GPT [25], [31], DeepMind's Chinchilla [32] and Tsinghua University's GLM-130B [33].

Due to the excellent performance and broad impact of GPT, we use GPT for vulnerability detection in our experiment.

B. Software Vulnerability Detection

Current vulnerability detection methods can be categorized into static analysis-based techniques and deep learning-based approaches.

Static analysis-based vulnerability detection involves directly analyzing the source code or binary code without executing the program to identify potential security vulnerabilities. This method relies on in-depth analysis of the code structure, data flow and control flow to detect potential security risks. Wang et al. [34] are the first to use abstract syntax trees (AST) for detecting vulnerabilities in Java. Peng et al. [35] combine N-gram and feature selection algorithms, defining features as continuous sequences of tokens in source files. Scandariato et al. [36] utilize bag-of-words to represent software components as a series of terms with associated frequencies, serving as features in vulnerability detection.

Methods based on deep learning can automatically capture vulnerability features [10], [11]. Russell et al. [13] utilize CNN to process lexical code representation. Vuldeepecker [12] employs bidirectional LSTM [37] to handle the code gadgets, a fine-grained code slice. Several works [14], [38] select LSTM as code encoders. Devign [18] and Reveal [39] employ the Gated Graph Neural Network (GGNN) [40], [41] to handle multiple directed graphs derived from source code.

C. Prompt Engineering

Designing effective prompts is the focus of research on GPT models across various tasks [42]. It primarily involves describing specific tasks through well-crafted text prompts and text structures [21]. Appropriate prompts enable GPT to achieve better results in code generation, code repair, and other fields [43]. Maddigan and Susnjak [44] investigate the impact of prompts on LLMs for visualizations. Liu et al. [45] propose various prompt designs for two tasks involving code generation. Liu et al. [46] investigate the effectiveness of prompt keywords affect image generation.

Several studies have explored the application of GPT in vulnerability detection. White et al. [47] explore various prompts to enhance requirements elicitation, rapid prototyping, code quality, deployment, and testing. Meanwhile, Cao et al. [19] develop prompt templates specifically for applying GPT in automated program repair. However, these works inadequately account for the features of vulnerability detection, lacking specific prompts and code-specific information. In Section 4, we conduct a comprehensive analysis of the impact of different prompts on the performance of GPT.

D. Code Search

The techniques for code search can be split into two categories: traditional methods and deep learning-based methods.

Traditional methods, also known as sparse retrieval, utilize information retrieval techniques to match keywords between queries and code snippets, focusing on leveraging semantic and structural information within the code [48]–[50]. For example, code can be represented as abstract syntax trees (AST), and tree-matching techniques can be applied for code search [51].

Deep learning-based methods typically involve embedding queries and code into vectors, then assessing the similarity between these vectors. The approaches include RNN based methods [52], CNN based methods [53], GNN based methods [54], and pre-training approaches [55].

In our study, we introduce a novel code search algorithm designed to identify similar code. This approach leverages both semantic information of the code and its structural representation (i.e., Abstract Syntax Tree, AST) to enhance BM-25 [56].

III. PRE-PROCESSING

This section introduces data pre-processing used in the experiment, as shown in figure 1.

A. Data Flow Graph Extraction

Data Flow Graph (DFG) is a structural representation used to describe the dependency relationships among data in computational tasks, and it is widely applied in the field of

