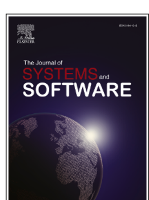
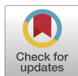
The Journal of Systems and Software 235 (2026) 112756

Contents lists available at ScienceDirect

The Journal of Systems & Software

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





RLV: LLM-based vulnerability detection by retrieving and refining contextual information

Fangcheng Qiu [a], Zhongxin Liu [b,*], Bingde Hu [c,d], Zhengong Cai [e], Lingfeng Bao [b], Xinyu Wang [c,*]

[a] Department of Computer Science and Technology, Zhejiang Gongshang University, Hangzhou, 310018, Zhejiang, China
[b] State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, 310027, Zhejiang, China
[c] College of Computer Science and Technology, Zhejiang University, Hangzhou, 310027, Zhejiang, China
[d] Bangsun Technology, Hangzhou, 310007, Zhejiang, China
[e] School of Software Technology, Zhejiang University, Ningbo, 315103, Zhejiang, China

ARTICLE INFO

Editor: Dario Di Nucci

Keywords:
Vulnerability detection
Large language models
Prompt engineering

ABSTRACT

Vulnerability detection plays a critical role in ensuring software quality during the processes of software development and maintenance. Automated vulnerability detection methods have been proposed to reduce the consumption of human and material resources. From traditional machine learning-based approaches to deep learning-based approaches, vulnerability detection techniques have continuously evolved and improved. Recently, Large Language Models (LLMs) have been increasingly applied to vulnerability detection. However, deep learning-based approaches and LLM-based approaches suffer from two main problems: (1) They suffer from poor generalization capabilities, which limit their performance in real-world scenarios. (2) They lack accurate contextual information of the target function, which hinders their ability to correctly understand the target function. To tackle these problems, in this paper, we propose a novel vulnerability detection approach, named RLV (**R**etrieving&Refining Contextual Information for **L**LM-based **V**ulnerability Detection), an LLM-based approach that enhances vulnerability detection by integrating project-level contextual information into the analysis process. RLV emulates how programmers reason about code, enabling the LLM to retrieve and refine relevant semantic context from the project repository to better understand the target function. Besides, RLV guides the LLM via effective prompts, avoiding task-specific training and enhancing its practicality in real-world scenarios. We conduct experiments on two vulnerability datasets with a total of 30,436 vulnerable functions and 306,269 non-vulnerable functions. The experimental results demonstrate that our approach achieves state-of-the-art performance. Moreover, our approach achieves a 26.83% improvement in terms of F_1-score over state-of-the-art baselines when tested on unseen projects.

1. Introduction

Software vulnerabilities in software systems are pervasive in cyberspace, leading to numerous system attacks and data breaches (Jang-Jaccard and Nepal, 2014). Detecting vulnerabilities has been a critical challenge in the field of software security for a long time. Automated vulnerability detection methods have been proposed and have undergone rapid development to reduce the consumption of human and material resources. From traditional machine learning approaches (Scandariato et al., 2014; Neuhaus et al., 2007) to deep learning-based techniques (Zhou et al., 2019; Chakraborty et al., 2021; Li et al., 2021b; Russell et al., 2018; Duan et al., 2019), the effectiveness of detection has continued to improve. Recently, Large Language Models (LLMs) pre-trained on extensive corpora have shown exceptional performance in a wide range of tasks spanning natural language processing and software engineering domains (Liu et al., 2024a,b; Deng et al., 2023; Zhang et al., 2023a, 2024b). Due to the powerful code comprehension capabilities of LLMs, they have been increasingly applied to vulnerability detection. Currently, LLM-based vulnerability detection models can be broadly categorized into two types: (1) Fine-tuning-based (Kenton and Toutanova, 2019), which involves updating the parameters of LLMs using labeled datasets. (2) Prompt engineering-based (Brown et al., 2020), which creates tailored prompts to guide LLMs in generating relevant responses without modifying their parameters. The first approach

* Corresponding authors.
E-mail addresses: fangchengq@zjgsu.edu.cn (F. Qiu), liu_zx@zju.edu.cn (Z. Liu), tonyhu@zju.edu.cn (B. Hu), xcstcaizg@zju.edu.cn (Z. Cai), lingfengbao@zju.edu.cn (L. Bao), wangxinyu@zju.edu.cn (X. Wang).

https://doi.org/10.1016/j.jss.2025.112756
Received 1 August 2025; Received in revised form 23 November 2025; Accepted 20 December 2025
Available online 2 January 2026
0164-1212/© 2026 Elsevier Inc. All rights reserved, including those for text and data mining, AI training, and similar technologies.

primarily utilizes lightweight LLMs (with less than 1 billion parameters), while the latter approach utilizes larger LLMs (with over 1 billion parameters). Fine-tuning large LLMs requires immense computational power and memory, making the process prohibitively expensive for most organizations. Besides, many large LLMs (e.g., OpenAI's GPT-4, Anthropic's Claude) do not provide access to their model weights, making fine-tuning impossible.

However, both deep learning-based approaches and LLM-based approaches have inherent limitations. **(1) They exhibit poor generalization capabilities. (2) They lack accurate contextual information, which prevents them from correctly understanding functions and tends to result in unsatisfactory performance.** Supervised learning-based approaches (including deep learning-based approaches and LLM fine-tuning-based approaches) often struggle to adapt to cross-project application. Since they primarily learn vulnerability patterns from training datasets, their detection performance largely depends on the quality and diversity of the training data. In practice, their performance drops significantly when applied to new projects with unseen coding styles or vulnerability patterns, unless additional fine-tuning is conducted on the target data (Chakraborty et al., 2021; Chen et al., 2023). Currently, the collection of vulnerability datasets faces many challenges, such as accurate data labeling and precise identification of vulnerable functions. Additionally, the amount of available data remains insufficient (Croft et al., 2023; Ding et al., 2024). Even in relatively large datasets like Big-Vul dataset (Fan et al., 2020) and DiverseVul dataset (Chen et al., 2023), the number of vulnerabilities barely exceeds 10,000. Moreover, most existing datasets include only single functions, thus missing the contextual information needed for comprehensive code understanding. In this case, these approaches are unable to understand the purpose of the function, the meaning of the functions it calls, and the significance of user-defined variables, ultimately failing to grasp the overall semantics of the source code. As a result, in practical applications, the detection results are often suboptimal. LLM Prompt-based approaches, regardless of whether they utilize zero-shot, few-shot, or chain-of-thought (CoT) prompting strategies (Fu et al., 2023; Zhou et al., 2024; Yin et al., 2024), have demonstrated suboptimal performance in both within-project and cross-project vulnerability detection settings. Despite leveraging large LLMs with strong code understanding abilities and rich prior knowledge, these models fail to fully comprehend program semantics when exposed solely to isolated functions lacking contextual information (Fang et al., 2024). As a result, they cannot achieve satisfactory performance (Fu et al., 2023; Yin et al., 2024).

In this study, we propose a novel vulnerability detection approach, named RLV (**R**etrieving&Refining Contextual Information for **L**LM-based **V**ulnerability Detection), which is based on LLMs and leverages project repository knowledge. For the first problem that current approaches exhibit poor generalization capabilities (Chen et al., 2023), RLV leverages large LLM and utilizes effective prompts to guide the LLM to generate outputs, eliminating the need for training on specific datasets. As a result, RLV can achieve robust performance even when applied to new projects, making it more adaptable and practical for real-world scenarios. For the second problem, we emulate the process programmers use to understand source code by utilizing the LLM to retrieve the information required to comprehend the target function from its project repository, constructing it as a knowledge base. Then we construct an appropriate prompt combing the target function and the knowledge base. Finally, this prompt is input into the LLM to obtain detection results. Specifically, RLV retrieves essential contextual information–namely, the callees, callers, and relevant data types of the target function–from the project repository (Ko and Uttl, 2003; Tao et al., 2012). Incorporating this contextual knowledge allows the LLM to develop a deeper semantic understanding of the target function, thereby enhancing its vulnerability detection capability.

To demonstrate the effectiveness of our approach, we evaluate RLV on two vulnerability datasets, i.e., FFmpeg + QEMU (Zhou et al., 2019), DiverseVul (Chen et al., 2023). We measure the performance of RLV using accuracy, precision, recall, and F_1-score. The experimental results demonstrate The experimental results demonstrate that our approach achieves state-of-the-art performance under the setting of randomly selected test sets. Moreover, our approach achieves a 26.83% improvement in terms of F_1-score over state-of-the-art baselines when tested on unseen projects. We open-source our replication package (rep, 2025).

```
1  uint32_t virtio_config_readw(VirtIODevice *vdev, uint32_t addr)
2  {
3      VirtioDeviceClass *k = VIRTIO_DEVICE_GET_CLASS(vdev);
4      uint16_t val;
5      k->get_config(vdev, vdev->config);
6      if (addr > (vdev->config_len - sizeof(val)))
7          return (uint32_t)-1;
8      val = lduw_p(vdev->config + addr);
9      return val;
10 }
```

Fig. 1. A vulnerable function in QEMU.

In summary, this paper makes the following contributions:

- We propose a novel function-level vulnerability detection model which leverages project repository knowledge.
- We utilize LLMs to retrieve repository information from the perspective of code comprehension and further refine the retrieved data, thereby enhancing the quality of the knowledge base.
- We perform extensive experiments to evaluate RLV. Our results demonstrate that RLV exhibits better generalization compared to state-of-the-art vulnerability detection tools.
- We demonstrate that prompt engineering-based methods are also feasible for function-level vulnerability detection.