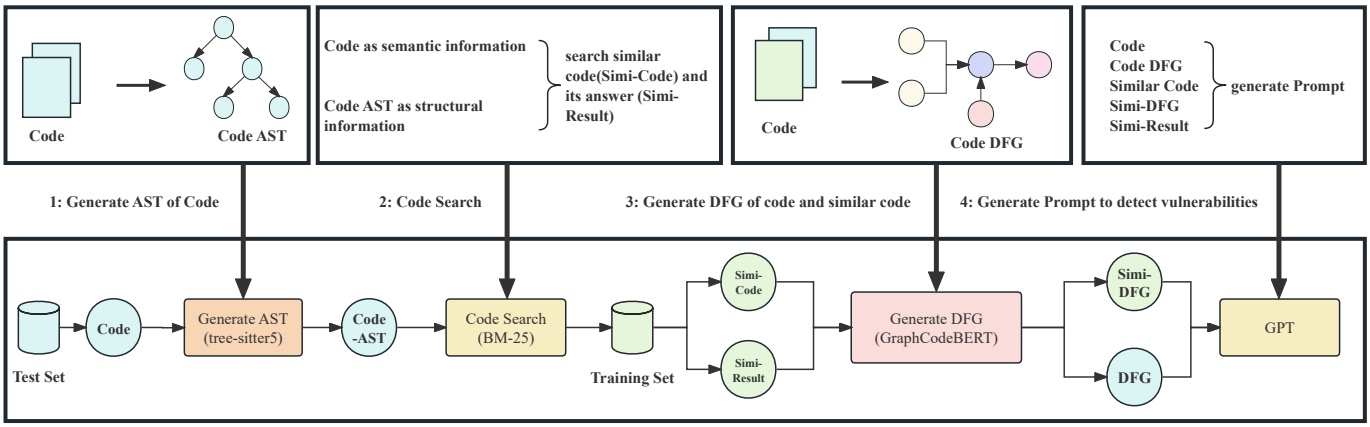

Fig. 1. Data Pre-processing. The bottom part of the figure demonstrates the process of prompt generation, while the top part demonstrates the details of data pre-processing. Initially, we generate the AST of code. Subsequently, we integrate semantic and structural information of the code into BM-25 to search for similar code and its answer. Then, we generate DFG of code and similar code. Finally, we assemble prompts using the aforementioned information to detect vulnerabilities.

vulnerability detection [18]. This structure aids in a deeper understanding and optimization of the computational process [57].

In this experiment, we consider incorporating DFG as code-specific information to assist GPT in vulnerability detection. Given the source code C. There are three steps to extract the data flow:

- Extract AST for the code C by python library tree-sitter.
- Detect variables based on the syntactic details and leaf nodes of the AST, represented as $V = v_1, v_2, \ldots, v_n$.
- Regard variables as nodes within the graph structure. For every variable in V, introduce a directed edge $\varepsilon =< v_i, v_j >$ into the dictionary D as a tuple (v_i, p_i, v_j, p_j), where p_i and p_j represent the positions of v_i and v_j. This tuple signifies the relationship where v_i is influenced by v_j.

B. Similar Code Collection

Previous research indicates that incorporating similar code information allows models to understand code more comprehensively [58], [59]. Therefore, we integrate similar code as code-specific information into prompts to detect vulnerabilities.

Given the test code that requires vulnerability detection, we use our code search algorithm to find similar code from the training set. In the code search algorithm, initially, we regard the code itself as semantic information and the AST of the code as structural information. The AST of the code is obtained using the Python library tree-sitter. Subsequently, we serialize these two types of information and integrate them into BM-25 [56] to search for similar code. BM-25, also known as Best Match 25, is a widely used ranking algorithm in information retrieval. Finally, the similar code, the information of similar code, and the test code are combined into prompts to detect vulnerabilities. Figure 1 shows the code search process mentioned above.

IV. Prompt Design

In this section, we present the prompts that we craft to augment the effectiveness of GPT in the field of vulnerability detection. We employ P_x to denote each prompt, where x represents the components of the prompt. Basic prompts and prompts with code-specific information need only one round of conversation; CoT prompts without similar code need two rounds of conversation, and CoT prompts with similar code need five rounds of conversation. Figure 2 shows the design of prompts.

A. Basic Prompts

Firstly, we employ the basic prompt, which involves a single round of conversation. To ensure clarity and prevent ambiguous answers, we require GPT to provide a definitive response (Yes or No).

- P_{b}: Is the following code buggy? Please answer Yes or No. [Code]

We further propose the role-based basic prompt to make GPT focus on the vulnerability detection:

- $P_{\text{r-b}}$: I want you to act as a vulnerability detection system. Is the following code buggy? Please answer Yes or No. [Code]

Finally, GPT may respond based on keywords rather than analyzing the code, such as answering ”No” because of the keyword ”buggy.” To address this, we design a reverse-question prompt to enhance the role-based basic prompt, testing whether GPT will make incorrect judgments due to specific keywords in the prompt.

- $P_{\text{r-r-b}}$: I want you to act as a vulnerability detection system. Is the following code correct? Please answer Yes or No. [Code]

B. Prompts With Code-specific Information

Basic prompts lack code-specific information, containing only the code itself. Therefore, we incorporate DFG and similar code as code-specific information into the basic prompts to

further improve the performance. This requires one round of conversation. Figure 1 shows the process of extracting code-specific information.

1) Prompts with DFG: Based on the basic prompts, we design the following prompts equipped with DFG. 'Code' represents the code we need to test, while 'DFG' represents the data flow graph of the code.

- $P_{\text{b-d}}$: There are the code and its data flow information, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Code]. [DFG].
- $P_{\text{d-b}}$: There are the code and its data flow information, and you will act upon them. [DFG]. Is the following code buggy? Please answer Yes or No. [Code].
- $P_{\text{r-b-d}}$: I want you to act as a vulnerability detection system. There are the code and its data flow information, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Code]. [DFG].
- $P_{\text{r-d-b}}$: I want you to act as a vulnerability detection system. There are the code and its data flow information, and you will act upon them. [DFG]. Is the following code buggy? Please answer Yes or No. [Code].

2) Prompts with similar code: We design the following prompts equipped with similar code based on basic prompts. 'Simi-Code' represents a similar code, while 'Simi-Result' represents the answer to a similar code.

- $P_{\text{b-s}}$: There are the code and its similar code and you will act upon them. Is the following code buggy? Please answer Yes or No.[Code]. [Simi-Code]. Similar code's answer is [Simi-Result].
- $P_{\text{s-b}}$: There are the code and its similar code, and you will act upon them. [Simi-Code]. Similar code's answer is [Simi-Result]. Is the following code buggy? Please answer Yes or No. [Code].
- $P_{\text{r-s-b}}$: I want you to act as a vulnerability detection system. There are the code and its similar code, and you will act upon them. [Simi-Code]. Similar code's answer is [Simi-Result]. Is the following code buggy? Please answer Yes or No. [Code].
- $P_{\text{r-b-s}}$: I want you to act as a vulnerability detection system. There are the code and its similar code and you will act upon them. Is the following code buggy? Please answer Yes or No.[Code]. [Simi-Code]. Similar code's answer is [Simi-Result].

3) Prompts with DFG and similar code: We design the following prompts equipped with DFG and similar code based on basic prompts.

- $P_{\text{r-s-b-d}}$: I want you to act as a vulnerability detection system. There are the code, its similar code and DFG, and you will act upon them. [Simi-Code]. Similar code's answer is [Simi-Result]. Is the following code buggy? Please answer Yes or No. [Code]. [DFG].
- $P_{\text{r-d-b-s}}$: I want you to act as a vulnerability detection system. There are the code, its similar code and DFG, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Code]. [DFG]. [Simi-Code]. Similar code's answer is [Simi-Result].

C. Chain-of-Thought Prompt

A notable advancement of GPT compared to other models lies in its ability to address intricate tasks through multiple rounds of reasoning. The pivotal factor behind this is the concept of CoT [26]. CoT involves several rounds of intermediate natural language reasoning that ultimately lead to the final answer. Therefore, we design CoT prompt in the field of vulnerability detection. Among them, CoT prompts without similar code require two rounds of conversation, while CoT prompts with similar code require five rounds of conversation.

1) CoT prompts without similar code: For CoT prompts without similar code, we design the following steps.

Step 1: Let GPT analyze the code first.

- $P_{1}^{(chain)}$: Please analyze the given code. [Code].

Step 2: Then ask GPT about code's vulnerability by previous conversations.

- $P_{\text{2,aux}}^{(chain)}$: I want you to act as a vulnerability detection system. There are the code and its data flow information, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Code]. [DFG].

The vulnerability detection process by prompt $P_{\text{2,r-b-d}}^{(chain)}$ is outlined above, as shown in figure 2. Step 2 can be replaced with any previously mentioned prompt that does not involve similar code.

2) CoT prompts with similar code: For CoT prompts with similar code, we design the following steps.

Step 1: Let GPT analyze the similar code first.

- $P_{1}^{(chain)}$: Please analyze the given code. [Simi-Code].

Step 2: Then ask GPT about similar code's vulnerability by previous conversations.

- $P_{\text{2,aux}}^{(chain)}$: I want you to act as a vulnerability detection system. There are the code and its data flow information, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Simi-Code]. [Simi-DFG].

Step 3: Inform GPT of the answer regarding similar code in the conversation. The purpose is to make GPT more attentive to errors made or relevant information during the detection of similar code.

- $P_{3}^{(chain)}$: The correct answer is: [Simi-Result]

Step 4: Let GPT analyze the code.

- $P_{4}^{(chain)}$: Please analyze the given code. [Code].

Step 5: Ask GPT about code's vulnerability by previous conversations.

- $P_{\text{5,aux}}^{(chain)}$: I want you to act as a vulnerability detection system. There are the code and its data flow information, and you will act upon them. Is the following code buggy? Please answer Yes or No. [Code]. [DFG].

Figure 2 shows the example of $P_{\text{r-s-b-d}}^{(chain)}$.

V. Experimental Setup

In our experiments, we run all tests on GPT-3.5-turbo.