The organization of this paper is as follows. Section 2 describes the motivation of RLV. Section 3 describes the background of this work. Section 4 introduces the detailed design of our approach. Section 5 describes our experiment setup. Section 6 reports the experimental results. Section 7 discusses further experiments. Section 8 shows threats to validity. Section 9 describes the related work. We conclude the paper and discuss the future work in Section 10.

2. Motivation

Fig. 1 shows a function collected from the QEMU project. This function contains two vulnerabilities. In detail, the first vulnerability is CWE-190: Integer Overflow or Wraparound. Both $vdec - config_len$ and $sizeof(vul)$ are unsigned integers. If $vdev-> config_len - sizeof(val)$ results in a negative value, it triggers an unsigned integer underflow, causing the result to wrap around and become a very large positive number. As a result, the condition $addr > (vdev-> config_len - sizeof(val))$ may be incorrectly evaluated as false. The second vulnerability is CWE-476: NULL Pointer Dereference. If $vdev-> config_len$ is 0, $vdev-> config$ could be NULL. Calling the get_config function in this scenario may lead to a null pointer dereference vulnerability, which causes the program to crash.

From the above example, we have the following observations: (1) The function under detection contains numerous user-defined functions, data types, and variables. The vulnerability is closely related to the user-defined data involved. (2) It is not possible to determine whether the function contains vulnerabilities based solely on the information provided within the function itself. To detect the vulnerability in the function shown in Fig. 1, existing function-level vulnerability detection methods are limited to analyzing the information contained within the function itself. The value of sizeof(val) is 2, but existing approaches lack knowledge of the data type $VirtIODevice$ of $vdev$. Therefore, they are unable to determine the range of values for $vdev-> config_len$ and cannot ascertain whether $addr > (vdev-> config_len - sizeof(val))$ could be less than zero. As a result, they cannot identify the potential unsigned

```
1  struct VirtIODevice
2  {
3    DeviceState parent_obj;
4    const char *name;
5    uint8_t status;
6    uint8_t isr;
7    uint16_t queue_sel;
8    /**
9     * These fields represent a set of VirtIO features at various
10    * levels of the stack. @host_features indicates the complete
11    * feature set the VirtIO device can offer to the driver.
12    * @guest_features indicates which features the VirtIO driver has
13    * selected by writing to the feature register. Finally
14    * @backend_features represents everything supported by the
15    * backend (e.g. vhost) and could potentially be a subset of the
16    * total feature set offered by QEMU.
17    */
18   size_t config_len;
19   void *config;
20   uint16_t config_vector;
21   …
22 };
```

Fig. 2. Source code of VirtIODevice Struct.

```
1  static void get_config(VirtIODevice *vdev, uint8_t *config_data)
2  {
3    VirtIOSerial *vser = VIRTIO_SERIAL(vdev);
4    struct virtio_console_config *config =
5      (struct virtio_console_config *)config_data;
6
7    config->cols = 0;
8    config->rows = 0;
9    config->max_nr_ports = virtio_tswap32(vdev,
10                               vser->serial.max_virtserial_ports);
11 }
```

Fig. 3. Source code of get_config Function.

integer underflow vulnerability that may occur if $vdev->config_len - sizeof(val)$ evaluates to a negative value. Furthermore, these methods lack contextual information about the $get_config()$ function, making it impossible to determine whether a null pointer check mechanism is implemented. Consequently, they are unable to assess whether passing a null pointer to this function would lead to a vulnerability.

The first observation mentioned above motivates us to accurately understand the semantic information of source code. Therefore, given a target function, we first extract information about user-defined functions and data types in the function. Next, we locate the definitions of these elements in the project repository to gain a deeper understanding of the semantic information of the source code. The second observation motivates us to analyze the additional contextual information surrounding the function under analysis. Therefore, we not only extract the callee functions within the target function but also identify its caller functions to construct project repository knowledge. Subsequently, we utilize LLM to further analyze the callee functions and caller functions of the target function. In this way, our approach can detect vulnerabilities that need additional context more effectively. Based on these techniques, for the function in Fig. 1, our approach analyzes the $VirtIODevice$ data type used within the function. By locating the definition of this structure (Fig. 2), we add the data type body in the project repository knowledge base. The LLM identifies $VirtIODevice$ as representing a virtualized I/O device within the VirtIO framework, used to simulate various devices in virtualized environments. It recognizes that $config_len$ can be less than 2, and that $vdev->config_len - sizeof(val)$ can lead to an unsigned integer underflow vulnerability. Additionally, the LLM understands the semantics of $VirtIODevice$, noting that certain devices may not require configuration space, meaning $config_len$ could be zero, rendering $config$ as NULL. By analyzing the callee $get_config()$ (Fig. 3), it is further identified that the function lacks a null pointer check mechanism, which could result in a null pointer dereference vulnerability and cause the program to crash.

3. Background

Our approach leverages LLMs to enhance vulnerability detection. In this section, we introduce the background knowledge of the related techniques.

3.1. Deep learning for vulnerability detection

Recently, numerous deep learning-based methods have been developed for vulnerability detection, primarily leveraging representation learning techniques to identify patterns associated with vulnerabilities. These approaches can be categorized into sequence-based approaches (Li et al., 2018; Russell et al., 2018; Li et al., 2021b) and graph-based approaches (Zhou et al., 2019; Chakraborty et al., 2021; Li et al., 2021a). Sequence-based approaches treat source code as input and focus on learning representations that help determine whether a given code snippet is vulnerable. Graph-based methods extract structured representations from source code, such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), and Code Property Graphs (CPGs) (Yamaguchi et al., 2014). These representations are then processed using Graph Neural Networks (GNNs) to learn graph-level features for effective vulnerability detection.

3.2. LLMs for vulnerability detection

Currently, LLMs are widely applied across various fields and have achieved impressive results. They have also been utilized in vulnerability detection, primarily through two approaches. The first involves fine-tuning LLMs for vulnerability detection, where the model is adapted using labeled data. The second approach focuses on constructing prompts that are fed into LLMs to obtain relevant feedback.

3.2.1. Fine-tuning-based approaches

Fine-tuning is an approach to transfer learning in which the parameters of a pre-trained neural network model are trained on new data (Yosinski et al., 2014). This method involves supplying labeled code samples, which specify whether the code is vulnerable or not, as input for training. Through supervised learning, the parameters of the model are updated and optimized based on these labeled datasets to enhance its ability to detect vulnerabilities.

3.2.2. Prompt-based approaches

Prompt engineering is the process of structuring an instruction that can be interpreted and understood by a generative artificial intelligence model (Brown et al., 2020). Theses approaches use source code to construct a prompt tailored for vulnerability detection. Then they feeds this prompt into the LLMs. Finally, the LLMs generate a response to detect whether the source code is vulnerable or not. There are three main prompting methods for utilizing LLMs in vulnerability detection. **Zero-shot Prompting**: Researchers design effective prompts to guide LLMs in performing vulnerability detection without requiring any labeled training data, relying solely on the model's pre-trained knowledge (Zhang et al., 2024a). **Few-shot Prompting** (Fu et al., 2023; Yin et al., 2024): In this approach, a few input-output example pairs, consisting of code snippets and their corresponding labels, are provided to LLMs as additional context to guide their predictions. **Retrieval-Augmented Prompting**: This technique enhances few-shot prompting by retrieving similar labeled data samples from the training dataset based on the test input. These retrieved samples are then used as examples to guide the LLMs' predictions on the test data (Wen et al., 2024b).

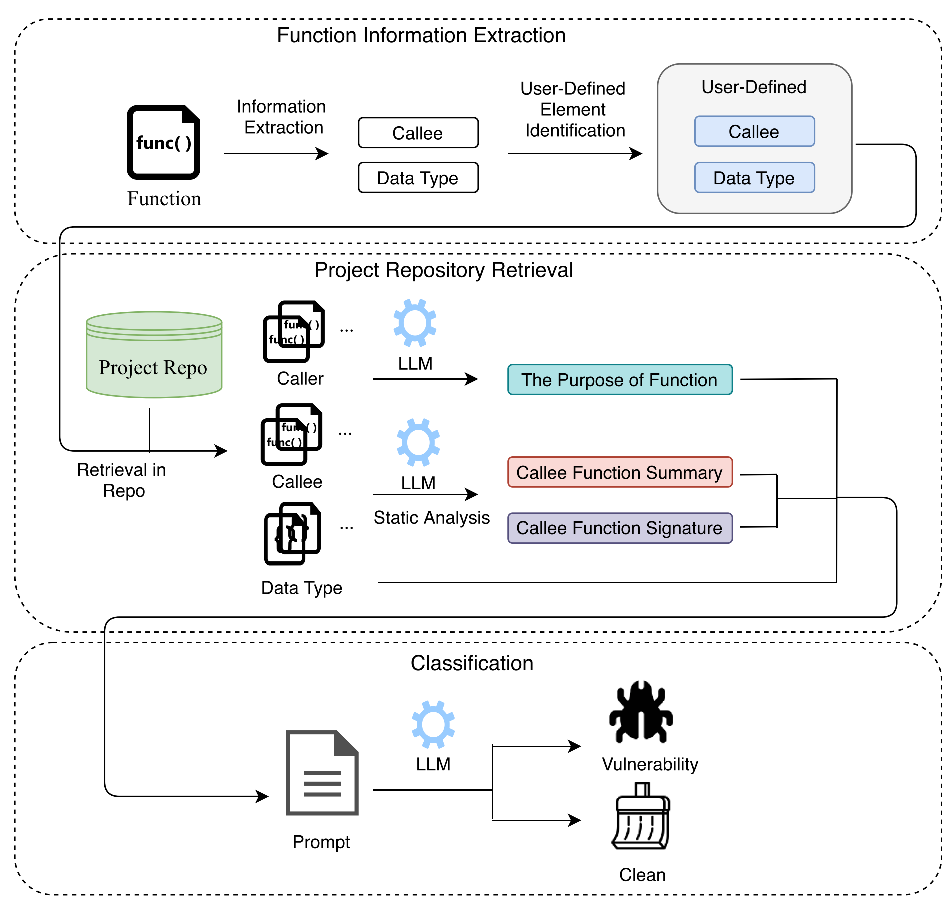


Fig. 4. Overall Framework of RLV.