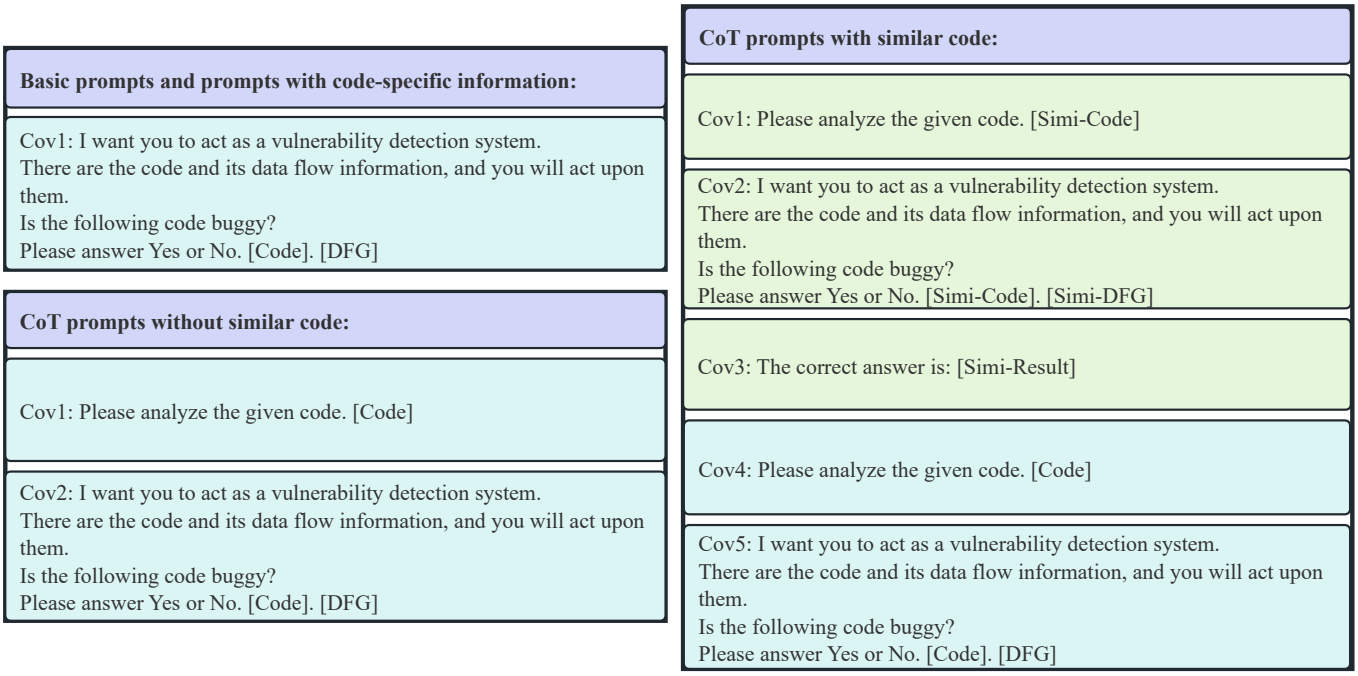

Fig. 2. Prompt Design. The upper-left of the figure demonstrates an example $P_{\text{r-b-d}}$ of the basic prompts and prompts with code-specific information, which need one round of conversation; The bottom-left of the figure demonstrates an example $P_{\text{r-b-d}}^{(chain)}$ of CoT prompts without similar code, which needs two rounds of conversation; The right of the figure demonstrates an example $P_{\text{r-s-b-d}}^{(chain)}$ of CoT prompts with similar code, which needs five rounds of conversation.

A. Research Questions

We want to answer the following research questions (RQ):

- **RQ1:** Can GPT detect vulnerabilities by using basic prompts, and what is the suitable temperature parameter for the performance for vulnerability detection?
- **RQ2:** Can prompts with code-specific information enhance the performance for vulnerability detection?
- **RQ3:** Can chain-of-thought prompt enhance the performance for vulnerability detection?
- **RQ4:** Does the position of code-specific information in the prompt affect the performance for vulnerability detection?

B. Datasets

In this work, we adopt Devign [18] and Reveal [13] datasets as our experimental datasets.

Devign is a dataset used for tasks related to software development, particularly for detecting vulnerabilities in source code. Collected by researchers from GitHub, it is widely utilized in both academic and industrial software engineering research. The Reveal dataset, on the other hand, is compiled from two open-source projects: the Linux Debian Kernel and Chromium.

The Devign and Reveal datasets are split into training, validation, and test sets. For this experiment, we use GPT for vulnerability detection, so we do not require the validation set. We search for similar code within the training set and evaluate GPT's performance using the test set. The statistics for both datasets are provided in Table I.

TABLE I
THE STATISTICS OF THE EVALUATION DATASET.

Dataset	Devign		Reveal	
	Training	**Test**	**Training**	**Test**
Vulnerable	7,078	879	801	104
Non-Vulnerable	8,526	1,064	10,371	1,296
Sum	15,604	1,943	11,172	1,400

C. Baseline

The baselines used for comparison in this experiment are SySeVR, VulDeePecker, and the model by Russell et al. These models are widely recognized for their effectiveness in vulnerability detection.

SySeVR [14] utilizes code statements, program dependencies, and program slicing as features. It employs a bidirectional recurrent neural network for detecting vulnerable code snippets.

VulDeePecker [12] employs word2vec to embed source code along with its data and control dependencies into a program slice. This embedding is subsequently inputted into a bidirectional LSTM-based neural network featuring an attention mechanism, aimed at identifying vulnerabilities.

Russell et al. [13] label source code and embed it into a matrix. They employ convolutional neural networks (CNNs), integrated learning techniques, and a random forest classifier for detecting code vulnerabilities.