4. Approach

We propose a novel model, named RLV (REPOSITORY-LLM-BASED VULNERABILITY DETECTION, to detect vulnerable functions via project repository knowledge. As shown in Fig. 4, it consists of three stages, i.e., function information extraction, project repository retrieval, and classification. The function information extraction stage extracts necessary information from the target function. The project repository retrieval stage retrieves related information from the project repository. The classification stage predicts whether the function contains vulnerabilities.

4.1. Function information extraction

Given a function, this stage aims to extract elements that contain essential semantic information, i.e., callees and data types. Considering the process a programmer follows when encountering unfamiliar functions or data types in code: the first step is to locate their definitions, understand their purpose, and then integrate this understanding back into the original code context to achieve a comprehensive interpretation of the target code. In other words, even for an experienced programmer, it is challenging to accurately grasp the precise semantics of a function when presented in isolation. For a computational model, this challenge is even greater. Previous work has not effectively addressed this issue, as it continues to rely on analyzing single functions in isolation, preventing these methods from truly understanding the semantic information of the function (Zhou et al., 2019; Chakraborty et al., 2021; Zhang et al., 2023b, 2024a). Therefore, incorporating the contextual information of the target code is crucial. The entire project repository can be utilized to assist the model in understanding the target code, as it contains the most comprehensive contextual information. However, due to the sheer size of code repositories, no model can support using the entire repository as input. Instead, it is necessary to identify and extract the context that is closely related to the target function from the repository. Previous studies (Siegmund, 2016; Liu et al., 2019; Theodoridis et al., 2022) have shown that callees and data types play a crucial role in understanding a snippet of source code. Therefore, we chose to include callee and data type information as the key context for the target function. By identifying and analyzing the key context within the function, we can achieve a deeper and more accurate comprehension of the semantic information of the function.

To do it specifically, we first leverage static analysis tool Tree-sitter[1] to extract all callees and data types from the target function. Callees and data types can be classified into two categories: those defined by standard libraries or third-party libraries, and those defined by the users. Since the LLM we employ for understanding functions has been pre-trained on a large corpus of source code, it already possesses knowledge of standard libraries and commonly used third-party libraries, allowing it to directly comprehend the semantics of such callees and data types. For uncommonly used third-party libraries, it is challenging to obtain relevant information due to their limited usage. We exclude such callees and data types from consideration. However, for user-defined callees and data types, the LLM lacks prior knowledge and cannot accurately interpret their semantics. Therefore, we need to extract all user-defined callees and data types for further analysis. We use gtags[2] and ctags[3] tools to generate tags for the entire project of the target function. We cross-reference all callees and data types with the data in the tags to extract the user-defined callees and data

[1] https://tree-sitter.github.io/tree-sitter/
[2] https://www.gnu.org/software/global
[3] https://github.com/universal-ctags/ctags

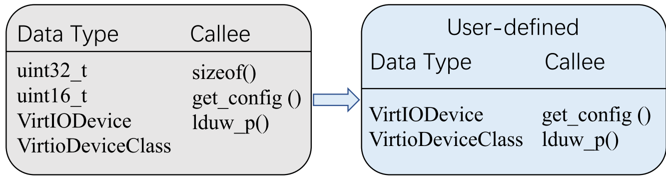


Fig. 5. Function information extraction example.

types. Take the function in Fig. 1 as an example. First, we extract all its callees, i.e., *sizeof*(), *get_config*(), *lduw_p*(), and data types, i.e., *uint32_t*, *uint16_t*, *VirtIODevice*, *VirtioDeviceClass*. Next, we identify the user-defined callees and data types through tags. The collected user-defined callees, i.e., *get_config*(), *lduw_p*(), and data types, i.e., *VirtIODevice*, *VirtioDeviceClass*, are presented in Fig. 5.

4.2. Project repository retrieval

In the previous step, we extract elements that influence the ability of the LLM to understand the function. The goal of this step is to help the LLM comprehend these elements, enabling it to better understand the semantic information of the target function. Similar to the process programmers follow to understand code snippets, we need to locate the specific definitions of the callees and data types within the project repository and comprehend these elements. During the repository retrieval step, we can identify another element containing critical semantic information: the caller of the target function. Including the caller can significantly enhance our understanding of the purpose of the target function (Liu et al., 2019; Theodoridis et al., 2022). Specifically, for each callee in the target function, we locate its definition using tags generated in previous section and collect its complete definition. For each data type, we apply the same approach to gather its full definition. Additionally, based on the tags, we retrieve all the callers of the target function.

We adopt DeepSeek-R1 (Guo et al., 2025) as the basic LLM in our framework. We consider multiple LLMs (including GPT-4, GPT-4o, OpenAI o1, Claude 3.5) and choose DeepSeek-R1 for the following reasons:

(1) Strong performance. DeepSeek-R1 performs on par with the state-of-the-art LLM OpenAI o1 on reasoning and coding-related tasks, while surpassing GPT-4, GPT-4o and Claude 3.5 (Guo et al., 2025; openaievals, 2025).
(2) Open-source and flexible deployment. Unlike proprietary models (e.g., GPT-4, GPT-4o, OpenAI o1, Claude 3.5), DeepSeek-R1 is fully open-source and can be used either via API or through local deployment. This flexibility facilitates large-scale and reproducible evaluations, avoids dependency on closed APIs, and allows us to maintain stable experimental conditions.
(3) Cost-effectiveness. DeepSeek-R1 provides competitive performance at a significantly lower cost, making it feasible for large-scale experiments on extensive code datasets. This balance of high performance and affordability makes it a practical choice for academic and applied research. According to the official API pricing, DeepSeek-R1 is priced at \$0.55 /M input tokens and \$2.19 /M output tokens (Deepseek, 2025), whereas OpenAI o1 costs \$15 /M input tokens and \$60 /M output tokens (Openai, 2025).

Although the selected DeepSeek-R1 model supports an input context length of up to 64K tokens, it is still impractical to feed all retrieved information into the LLM at once. Therefore, we need to refine this information further, minimizing its length while retaining key elements to effectively assist the LLM in understanding the code. In the project repository, the target function may have multiple callers. However, the primary contribution of the callers is to help us understand the purpose of the target function. Thus, it is unnecessary to include all callers. Instead, we select the most frequently used caller in the project and combine it with the target function to form a prompt. This prompt is then fed into Deepseek-R1 to extract “The Purpose of the Function”. The specific prompt template is shown in Fig. 6. In this way, “The Purpose of the Function” of the target function in Fig. 1 is illustrated in Fig. 7.

When the target function contains numerous callees and data types, we use Deepseek-R1 to filter them. A tailored prompt is constructed for the target function, enabling the LLM to select the 10 most critical callees and 5 most important data types that need to be understood. The specific prompt format is shown in Fig. 8. For each callee, our primary focus is its purpose as well as its input and output. Therefore, we generate a “Callee Function Signature” and “Callee Function Summary” for each callee. The function signature illustrates the parameters and return of a function. We leverage static analysis tool Tree-sitter to generate a function signature for each callee. In the project repository, not all functions have comments, and existing comments may not accurately describe the functionality of the functions. Therefore, we generate a function summary for each callee to describe its purpose and functionality.

We design a prompt template, as shown in Fig. 9, which is then populated with the callee source code and input into Deepseek-R1 to generate the output. For each data type, we do not perform further processing, but retain its definition to ensure that the LLM can accurately interpret the semantics of the corresponding variables in subsequent steps. The target function in Fig. 1 has two user-defined callees, whose “Callee Function Signature” and “Callee Function Summary” are presented in Fig. 10. We retain the complete definitions of the two user-defined data types (*VirtIODevice* and *VirtIODeviceClass*) in the target function.

4.3. Classification

In this step, we combine the target function with the information retrieved from the project repository, aiming to predict whether the input function is vulnerable or not. We also use DeepSeek-R1 as our classifier.

To enable the classifier to leverage the project repository knowledge, we adopted a Zero-shot Chain of Thought (CoT) prompting approach and designed a prompt. The CoT prompting technique guides the LLM to generate intermediate reasoning steps before producing the final answer, enabling more accurate and interpretable predictions. In this setting, the classifier receives the target function together with the retrieved contextual information from the project repository and is instructed to reason step by step before producing the final prediction. In our prompt, we explicitly state: “You need to understand the Target Function” and “Then I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities.”. These instructions encourage the LLM to reason step by step, simulating the thought process of a programmer who first understands the program and then decides whether it contains vulnerabilities. Specifically, the classification model is guided to (1) understand the target function using the retrieved context, (2)detect whether the target function contains vulnerabilities and output a classification label. The CoT prompting strategy complements our context retrieval module, which already provides essential project-level information. By explicitly reasoning over this contextual knowledge, the CoT process enhances both interpretability and classification accuracy without increasing prompt complexity.