TABLE II
PERFORMANCE OF GPT ON VULNERABILITY DETECTION BY DIFFERENT PROMPTS.

Metrics(%) \ Database		Devign				Reveal			
		Accuracy	**Precision**	**Recall**	**F1**	**Accuracy**	**Precision**	**Recall**	**F1**
Baseline	VulDeePecker	49.6	46.1	32.6	38.1	76.4	21.1	13.1	16.2
	Russell et al.	57.6	54.8	40.7	46.7	68.5	16.2	52.7	24.8
	SySeVR	47.9	46.1	58.8	51.7	74.3	40.1	24.9	30.7
Basic Prompt	P_{b}	38.9	40.5	74.3	52.3	17.1	7.9	**95.2**	14.6
	$P_{\text{r-b}}$	51.6	48.0	86.8	61.9	25.6	8.6	93.3	15.8
	$P_{\text{r-r-b}}$	52.4	41.3	12.1	18.6	**79.3**	6.9	14.4	9.7
P(DFG)	$P_{\text{b-d}}$	47.4	45.7	85.1	59.4	22.9	7.4	80.8	17.0
	$P_{\text{d-b}}$	48.6	44.8	59.1	50.9	41.0	6.9	55.8	12.2
	$P_{\text{r-b-d}}$	51.5	48.2	**95.0**	**64.2**	27.9	8.6	90.4	15.7
	$P_{\text{r-d-b}}$	50.7	47.6	85.0	61.2	28.4	8.0	82.7	14.8
P (Simi-Code)	$P_{\text{b-s}}$	49.8	45.2	51.8	48.2	49.0	7.5	52.0	13.0
	$P_{\text{s-b}}$	51.3	46.8	56.2	51.0	47.9	7.6	53.9	13.3
	$P_{\text{r-b-s}}$	52.1	47.7	57.3	51.9	52.0	8.2	53.9	14.3
	$P_{\text{r-s-b}}$	54.1	49.3	55.2	52.0	53.6	8.8	55.8	15.1
P (Simi-Code, DFG)	$P_{\text{r-s-b-d}}$	54.3	49.6	56.6	52.8	52.8	8.1	51.9	14.0
	$P_{\text{r-d-b-s}}$	53.4	48.8	58.8	53.4	50.6	8.8	60.6	15.4
P (Chain)	$P^{(chain)}_{\text{r-b}}$	59.3	53.7	75.1	62.7	48.0	9.1	66.4	16.0
	$P^{(chain)}_{\text{r-s-b}}$	**61.9**	**57.6**	60.6	59.2	60.8	11.3	62.5	**19.1**
	$P^{(chain)}_{\text{r-b-d}}$	60.7	55.9	61.7	58.6	58.9	10.3	58.7	17.5
	$P^{(chain)}_{\text{r-s-b-d}}$	61.1	56.7	59.7	58.3	**61.4**	**10.4**	54.8	17.4

TABLE III
VULNERABILITY DETECTION EFFECT IN DIFFERENT TEMPERATURES WHEN PROMPT IS $P_{\text{R-B}}$.

Metrics(%) \ Database		Devign				Reveal			
		Accuracy	**Precision**	**Recall**	**F1**	**Accuracy**	**Precision**	**Recall**	**F1**
Temperature	0	**51.6**	**48.0**	86.8	61.9	**25.6**	**8.6**	**93.3**	15.8
	0.3	50.1	47.1	85.0	60.7	25.2	7.8	83.7	16.9
	0.6	51.0	47.8	**89.4**	**62.3**	23.6	7.9	86.5	16.3
	0.9	50.7	47.6	87.5	61.7	24.8	7.9	85.6	14.4
	1	50.4	47.3	84.5	60.6	25.5	7.7	82.7	**17.7**

D. Temperature

The temperature in the GPT is an essential hyperparameter used to adjust the diversity of generated text. It is typically a positive number ranging between 0 and 1. Because $P_{\text{r-b}}$ achieves relatively good results on both accuracy and F1 in basic prompts. Therefore, we test the vulnerability detection performance at five temperature values: 0, 0.3, 0.6, 0.9, and 1 when prompt is $P_{\text{r-b}}$. After getting a suitable temperature parameter, we use it to conduct our subsequent experiments.

E. Evaluation Metrics

We use Accuracy (Acc), Precision (P), Recall (R) and F1 score (F1) to test the performance of GPT. They are commonly used to evaluate vulnerability detection methods.

In this experiment, Precision evaluates how accurately a model predicts samples as vulnerable.

$$P = \frac{TP}{TP + FP} \tag{1}$$

TP signifies the count of vulnerable code instances correctly identified by the model, while TN denotes the count of non-vulnerable code instances correctly identified by the model. FP refers to the count of non-vulnerable code instances incorrectly identified as vulnerable, and FN indicates the count of vulnerable code instances incorrectly identified as non-vulnerable by the model. These terms are consistently defined in the subsequent formulas in the same manner.