Prior work found that using clearly articulated instructions and structuring the prompt into components such as instructions, the target function, and output indicators helps guide LLMs toward better performance on software vulnerability tasks (Fu et al., 2023; Yin et al., 2024). Accordingly, in our prompt design we followed their prompt structure.

The prompt used for vulnerability detection task is shown in Fig. 11. This prompt consists of four components: (1) Instruction, (2) Repository Knowledge, (3) Target Function, and (4) Output Indicator.

- **Instruction**. We provide LLM with the task instruction as “You need to understand the Target Function. This function is extracted from a project, the Repository Knowledge can help you understand the

You are an experienced software development engineer. Please summarize the purpose of the target function based on the target function itself and its caller function, in no more than 100 words.

Target Function: [ ];
Calleer Function: [ ]

Fig. 6. Target function summary prompt template.

The virtio_config_readw function reads a 16-bit value from the VirtIO device configuration space. It retrieves the configuration data for the device, checks if the requested address is within valid bounds, and returns the value at the specified address. If the address is out of bounds, it returns -1 (indicating an error). This function enables access to the device's configuration parameters for communication between the guest and the VirtIO device.

Fig. 7. Target function summary example.

You are an experienced software development engineer. You need to understand the following code. Please return no more than 10 most helpful user-defined callee functions in Callees below and no more than 5 user-defined data types in Datatypes below that would best help you understand this function. Please ensure that the callee and data type classifications are correct.

Return the information in the format {\"callee\": [,], \"datatype\":[,]}. Output in this format only, do not include any extra content.

CODE: [ ];
Callees: [ ];
Datatypes: [ ]

Fig. 8. Callee and data type filter prompt template.

You are an experienced software development engineer. Please summarize the following function in no more than 100 words.

Return the information in the format {\"signature\": [,], \"summary\":[,]}. Output in this format only, do not include any extra content.

CODE: [ ]

Fig. 9. Callee summary prompt template.

{"signature": ["static void get_config(VirtIODevice *vdev, uint8_t *config_data)"],
"summary": ["The function retrieves the configuration data for a VirtIO serial device, specifically setting the number of columns, rows, and maximum number of virtual serial ports. It converts the maximum number of virtual serial ports to a host-endian format before storing it in the provided configuration data structure."]},

{"signature": ["static inline uint16_t lduw_p(void *p)"],
"summary": ["The function `lduw_p` loads a 16-bit unsigned value from the memory address pointed to by `p`. It uses `memcpy` to copy 2 bytes of data into a local variable `val` and then returns the value. This is typically used for reading data from memory or buffers in little-endian format."]}

Fig. 10. Callee signature and summary example.

Instruction:
You need to understand the Target Function. This function is extracted from a project, the Repository Knowledge can help you understand the function. Then I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities.

Target Function: [ ]

Repository Knowledge:
Purpose of Target Function: [ ]
Callees: [{signature: , summary: }, …]
Data Types: [, …]

Context:
If there is a vulnerability, return 1; if there is no vulnerability, return 0. Please only output numbers.

Fig. 11. Classification prompt template.

function. Then I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities.”
- **Target Function**. We provide LLM with the source code of the target function.
- **Repository Knowledge**. We provide LLM with project repository knowledge that can steer the model to better responses.
- **Context**. We instruct LLM the format of the output as “If there is a vulnerability, return 1; if there is no vulnerability, return 0. Please only output numbers.”

5. Experiment setup

In this section, we first introduce the datasets used to evaluate RLV. Then, we present the information of our baselines. Finally, we introduce the evaluation metrics.

5.1. Datasets

In this work, we focus on detecting C/C++ vulnerabilities at the function level and conduct our study on two well-known C/C++ vulnerability datasets FFmpeg+QEMU (Zhou et al., 2019) and DiverseVul (Chen et al., 2023). These two datasets have been extensively utilized in prior research (Chakraborty et al., 2021; Li et al., 2021a; Wen et al., 2024b), and each data entry in these two datasets is accompanied by corresponding commit information, allowing us to locate the respective version of the repository. FFmpeg+QEMU dataset gathered security-related commits through keyword matching and manually labeled the before-commit functions as vulnerable. We use the FFmpeg+QEMU dataset released on CodeXGLUE. DiverseVul dataset gathered security-related commits and labeled the before-commit version of a changed function to be vulnerable, and the post-commit version to be non-vulnerable. Since our approach relies on project repositories for vulnerability detection and some projects in the DiverseVul dataset do not have accessible repositories, we selected only the entries belonging to downloadable project repositories. In total, this resulted in 309,387 data entries spanning 754 project repositories. The statistics of the two datasets are shown in Table 1.

5.2. Evaluation metrics

We adopt fourth widely-used classification metrics, i.e., accuracy, precision, recall and F_1-score to evaluate the performance of RLV.

- **Accuracy:** Accuracy is the proportion of functions correctly predicted out of the total number of functions. It is calculated as $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, where TP is the number of functions that are predicted by the model to be vulnerable and indeed contain vulnerabilities, TN is the number of functions that are correctly predicted as non-vulnerable, FP is the number of functions that are predicted to be vulnerable but do not contain vulnerabilities, FN is the number of functions that actually contain vulnerabilities but are predicted to be non-vulnerable.
- **Precision:** Precision refers to the ratio of correctly identified vulnerable functions to the total number of functions predicted by the model as vulnerable. It is calculated as $Precison = \frac{TP}{TP+FP}$.
- **Recall:** Recall represents the ratio of actual vulnerable functions that are correctly identified as vulnerable by the model. It is calculated as $Recall = \frac{TP}{TP+FN}$.
- **F_1-score:** F_1-score considers both precision and recall, serving as their harmonic mean to provide a balanced measure of model performance. It is calculated as $F_1\text{-score} = \frac{2*Precision*Recall}{Precision+Recall}$

Table 1
The statistics of datasets.

Dataset	Vul	Not-Vul	Ratio	Total
FFmpeg+QEMU	12,460	14,858	1:1.19	27,318
DiverseVul	17,976	291,411	1:16.21	309,387

5.3. Baselines

We compare RLV with three types of vulnerability detection techniques as follows:

Deep Learning-based Approaches. We choose two widely used baselines and a newest baseline.

- Devign (Zhou et al., 2019). Devign is a GNN-based model. It utlizes AST, CFG, DFG and code sequences for graph-based learning. The AST serves as its backbone, while the CFG and DFG are integrated into a unified graph. Then it employs a Gated Graph Neural Network to learn the node features for graph-level classification.
- REVEAL (Chakraborty et al., 2021). REVEAL is a GNN-based model designed for function-level vulnerability detection. It begins by extracting the Code Property Graph (CPG), which integrates the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) of a function. Then it encodes graph nodes into vectors and inputs them into Gated Graph Neural Network to identify vulnerabilities.
- MGVD (Qiu et al., 2024) MGVD is a novel vulnerability detection approach that uses multiple graph representations and a token se-

quence to encode structural and semantic features of functions into a three-channel feature matrix, with a weight allocation layer balancing these features, and applies a CNN to identify vulnerabilities.

LLM fine-tuning-based Approaches. We selected the three best-performing methods as demonstrated in previous studies (Guo et al., 2022; Wen et al., 2024a).

- UniXcoder (Guo et al., 2022). UniXcoder is a unified cross-modal pre-trained model for programming languages, designed to integrate both semantic and syntactic information from code comments and Abstract Syntax Trees (AST).
- CodeT5 (Wang et al., 2021). CodeT5 is a pre-trained language model designed for code-related tasks. By training on large-scale code datasets, CodeT5 effectively learns the semantics and structure of programming languages, achieving state-of-the-art results on various code understanding and generation benchmarks.
- LineVul (Fu and Tantithamthavorn, 2022). LineVul is a Transformer-based model that first employs CodeBERT to identify vulnerable functions and then leverages attention mechanisms to pinpoint vulnerable lines of code. In our work, we adapt its function-level vulnerability detection component as a baseline.

Prompt Engineering-based Approaches. Our approach is based on large LLMs, and we believe that future advancements will continue to favor LLMs with larger parameter sizes. Therefore, we selected two LLMs with over 100 billion parameters and the highest user adoption as baselines.

- QWen3-32B (Yang et al., 2025). Qwen3 is Alibaba Cloud's latest language model family, featuring both dense and Mixture-of-Experts variants. It uniquely supports thinking mode and non-thinking mode, multilingual understanding across 119 languages.
- CodeLlama-70B (Roziere et al., 2023). CodeLlama is a family of code-specialized LLMs derived from Llama2. It supports long-context prompts, offers infilling and instruction-following variants, and achieves best-in-class open-source performance on benchmarks.
- DeepSeek-R1-Distill-Llama-70B (Guo et al., 2025). DeepSeek-R1-Distill-Llama-70B is a 70B-parameter model distilled from Llama3.3-70B using reasoning signals generated by DeepSeek-R1.
- Llama3.1-70B (Dubey et al., 2024). LLaMA3.1-70B is Meta's instruction-tuned 70B-parameter multilingual transformer. It excels in reasoning, coding, multilingual dialogue, and tool use.
- DeepSeek-R1 (Guo et al., 2025). DeepSeek-R1 is a reasoning-focused open-weight large language model trained via chain-of-thought reinforcement learning, culminating in the first-generation reasoning series.
- Doubao (DLM, 2024). Doubao is an AI tool developed by ByteDance based on the Lingque model. It offers functionalities such as a chatbot, writing assistant, and English learning assistant. Doubao can answer various questions, engage in conversations, and help users acquire information.
- GPT-4o (Gpt, 2024). GPT-4o is a multilingual, multimodal generative pre-trained transformer developed by OpenAI. It can process and generate text, images and audio. Its application programming interface (API) is twice as fast and half the price of its predecessor, GPT-4 Turbo.

6. Evaluation

We evaluate the performance of RLV on the two vulnerability datasets. We attempt to answer the following research questions:

- RQ1: How effective is our approach compared with the state-of-the-art baselines?
- RQ2: How does the generalization capability of our approach compare to state-of-the-art baselines?
- RQ3: How effective are the techniques in our approach?

6.1. RQ1: effectiveness evaluation

6.1.1. Experimental setup

For the FFmpeg + QEMU dataset, we use the training, validation, and test set divisions provided by CodeXGLUE. For the DiverseVul dataset, we split the dataset into training, validation, and test sets with a ratio of 8:1:1 and perform a deduplication process between the training and test sets, following previous works (Zhou et al., 2019; Chakraborty et al., 2021; Chen et al., 2023). We use the training set to train the baseline models, use the validation set for selecting best-performance models, and evaluate the performance of RLV and other baselines in the test set. We train the baselines with hyperparameters set according to their original paper and select the models with the best F_1-score on the validation set for evaluation.

To further compare the performance of RLV with the state-of-the-art baseline, we add a statistical test. Specifically, we conduct four additional experiments comparing RLV with the best-performing baseline (in terms of F_1-score) on the DiverseVul dataset, using different random seeds to split the training, validation, and test sets.

6.1.2. Experimental results

In this research question, we compare the performance of RLV with the baselines across two datasets. Table 2 presents the accuracy, precision, recall, and F_1-score of RLV and the baselines.

Table 3 shows the statistical test results comparing RLV with the best-performing baseline on the DiverseVul dataset.

We can make the following observations from the tables:

(1) RLV achieves performance comparable to fine-tuning–based LLM methods and attains state-of-the-art results. RLV achieves the highest F1-socre and recall on two datasets. Please note that the LLM fine-tuning baselines require fine-tuning with training data, whereas RLV is zero-shot and does not rely on training data. Thus the comparisons between RLV and them are not entirely fair. Nevertheless, RLV still achieves better performance to the state-of-the-art approaches. RLV improves upon prompt engineering-based approaches by 24.49% in terms of F_1-score. The comparisons with prompt engineering-based approaches demonstrate that RLV substantially improves their performance, indicating the repository information extracted and integrated by RLV effectively aids large LLMs in performing vulnerability detection.

CodeT5 achieves the second highest F_1-scores on both FFmpeg + QEMU and DiverseVul datasets. LineVul achieves the highest precision on FFmpeg + QEMU dataset. LLM fine-tuning-based approaches leverage lightweight LLMs, compared to traditional deep learning-based approaches, have larger parameter capacities, enabling them to better capture the characteristics of vulnerabilities. We believe that LLMs offer significant potential for deep learning-based vulnerability detection.

In the statistical test, the Wilcoxon signed-rank test yields a p-value of approximately 0.0625. Although this difference is not statistically significant at the 0.05 level, RLV outperforms CodeT5 in four out of five runs. These results indicate that RLV achieves performance comparable to CodeT5 while reaching the best overall performance.

(2) Prompt engineering-based approaches exhibit the poorest performance. GPT-4o achieves high recall on two datasets. However, its precision is relatively low, indicating a high rate of false positives. When it fails to accurately understand the target function, prompt engineering-based approaches tend to classify all potential defects as vulnerabilities. This limitation leads to suboptimal performance in vulnerability detection. Although Doubao is one of the most advanced large language models available, the results indicate that it performs poorly on vulnerability detection tasks, achieving the lowest scores on both datasets.

(3) All approaches exhibit unsatisfactory performance on Diversevul dataset. All approaches achieve relatively low F_1-scores on the DiverseVul dataset. Compared with the FFmpeg + QEMU dataset, DiverseVul dataset is highly imbalanced, where the number of non-vulnerable

Table 2
The performance of RLV and the baselines.

		FFmpeg + QEMU				DiverseVul			
Type	Approach	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
	Devign	0.5772	0.5397	0.5418	0.5408	0.8955	0.2120	0.2937	0.2463
	REVEAL	0.6025	0.5697	0.5506	0.5600	0.9077	0.2530	0.3014	0.2751
Deep Learning	MGVD	**0.6640**	0.6509	0.5793	0.6130	0.9084	0.2591	0.3098	0.2822
	UniXcoder	0.6636	0.6503	0.5793	0.6127	**0.9268**	**0.3421**	0.2809	0.3085
	CodeT5	0.6501	0.6102	0.6598	0.6340	0.9217	0.3253	0.3226	0.3239
LLM Fine-tuning	LineVul	0.6633	**0.6516**	0.5737	0.6102	0.9209	0.3162	0.3109	0.3135
Prompt	Qwen3-32B	0.5630	0.5257	0.4964	0.5107	0.6309	0.0786	0.4989	0.1358
	CodeLlama-70B	0.5388	0.3529	0.0048	0.0094	0.7037	0.0900	0.4499	0.1500
	DeepSeek-R1-Distill-Llama-70B	0.5102	0.4749	0.6247	0.5396	0.5944	0.0800	0.5695	0.1403
	Llama-3.1-70B	0.4766	0.4642	**0.9044**	0.6135	0.2467	0.0658	**0.9060**	0.1227
	DeepSeek-R1	0.5253	0.4901	0.8295	0.6162	0.8623	0.1900	0.4199	0.2617
	Doubao	0.4535	0.4202	0.4996	0.4565	0.8569	0.1030	0.1897	0.1335
	GPT-4o	0.5007	0.4751	0.8303	0.6044	0.7769	0.1120	0.4099	0.1759
	RLV	0.5684	0.5186	0.8454	**0.6428**	0.8898	0.2527	0.4583	**0.3258**

Table 3
Statistical test of RLV and the best-performing baseline.

		DiverseVul			
Testset	Approach	Accuracy	Precision	Recall	F1-score
	CodeT5	0.9217	0.3253	0.3226	0.3239
Testset1	RLV	0.8898	0.2527	0.4583	**0.3258**
	CodeT5	0.9302	0.4012	0.4099	**0.4055**
Testset2	RLV	0.9126	0.3301	0.4900	0.3944
	CodeT5	0.9194	0.3165	0.3343	0.3251
Testset3	RLV	0.8911	0.2645	0.4911	**0.3438**
	CodeT5	0.9151	0.2915	0.3220	0.3060
Testset4	RLV	0.8923	0.2620	0.4700	**0.3365**
	CodeT5	0.9219	0.3356	0.3504	0.3429
Testset5	RLV	0.8989	0.2844	0.4878	**0.3593**
p-value					*0.0625*

functions greatly exceeds that of vulnerable ones, making the classification task particularly challenging. Moreover, many vulnerabilities in DiverseVul are non-local, involving interactions across multiple functions rather than being confined to a single one. This characteristic limits the effectiveness of all methods that operate at the single-function level, thereby contributing to the overall lower F_1-scores.

6.2. RQ2: generalization evaluation

6.2.1. Experimental setup

In real-world deployment scenarios, a vulnerability detection model may be applied to new developer projects that were not included in its training data. The model must still be capable of identifying vulnerabilities within these previously unseen projects. Since the FFmpeg + QEMU dataset consists of only two projects, i.e., FFmpeg and QEMU, training on one project and testing on the other typically does not align with real-world application scenarios. In contrast, the DiverseVul dataset includes 754 projects, making it more suitable for experimentation. Therefore, we choose to conduct our experiments solely on the DiverseVul dataset. We follow the approach outlined in DiverseVul (Chen et al., 2023) by randomly selecting 100 unique projects as the unseen projects test set to evaluate all models in this experiment. The remaining projects are designated as seen projects and are included in both the training and validation sets. We randomly sample 90% of the seen projects as the training set, and 10% remaining projects are the validation set. The statistics of the dataset are shown in Table 4.

Table 4
The statistics of DiverseVul.

	Projects	Vul	Not-Vul	Ratio	Total
Train	585	14,423	241,429	16.74	255,852
Valid	66	1945	26,865	13.81	28,810
Test	100	1608	23,117	13.38	24,725

6.2.2. Experimental results

In this research question, we compare the performance of RLV with the baselines on unseen projects. Table 5 presents the accuracy, precision, recall, and F_1-score of RLV and the baselines. We can make the following observations from the table:

(1) Compared to RQ1, the F_1-scores of most baselines show a significant decline. Deep learning-based approaches and LLM fine-tuning-based approaches show no substantial differences in performance, as their F_1-scores are approximately similar. We refer to these two types of approaches collectively as supervised learning-based approaches, as they both rely on learning vulnerability patterns from the training dataset. However, there can be substantial new vulnerability patterns in the projects outside the training dataset, which supervised learning-based approaches cannot accurately capture. As a result, their detection outcomes are often unsatisfactory. Prompting-based approaches perform similarly to their results in RQ1. Since they do not require training, encountering unseen projects does not affect their performance.