Accuracy refers to the proportion of correctly classified instances, indicating whether the code has vulnerabilities or not.

$$Acc = \frac{TP + TN}{TP + TN + FN + FP} \tag{2}$$

The F1 score integrates precision and recall into a unified metric by calculating their harmonic mean.

$$F1 = \frac{2 * TP}{2 * TP + FN + FP} \tag{3}$$

Recall assesses the model's capability to accurately identify positive instances. It measures the model's ability to capture or recall all instances of vulnerability.

$$R = \frac{TP}{TP + FN} \tag{4}$$

VI. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we show and analyze the experimental results. Table II shows the performance in two datasets under different prompts.

A. Performance Of Basic Prompts And Suitable Temperature Parameters (RQ1)

In experiments, Devign dataset has a relatively balanced distribution of positive and negative samples, while Reveal dataset predominantly consists of negative samples.

GPT's documentation mentions that specifying task roles for GPT can enhance its performance [25]. In Table II, it can be observed that in Devign dataset, the accuracy of $P_{\text{r-b}}$ is 12.7% higher compared to P_{b}. In Reveal dataset, the accuracy of $P_{\text{r-b}}$ is 8.5% higher compared to P_{b}. This indicates that specifying task roles for GPT can indeed improve the performance of vulnerability detection.

We also design $P_{\text{r-r-b}}$, where we replace "buggy" with "correct." The experimental results, as shown in Table II, indicate that in Reveal dataset, the accuracy of $P_{\text{r-r-b}}$ is 53.7% higher than $P_{\text{r-b}}$. The reason for this is that the number of non-vulnerable codes in Reveal dataset is 1,296, while the number of vulnerable codes is 104. It shows that GPT may guess answers based on the words in the prompts.

In conclusion, $P_{\text{r-b}}$ achieves relatively good results on both accuracy and F1. In subsequent experiments, we will conduct experiments based on $P_{\text{r-b}}$.

Then, we study the performance of $P_{\text{r-b}}$ under five different temperature parameters. The experiment results are shown in Table III. In Reveal dataset, we observe that GPT achieves the best performance of vulnerability detection with accuracy of 25.6%, precision of 8.6% and recall of 93.3% when the temperature value is 0. In Devign dataset, GPT also achieves good performance with accuracy of 51.6% and precision of 48.0%, when the temperature value is 0. Therefore, the temperature in the subsequent experiment is set as 0.

> **Answer to RQ1:** GPT is able to detect vulnerabilities by basic prompts. Specifically, incorporating task roles into the prompts enhances GPT's performance in vulnerability detection. GPT has better vulnerability detection effectiveness when the temperature value is 0.

B. Prompt With Code-specific Information (RQ2)

In this RQ, we study the performance of incorporating code-specific information into the prompts, including DFG and similar code.

As shown in Table II, in Devign dataset, $P_{\text{r-s-b-d}}$ achieves the best performance with accuracy of 54.3%, which is 2.7% higher than $P_{\text{r-b}}$. The utilization of similar code show better performance compared to DFG. Specifically, the best-performing $P_{\text{r-s-b}}$ using similar code outperforms the best-performing $P_{\text{r-b-d}}$ using DFG by 2.6%.

In Reveal dataset, $P_{\text{r-s-b}}$ achieves the best performance with accuracy of 53.6%, which is 28% higher than $P_{\text{r-b}}$. Furthermore, our experiments indicate that using similar code can further enhance performance compared to DFG. The worst-performing similar code, $P_{\text{s-b}}$, outperforms the best-performing DFG, $P_{\text{d-b}}$, by 6.9%.

> **Answer to RQ2:** Incorporating code-specific information, such as DFG and similar code, into the prompt is more effective than using basic prompts. Furthermore, the overall effectiveness of similar code surpasses that of DFG.

C. Chain-of-Thought Prompt (RQ3)

In this RQ, we study the performance of CoT prompt in the field of vulnerability detection.

The experimental results, as shown in Table II, indicate that in Devign dataset, $P^{(chain)}_{\text{r-s-b}}$ achieves the best performance with accuracy of 61.9%, surpassing $P_{\text{r-s-b}}$ by 7.8% and $P^{(chain)}_{\text{r-b}}$ by 2.6%. Furthermore, in Reveal dataset, $P^{(chain)}_{\text{r-s-b-d}}$ achieves the best performance with accuracy of 61.4%, surpassing $P_{\text{r-s-b-d}}$ by 8.6% and $P^{(chain)}_{\text{r-b}}$ by 13.4%.