(2) RLV outperformed all baselines. RLV outperforms deep learning-based approaches and LLM fine-tuning-based approaches by 53.84%, 119.27% and 82.78% in terms of precision, recall and F_1-score respectively. Additionally, RLV outperforms prompting-based methods by 31.15%and 26.83% in terms of precision and F_1-score respectively. It indicates that RLV has a better superior generalization capability. RLV does not require additional training on new datasets. Instead, it enhances the model's ability to accurately understand the target function by incorporating project repository knowledge before performing vulnerability detection. As a result, our method maintains strong performance even when encountering unknown test projects. In real-world scenarios, the current dataset sizes remain inadequate for supervised learning-based methods to capture vulnerability patterns across different projects. This highlights the need for models with strong generalization capabilities. RLV is better suited to practical applications.

6.3. RQ3: ablation studies

6.3.1. Experimental setup

Our approach enhances existing approaches in several ways:

1) We add callee information in repository knowledge base.
2) We add data type information in repository knowledge base.
3) We add caller information in repository knowledge base.
4) We use the summary of callee as its information.
5) We use LLM to filter callees and data types.
6) We use a Zero-shot Chain of Thought (CoT) prompting approach.

Table 5
The performance of RLV and the baselines on unseen project DiverseVul.

		DiverseVul			
Type	Approach	Accuracy	Precision	Recall	F1-score
Deep Learning	Devign	0.8285	0.0900	0.1797	0.1199
	REVEAL	0.8574	0.1300	0.2096	0.1605
	MGVD	0.8716	0.1320	0.1748	0.1504
	UniXcoder	**0.8863**	0.1430	0.1499	0.1464
	CodeT5	0.8797	0.1560	0.1928	0.1725
LLM Fine-tuning	LineVul	0.8705	0.1381	0.1891	0.1596
Prompt	Qwen3-32B	0.5915	0.0900	0.5796	0.1558
	CodeLlama-70B	0.6665	0.0850	0.4229	0.1416
	DeepSeek-R1-Distill-Llama-70B	0.5821	0.0870	0.5715	0.1510
	Llama-3.1-70B	0.3069	0.0800	**0.9198**	0.1472
	DeepSeek-R1	0.8477	0.1830	0.3874	0.2486
	Doubao	0.8210	0.1220	0.2830	0.1705
	GPT-4o	0.8469	0.1710	0.3520	0.2302
	RLV	0.8702	**0.2400**	0.4596	**0.3153**

Table 6
The performance of RLV's variants.

	FFmpeg + QEMU				DiverseVul			
Approach	Accuracy	Precision	Recall	F_1-score	Accuracy	Precision	Recall	F_1-score
RLV-ce	0.5253	0.4901	0.8295	0.6162	0.8744	0.2161	0.4416	0.2902
RLV-dt	0.5329	0.4950	0.8295	0.6200	0.8737	0.2050	0.4077	0.2728
RLV-cr	0.5300	0.4931	0.8303	0.6188	0.8588	0.1850	0.4199	0.2568
RLV-ce_dt	0.5487	0.5052	**0.8494**	0.6336	0.8833	0.2301	0.4299	0.2997
RLV-ce_cr	0.5556	0.5100	0.8295	0.6317	0.8781	0.2200	0.4316	0.2915
RLV-dt_cr	0.5329	0.4950	0.8359	0.6218	0.8693	0.2010	0.4199	0.2719
RLV-ns	0.5575	0.5111	0.8406	0.6357	0.8890	0.2490	0.4516	0.3210
RLV-nf	0.5454	0.5031	0.8375	0.6286	**0.8913**	0.2521	0.4427	0.3212
RLV-zs	0.5560	0.5102	0.8398	0.6347	0.8873	0.2450	0.4516	0.3177
RLV	**0.5684**	**0.5186**	0.8454	**0.6428**	0.8898	**0.2527**	**0.4583**	**0.3258**

To evaluate the effectiveness of these techniques in RLV, we also perform ablation studies. Specifically, we construct several variants of RLV and compare the performance between RLV and these variants:

The effect of different information types in the repository knowledge base. To explore the impact of each type of information in the repository knowledge base, we compare RLV with its variants, as follows:

RLV-ce (callee information): RLV-ce only keeps the callee information in the repository knowledge base.

RLV-dt (data type information): RLV-dt only keeps the data type information in the repository knowledge base.

RLV-cr (caller information): RLV-cr only keeps the caller information in the repository knowledge base.

RLV-ce_dt (callee + data type information): RLV-ce_dt drops the caller information but keeps the callee and data type information in the repository knowledge base.

RLV-ce_cr (callee + caller information): RLV-ce_cr drops the data type information but keeps the callee and caller information in the repository knowledge base.

RLV-dt_cr (data type + caller type information): RLV-dt_cr drops the callee information but keeps the data type and caller information in the repository knowledge base.

The effect of callee summary. To investigate the impact of adding summary of callee information, we compare RLV with its variant that do not include summary information, as follows:

RLV-ns (not include summary): RLV-ns includes only the function signature in the callee information, without the summary.

The effect of LLM filter. To explore the impact of using LLM to filter 10 most helpful user-defined callee functions and 5 usef-defined data types, we compare RLV with its variants that do not utilize LLM for filtering, as follows:

RLV-nf (not include filtering): RLV-nf selects 10 callees and 5 data types randomly for target functions containing more than 10 callees or 5 data types.

The effect of Zero-shot Chain of Thought (CoT) prompting. We also perform an ablation study to investigate the effectiveness of Chain of Thought (CoT) prompting in RLV. We compare RLV with Zero-shot prompting variant as follows:

RLV-zs (zero-shot prompting): RLV-zs uses Zero-shot prompting instead of Zero-shot Cot prompting. We modified the Instruction section of the prompt to "I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities. This function is extracted from a project, the Repository Knowledge can help you understand the function."

6.3.2. Experimental results

The effect of different information types in the repository knowledge base. Table 6 demonstrates the performance of RLV-ce, RLV-dt, RLV-cr, RLV-ce_dt, RLV-ce_cr and RLV-dt_cr compared with RLV. From this table, we see that **RLV outperforms the six variants**. In detail, RLV outperforms the single-type information variants in terms of each metric on two datasets. RLV performs better than the two types information variants in terms of all evaluated metrics on the FFmpeg + QEMU dataset and achieves the highest F_1-score on the DiverseVul dataset. By comparing RLV with RLV-ce, RLV-dt, and RLV-cr, we observe that using data type or caller information alone results in suboptimal performance. Further comparisons with RLV-ce_dt, RLV-cd_cr, and RLV-ct_cr reveal the following: (1) Each type of information in the repository knowledge base contributes to improving detection performance, demonstrating the effectiveness of the repository knowledge base in RLV. (2) Among all types of information in the repository knowledge base, callee information provides the most significant enhancement in vulnerability detection performance. These results confirm the effectiveness and necessity of the three types of information incorporated in the repository knowledge base.

The effect of callee summary. Table 6 demonstrates the performance of RLV-ns and RLV. From this table, we find that **RLV achieves better performance than RLV-ns in all evaluated metrics on both**

Table 7
The performance of RLV's LLM variants.

Approach	FFmpeg + QEMU				DiverseVul			
	Accuracy	Precision	Recall	F_1-score	Accuracy	Precision	Recall	F_1-score
RLV-qw3-32b	0.4898	0.4530	0.5339	0.4901	0.8648	0.1172	0.2030	0.1486
RLV-clm-70b	0.5359	0.3673	0.0143	0.0276	0.7230	0.1100	0.4596	0.1775
RLV-dl-70b	0.5165	0.4823	0.7147	0.5759	0.7314	0.1300	0.5498	0.2103
RLV-lm3.1-70b	0.4967	0.4750	0.9100	0.6242	0.1834	0.0600	**0.8899**	0.1124
RLV-dq-8b	0.5428	0.5029	0.4151	0.4548	0.8555	0.1100	0.2097	0.1443
RLV-lm3.1-8b	0.4616	0.4601	**0.9920**	0.6286	0.0793	0.0589	0.9911	0.1112
RLV-doubao	0.4736	0.4355	0.4924	0.4622	0.8718	0.1223	0.1952	0.1504
RLV-gpt4t	0.5553	0.5097	0.8406	0.6346	0.8739	0.2182	0.4527	0.2944
RLV-gpt4o	0.5403	0.4998	0.8486	0.6291	0.8754	0.2206	0.4516	0.2964
RLV	**0.5684**	**0.5186**	0.8454	**0.6428**	**0.8898**	**0.2527**	0.4583	**0.3258**

datasets. This indicates that using callee summaries provides additional information, helping the LLM better understand the target function and significantly enhancing the effectiveness of RLV.

The effect of The effect of LLM filter. Table 6 shows the performance of RLV-nf and RLV. From this table, we find that **RLV achieves better performance than RLV-nf in most evaluated metrics on both datasets.** In the FFmpeg + QEMU dataset, there are 319 functions with callee count greater than 10 or data type count greater than 5. Among the 319 functions, RLV with LLM filtering detect 10 more vulnerable functions than RLV-nf with random filtering, and reduces 53 false positives. In the DiverseVul dataset, there are 2982 functions with callee count greater than 10 or data type count greater than 5. Among the 2982 functions, RLV with LLM filtering detect 28 more vulnerable functions than RLV-nf with random filtering, and reduces 75 false positives. This demonstrates that using LLM filtering can more effectively identify callees and data types that assist the LLM in understanding the target function, thereby improving the performance of RLV.

The effect of Chain of Thought (CoT) prompting. Table 6 shows the performance of RLV-zs compared with RLV. We observe that **RLV, which uses CoT, slightly outperforms RLV-zs in all evaluated metrics on all used datasets.** Compared to Zero-shot prompting, Zero-shot CoT prompting guides the model to generate a reasoning process before arriving at the final answer. This method significantly improves the model's capability to handle complex tasks by breaking them into logical steps, resulting in more accurate and interpretable outcomes. We guide the model to first understand the target function before performing vulnerability detection, enabling the model to decompose the task and process each logical layer step by step. This approach enhances the performance of vulnerability detection.

7. Discussion

7.1. The effects of different LLMs

Our model utilizes Deepseek-R1 as the classification model. Specifically, we conduct additional experiments by employing all prompt engineering-based LLM baselines from RQ1, along with the following models, as classifiers within the RLV framework. These include both closed-source and open-source models.

- GPT-4 Turbo (Achiam et al., 2023). GPT-4 Turbo is an optimized version of OpenAI's GPT-4 model, designed to deliver high-quality natural language understanding and generation while being faster and more cost-efficient. It retains the capabilities of GPT-4 but is fine-tuned for improved performance.
- DeepSeek-R1-Qwen3-8B. DeepSeek-R1-Qwen3-8B is an 8B-parameter model distilled from DeepSeek-R1-0528 onto the Qwen3-8B base using chain-of-thought supervision.
- Llama-3.1-8B. Llama-3.1-8B is an 8B-parameter instruction-tuned model in Meta's Llama-3.1 family, optimized for multilingual dialogue, reasoning, code understanding, and long-context support up to 128K tokens.

We denote the variants using Qwen3-32B, CodeLlama-70B, DeepSeek-R1-Distill-Llama-70B, Llama-3.1-70B, GPT-4 Turbo, GPT-4o, DeepSeek-R1-Qwen3-8B,Llama-3.1-8B as RLV-qw3-32b, RLV-clm-70b, RLV-dl-70b, RLV-lm3.1-70b, RLV-gpt4t, RLV-gpt4o, RLV-dq-8b, RLV-lm3.1-8b, respectively.

Table 7 demonstrates the performance of RLV variants. From this table, we see that: **(1)** Only a subset of RLV variants achieve promising performance. Specifically, RLV-gpt4t and RLV-gpt4o perform well. On the FFmpeg + QEMU dataset, their F_1-scores are 0.6346 and 0.6291, respectively. On the DiverseVul dataset, their F_1-scores are 0.2944 and 0.2964, respectively. However, most open-source models still performed poorly. **(2)** RLV variants based on smaller-parameter LLMs exhibit subpar performance. On the DiverseVul dataset, RLV-dq-8b, RLV-lm3.1-8b achieve F_1-scores of merely 0.1443 and 0.1112, respectively. **(3)** Compared to Table 2, most prompt engineering-based LLM baselines from RQ1 show improved performance when integrated into the RLV framework. Except on the FFmpeg + QEMU dataset, where RLV-qw3-32b shows no improvement over the Qwen3-32B baseline in terms of F_1-score, RLV-clm-70b, RLV-dl-70b, RLV-lm3.1-70b and RLV-gpt4o consistently outperform their corresponding baselines on both datasets. This demonstrates that the RLV framework effectively enhances the applicability of LLMs to vulnerability detection tasks. It also suggests that, as LLMs continue to improve, prompt engineering-based vulnerability detection tools hold substantial potential for further advancement.

The computational cost of LLMs is also an important factor to consider. In our experiments, all open-source models, including DeepSeek-R1 and other baseline models, are deployed on a GPU cluster equipped with NVIDIA H20 GPUs. For the three closed-source models (Doubao, GPT-4o, and GPT-4 Turbo), we access them via their public APIs. On average, completing the vulnerability detection task for a single function using the closed-source models costs approximately \$0.0143 for Doubao, \$0.0164 for GPT-4o, and \$0.0238 for GPT-4 Turbo, respectively.

7.2. The effects of prompt strategies

In RQ1, the prompt-engineering-based baselines use the same prompting strategy as RLV, the Zero-shot CoT setting. Since other prompting strategies, such as Zero-shot and Few-shot prompting, are also widely used in practice, we further evaluate these baselines under different prompting configurations. Following prior studies (Fu et al., 2023; Yin et al., 2024), we design specific templates for each strategy, where the Zero-shot template is shown in Fig. 12, and the Few-shot template is shown in Fig. 13. Table 8 presents the performance of the baseline variants. From this table, we observe that for each LLM, when performing the vulnerability detection task alone, different prompting strategies show no noticeable difference in performance. To-

Instruction:
I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities.

Target Function: [ ]

Context:
If there is a vulnerability, return 1; if there is no vulnerability, return 0. Please only output numbers.

Fig. 12. Zero-shot prompt template.

Instruction:
I want you to act as a vulnerability detection system, to detect whether the Target Function contains vulnerabilities.

Here are some examples:
Example 1:
[Code]
void process_input(char *data) {char buffer[10];strcpy(buffer, data);}
[Label]
1
Example 2:
[Code]
int add(int a, int b) {return a + b;}
[Label]
0

Now analyze the Target Function below.
Target Function: [ ]
Context:
If there is a vulnerability, return 1; if there is no vulnerability, return 0. Please only output numbers.

Fig. 13. Few-shot prompt template.

Table 8
The performance of baselines' variants.

		FFmpeg + QEMU				DiverseVul			
Approach	Prompt Strategies	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
	Zero-shot	0.5593	0.5214	0.4948	0.5078	0.6226	0.0770	0.5000	0.1335
	Few-shot	**0.5666**	0.5302	0.4972	0.5132	0.5986	0.0710	0.4889	0.1240
Qwen3-32B	Zero-shot CoT	0.5630	0.5257	0.4964	0.5107	0.6309	0.0786	0.4989	0.1358
	Zero-shot	0.5392	0.3750	0.0048	0.0094	0.7066	0.0910	0.4505	0.1514
	Few-shot	0.5392	0.3571	0.0040	0.0079	0.7112	0.0912	0.4427	0.1512
CodeLlama-70B	Zero-shot CoT	0.5388	0.3529	0.0048	0.0094	0.7037	0.0900	0.4499	0.1500
	Zero-shot	0.4912	0.4601	0.6199	0.5282	0.6052	0.0800	0.5517	0.1397
	Few-shot	0.5044	0.4702	0.6215	0.5353	0.6491	0.0910	0.5606	0.1566
DeepSeek-R1-Distill-Llama-70B	Zero-shot CoT	0.5102	0.4749	0.6247	0.5396	0.5944	0.0800	0.5695	0.1403
	Zero-shot	0.4590	0.4551	0.9004	0.6046	0.2276	0.0630	0.8860	0.1176
	Few-shot	0.4747	0.4631	0.8996	0.6114	0.2231	0.0632	0.8949	0.1181
Llama-3.1-70B	Zero-shot CoT	0.4766	0.4642	**0.9044**	0.6135	0.2467	0.0658	**0.9060**	0.1227
	Zero-shot	0.5223	0.4882	0.8207	0.6122	0.8618	0.1880	0.4155	0.2589
	Few-shot	0.5300	**0.4931**	0.8295	**0.6185**	0.8576	0.1810	0.4116	0.2514
DeepSeek-R1	Zero-shot CoT	0.5253	0.4901	0.8295	0.6162	0.8623	**0.1900**	0.4199	**0.2617**
	Zero-shot	0.4660	0.4300	0.4996	0.4622	**0.8637**	0.1100	0.1897	0.1392
	Few-shot	0.4561	0.4202	0.4845	0.4500	0.8602	0.1040	0.1846	0.1331
Doubao	Zero-shot CoT	0.4535	0.4202	0.4996	0.4565	0.8569	0.1030	0.1897	0.1335
	Zero-shot	0.4945	0.4712	0.8199	0.5984	0.7703	0.1090	0.4116	0.1724
	Few-shot	0.4967	0.4721	0.8096	0.5964	0.7863	0.1130	0.3910	0.1753
GPT-4o	Zero-shot CoT	0.5007	0.4751	0.8303	0.6044	0.7769	0.1120	0.4099	0.1759

gether with the results from RQ1, RQ3, and Section 7.1, we find that performance improves substantially only when the prompt incorporates additional contextual information relevant to vulnerability detection, as achieved in RLV.

7.3. Validity of code summaries generated by LLMs

In our approach, we employ the LLM DeepSeek-R1 to generate function summaries that provide contextual information for downstream vulnerability detection. However, LLM-generated summaries may contain semantic inaccuracies due to the inherent limitations of current models in understanding complex code semantics. To assess their reliability, we conduct a user study on 50 randomly selected functions from the FFmpeg project in the FFmpeg + QEMU dataset. Three experienced programmers familiar with the FFmpeg project independently examine whether each summary accurately describes the corresponding function's purpose and functionality.

All three annotators agree that 45 out of 50 summaries (90%) are accurate. For the remaining five cases, the main issue is the incorrect description of the behavior of invoked functions, while the input and output descriptions remain correct.Moreover, the ablation study results in RQ3 confirm that incorporating these summaries into our framework improves detection performance, even though the summaries are not perfectly accurate. This suggests that the LLM-generated summaries, while imperfect, effectively capture high-level semantic cues that enhance the understanding of target functions, thereby improving the overall effectiveness of our vulnerability detection approach.