The experimental data indicates that both CoT and code-specific information effectively enhance the performance of GPT in the field of vulnerability detection. Overall, CoT prompts achieve the best results on both datasets, surpassing the basic prompts and prompts with code-specific information. Especially in Reveal dataset, the influence of CoT is more pronounced. The accuracy of $P^{(chain)}_{\text{r-b-d}}$ is 31% higher than $P_{\text{r-b-d}}$, and the accuracy of $P^{(chain)}_{\text{r-s-b}}$ is 7.2% higher than $P_{\text{r-s-b}}$. This suggests that the CoT further improves the effectiveness of GPT in the field of vulnerability detection.

> **Answer to RQ3:** Chain-of-thought prompts can enhance the effectiveness of GPT in the field of vulnerability detection, achieving the best results on both datasets in the experiments.

D. Impacts of Position (RQ4)

Due to the various possible positions of code-specific information in prompts, we also investigate how the position of code-specific information affects the performance of GPT in the field of vulnerability detection. We conduct experiments with two different sequences by placing [Simi-Code] or [DFG] before or after the source code of basic prompts P_{b} and role-based basic prompts $P_{\text{r-b}}$.

The experimental results are shown in Table II. For similar code, we observe that placing it before the source code of basic prompts and role-based basic prompts generally shows better overall performance in the field of vulnerability detection compared to placing it after the source code of basic prompts and role-based basic prompts. As observed in Devign dataset, $P_{\text{s-b}}$ outperforms $P_{\text{b-s}}$ and $P_{\text{r-s-b}}$ outperforms $P_{\text{r-b-s}}$. In Reveal dataset, $P_{\text{r-s-b}}$ outperforms $P_{\text{r-b-s}}$.

Similarly, for DFG, placing it before the source code of basic prompts and role-based basic prompts generally shows better performance. In Devign dataset, $P_{\text{d-b}}$ outperforms $P_{\text{b-d}}$. In Reveal dataset, $P_{\text{d-b}}$ outperforms $P_{\text{b-d}}$ and $P_{\text{r-d-b}}$ outperforms $P_{\text{r-b-d}}$.

Answer to RQ4: Placing similar codes or DFG information in front of the source code of basic prompts and role-based basic prompts can achieve better performance in the field of vulnerability detection overall.

VII. Threats To Validity

The limited types of LLM: The experimental results presented in this paper are specific to ChatGPT 3.5. The conclusions drawn here may not apply to other LLMs.

The limited number of experimental datasets: We evaluate the performance of GPT using two datasets: Devign and Reveal. Although these datasets encompass various types of open-source projects, we plan to incorporate additional datasets in future investigations to evaluate the effectiveness of our methods further.

Different Programming Languages: Our experiments are exclusively conducted on C/C++ datasets, without consideration for datasets from other programming languages like Java and Python. In the future, we intend to explore a wider range of dataset selections, encompassing more programming languages, for a comprehensive evaluation of our methods.

VIII. Conclusion

This paper explores the performance of GPT in the field of vulnerability detection. We design the following prompts step by step: basic prompts, prompts with code-specific information (DFG and similar code), and chain-of-thought (CoT) prompts. Then, we investigate the performance under different temperature parameters and explore the influence of position of code-specific information in the prompts. We also introduces a method of searching similar code based on structural structure (AST) and semantic structure of the code.

Through experiments with a variety of prompts on two datasets, Devign and Reveal, we identify the most effective prompt in the field of vulnerability detection, which is chain-of-thought prompts with code-specific information. In future research, using better large language models and providing more comprehensive semantic or syntactic information related to code may be a new breakthrough.

References

[1] "The exactis breach: 5 things you need to know." 2000. [Online]. Available: https://blog.infoarmor.com/individuals-and-families/ the-exactis-breach-5-things-you-need-to-know

[2] "Wannacry ransomware attack." 2000. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry ransomware attack

[3] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 480–491.

[4] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 72–82.

[5] D. L. Heine and M. S. Lam, "Static detection of leaks in polymorphic containers," in 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 252–261.

[6] D. Kroening and M. Tautschnig, "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, ser. Lecture Notes in Computer Science, E. 'Abrah ' am and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391.

[7] X. Ji, J. Yang, J. Xu, L. Feng, and X. Li, "Interprocedural path-sensitive resource leaks detection for C programs," in Proceedings of the Fourth Asia-Pacific Symposium on Internetware, Internetware 2012, QingDao, China, October 30-31, 2012, H. Mei, J. Lv, Q. Wang, and L. Liu, Eds. ACM, 2012, pp. 19:1–19:9.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proc. IEEE, vol. 86, no. 11, pp. 22782324, 1998.

[10] S. O. Alhumoud and A. A. A. Wazrah, "Arabic sentiment analysis using recurrent neural networks: a review," Artif. Intell. Rev., vol. 55, no. 1, pp. 707–748, 2022.

[11] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model," in 2012 IEEE Spoken Language Technology Workshop (SLT), Miami, FL, USA, December 2-5, 2012. IEEE, 2012, pp. 234239.

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

[13] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, Eds. IEEE, 2018, pp. 757–762.