8. Threats to validity

Threats to internal validity refer to errors in our experiments. For each baseline, we carefully reuse its existing implementation. The selected baselines provide publicly available replication packages online, which we obtained from GitHub. For each baseline, we followed the instructions outlined in its README file to replicate it, using the settings and parameters provided in the replication package without any modifications. Since none of the existing baseline methods provide their dataset splits, it is not possible to fully replicate their original results. We follow the dataset provider's original division method as described in their documentation to ensure fairness to the greatest extent possible. Threats to external validity relate to the generalizability of our approach. This study focuses exclusively on C/C++ vulnerabilities. While our approach has proven effective on two C/C++ vulnerability datasets, we cannot assert that it will deliver comparable performance on datasets from other programming languages. Threats to construct validity pertain to the evaluation metrics employed in our study. To mitigate this threat, we utilize four commonly used classification metrics: accuracy, precision, recall, and F_1-score, which have been widely adopted in previous vulnerability detection research.

Besides, our approach still has the following limitations: (1) At present, our method is constrained by the maximum input context length supported by LLMs. Although the input capacity of large language models can reach up to 256k tokens, and some models are limited to 64k. This results in only a subset of callees and data types being selected as repository knowledge when handling certain long functions, which may affect the model's understanding of the code. To address this limitation, we plan to explore a segmented, multi-round input approach in future work. (2) Our approach may face challenges in detecting vulnerabilities involving uncommonly used third-party libraries, as we rely on LLMs to understand the function calls and data types from third-party libraries and LLMs may be ineffective in understanding the knowledge in uncommonly used third-party libraries. In the future, we plan to introduce an effective online retrieval strategy to enable the LLM to access information from used third-party libraries. (3) Our approach is currently restricted to function-level vulnerability detection. We may explore linking multiple functions through function call relationships to build a vulnerable function path dataset. This can provide guidance for vulnerability detection involving multiple functions.

9. Related work

Our work adopts several advanced techniques from the fields of deep learning-based vulnerability detection and LLM-based vulnerability detection. In this section, we discuss the related work in these fields.

9.1. Deep-learning-based vulnerability detection

With the progression of deep learning, an increasing number of studies have begun to employ deep learning techniques for vulnerability detection. Li et al. proposed a methodology that uses code gadgets generated through data dependence and leverages Bi-directional Long Short-Term Memory (BiLSTM) to detect vulnerabilities (Li et al., 2018). However, this approach is limited to detecting vulnerabilities related to library or API function calls. Zhou et al. proposed a model called Devign (Zhou et al., 2019), which applies a general graph neural network to detect vulnerability. Devign contains a novel convolutional component that can effectively extract useful features from the learned rich node representations for graph-level classification tasks. Chakraborty et al. proposed a GNN-based model named REVEAL (Chakraborty et al., 2021) for function-level vulnerability detection. REVEAL translates real-world code into a graph, then leverages representation learning techniques for model building, which makes the model automatically learn to represent features of vulnerable and benign code in the feature space. Li et al. proposed IVDETECT (Li et al., 2021a), which leverages a PDG to represent a function then inputs the PDG into a GNN for classification to determine whether the function is vulnerable. Cheng et al. proposed DeepWukong (Cheng et al., 2021), which extracts program slices based on PDG and feeds these slices into a GCN to detect vulnerability. Qiu et al. proposed a multiple-graph-based model named MGVD (Qiu et al., 2024) for function-level vulnerability detection. It uses three different ways to represent each function into multiple form and use CNN for vulnerability detection.

9.2. LLM-based vulnerability detection

9.2.1. Fine-tuning-based vulnerability detection

Peng et al. (2023) employed program slicing to capture control and data dependency information, which was used to assist LLMs in detecting vulnerabilities. Zhang et al. (2023b) introduced a method that breaks down syntax-based Control Flow Graphs (CFGs) into individual execution paths and inputs these paths into LLMs for vulnerability detection. Fu and Tantithamthavorn (2022) proposed a transformer-based model named LineVul, which utilizes CodeBERT to perform line-level vulnerability detection. Wang et al. (2021) presented CodeT5, a unified pre-trained transformer model that leverages the code semantics conveyed from the developer-assigned identifiers. CodeT5 has also been applied to vulnerability detection, demonstrating strong performance. Guo et al. (2022) presented UniXcoder, a unified cross-modal pre-trained model, which utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages ASTs and code comments to enhance code representation. UniXcoder achieved the best performance on the CodeXGLUE benchmark.

9.2.2. LLM + DL vulnerability detection

Bagheri and Hegedűs (2021) leveraged BERT for source code representation and incorporated an LSTM to detect vulnerabilities in Python programs. They only needed to train the LSTM classifier, and the experimental results showed that using BERT for representation yielded better performance than using word2vec or fastText. Yuan et al. (2022) incorporated a CodeBERT-based embedding approach for vulnerable function detection and demonstrated performance gains when using a frozen pre-trained encoder with a downstream classifier. Kalouptsoglou

et al. (2025) explored transfer learning with Transformer-based models (CodeBERT and CodeGPT) through full fine-tuning, partial freezing with a classification head, and embedding extraction with alternative classifiers, showing that frozen and embedding-based strategies can achieve competitive yet more efficient results.

9.2.3. Prompt engineering-based vulnerability detection

Zhang et al. (2024a) explored different prompts to enhance ChatGPT's performance in vulnerability detection. Yin et al. (2024) leveraged between 1 and 6 few-shot learning examples to utilize the fixed context window effectively. Ren et al. (2024) proposed ProRLearn, which is a recent vulnerability detection framework that integrates prompt tuning and reinforcement learning to enhance code representation learning and detection performance. It guides the model to generate vulnerability-relevant features using tailored prompt templates and leverages reinforcement learning to iteratively refine predictions based on feedback from the environment. Unlike our approach, ProRLearn still relies on supervised training.

Wen et al. (2024b) proposed Vuleval, the first attempt to build a repository-based vulnerability detection dataset. Vuleval employs Jaccard Similarity or Edit Similarity retrieval methods to predict vulnerability-related dependencies (i.e., a callee and a caller) within the all callees and callers of the target function. Vuleval then combines the target function with vulnerability-related dependencies and input them into the model to obtain detection result. However, Vuleval faces two primary problems: (1) The retrieval methods it employs often fail to accurately identify vulnerability-related dependencies; (2) Even with the inclusion of vulnerability-related dependencies, Vuleval has not considered the contextual information that could assist LLMs in understanding the semantic context of the function, resulting in LLMs struggling to correctly understand the target function. Consequently, Vuleval's experimental results demonstrate that GPT-3.5-instruct's performance declines after incorporating vulnerability-related dependencies, while ChatGPT only shows a marginal F_1-score improvement of 0.0063 after the inclusion. However, our approach adopts a different strategy from Vuleval. We aim to assist the LLM in understanding the target function before conducting vulnerability detection. We follow the process by which programmers understand code, directing the LLM to retrieve and apply various LLM-based refinement techniques (e.g., filtering and generating function summaries) from the project repository, which it deems necessary to aid in its comprehension of the target function. Consequently, our experiments demonstrate a greatly improvement over approaches that use the LLM to detect vulnerability directly.

Du et al. (2024) proposed Vul-RAG, which builds a vulnerability knowledge base from existing CVE instances. When detect vulnerability, Vul-RAG retrieves the relevant vulnerability knowledge from the constructed knowledge base based on functional semantics. Vul-RAG's key innovation is that Vul-RAG first retrieves relevant knowledge from vulnerability knowledge base based on functional semantics and then detects vulnerability by reasoning from the vulnerability causes and fixing solutions. Our approach differs from Vul-RAG in its key idea. Vul-RAG builds a knowledge base using publicly available CVE instances, which are external information, and performs enhanced retrieval within that knowledge base. In contrast, our approach treats the target function's project repository as the knowledge base and conducts enhanced retrieval within the project repository, which means our approach does not require an external vulnerability information database.

10. Conclusions and future work

We propose a novel approach named RLV which leverages project repository knowledge to detect function-level vulnerabilities. RLV emulates the process programmers use to understand the target function, leveraging the LLM to retrieve the information that the LLM finds most helpful for understanding the target function from the project repository. RLV uses the LLM to further refine and enhance the retrieved information (e.g., filtering and generating function summaries) and then uses the refined information as the knowledge base of the target function. Finally, RLV combines the target function with the knowledge base to form a prompt, which is then input into the LLM to obtain detection results. In this way, the LLM can accurately understand the target function. Our experimental results show that RLV outperforms state-of-the-art baselines in the cross-project setting and achieves competitive performance in the within-project setting. Moreover, the results demonstrate that the proposed context extraction and refinement mechanism effectively enhances prompt-based vulnerability detection methods by enabling LLMs to better utilize project-level contextual information.

In future work, we plan to improve the effectiveness of our approach by using more effective RAG techniques. We also plan to extend our approach to detect vulnerabilities in other programming languages (e.g., C# and Python). We will also explore the detection of vulnerabilities that span across multiple functions.

CRediT authorship contribution statement

Fangcheng Qiu: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Data curation, Conceptualization; **Zhongxin Liu:** Writing – review & editing, Supervision, Funding acquisition; **Bingde Hu:** Writing – review & editing; **Zhengong Cai:** Writing – review & editing; **Lingfeng Bao:** Writing – review & editing; **Xinyu Wang:** Writing – review & editing, Supervision.

Data availability

I have shared the link to my data in the paper.