[14] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," IEEE Trans. Dependable Secur. Comput., vol. 19, no. 4, pp. 2244–2258, 2022.

[15] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," CoRR, vol. abs/1708.02368, 2017.

[16] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An imageinspired scalable vulnerability detection system," in 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, 2022, pp. 2365–2376.

[17] F. Yamaguchi, N. Golde, D. Arp, and 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.

[18] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 2019, pp. 10 197–10 207.

[19] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-Chi Cheung. 2023. A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. arXiv Preprint https://arxiv.org/abs/2304.08191 (2023).

[20] Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making Pre-trained Language Models Better Few-shot Learners. In ACL/IJCNLP (1). 3816–3830.

[21] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. ACM Comput. Surv. 55, 9 (2023), 195:1–195:35.

[22] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI Blog, 1(8), 9.

[23] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., et al. (2019). RoBERTa: A robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692.

[24] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., et al. (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461.

[25] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In NeurIPS, Vol. 33. 1877–1901.

[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In NeurIPS, Vol. 35. 24824–24837.

[27] R. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA), Dec. 2018, pp. 757–762.

[28] Ramos, J. (2003). Using TF-IDF to determine word relevance in document queries. In Proceedings of the first instructional conference on machine learning (Vol. 242, No. 1, pp. 133-142).

[29] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with AI," 2018, arXiv:1808.09897.[Online]. Available: http://arxiv.org/abs/1808.09897

[30] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv Preprint (2023). https://arxiv.org/abs/2303.18223

[31] OpenAI. 2023. GPT-4 Technical Report. arXiv Preprint (2023). https://arxiv.org/ abs/2303.08774

[32] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv Preprint (2022). https://arxiv.org/abs/2203.15556

[33] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Zhiyuan Liu, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. GLM-130B: An Open Bilingual Pre-trained Model. In ICLR. https://openreview.net/forum?id=-Aw0rrrPUF

[34] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In ICSE. 297–308.

[35] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. 2015. Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection. In ICMLA. 543–548.

[36] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. IEEE Trans. Software Eng. 40, 10 (2014), 993–1006.

[37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[38] X. Gong, Z. Xing, X. Li, Z. Feng, and Z. Han, "Joint prediction of multiple vulnerability characteristics through multi-task learning," in 24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019. IEEE, 2019, pp. 31–40.

[39] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" CoRR, vol. abs/2009.07235, 2020.

[40] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 2, 2005, pp. 729–734 vol. 2.

[41] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in 4th International Conference on Learning Representations, ICLR 2016, 2016.

[42] Silviu Pitis, Michael R. Zhang, Andrew Wang, and Jimmy Ba. 2023. Boosted Prompt Ensembles for Large Language Models. arXiv Preprint (2023). https: //arxiv.org/abs/2304.05970

[43] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In ICML, Vol. 139. 12697–12706.

[44] Paula Maddigan and Teo Susnjak. 2023. Chat2VIS: Generating Data Visualisations via Natural Language using ChatGPT, Codex and GPT-3 Large Language Models. arXiv Preprint (2023). https://arxiv.org/abs/2302.02094

[45] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. arXiv Preprint (2023). https://arxiv.org/abs/2305.08360

[46] Vivian Liu and Lydia B. Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In CHI. 384:1–384:23.

[47] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. arXiv Preprint (2023). https: //arxiv.org/abs/2303.07839

[48] Hui-Hui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In Proceedings of the 26th International Joint Conference on Artificial Intelligence. AAAI Press, 3034–3040.

[49] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). 141–151.

[50] Smith, John, & Johnson, Emily. (2021). Syntactic-based Code Search: Techniques and Applications. IEEE Transactions on Software Engineering, 45(3), 312-325. DOI: 10.1109/TSE.2021.12345678.

[51] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 783–794.

[52] Gu, X., Zhang, H., & Kim, S. 2018. Deep code search. In Proceedings of the International Conference on Software Engineering. ICSE. 933–944.

[53] Li, W., Qin, H., Yan, S., Shen, B., & Chen, Y. 2020. Learning code-query interaction for enhancing code searches. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution. ICSME. 115–126.

[54] Wan, Y., Shu, J., Sui, Y., Xu, G., Zhao, Z., Wu, J., & Yu, P. S. 2019. Multi-modal attention network learning for semantic source code retrieval. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE. 13–25.

[55] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. 2020. CodeBERT: A pre-trained model for programming and natural languages. In Proceedings of the Conference on Empirical Methods in Natural Language Processing. EMNLP. 1536–1547.

[56] Robertson, S., & Walker, S. (1994). Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval (pp. 232-241).

[57] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode-BERT: Pre-training Code Representations with Data Flow. In ICLR. https://openreview.net/pdf?id=jLoC4ez43PZ

[58] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? arXiv preprint arXiv:2202.12837 (2022).

[59] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE.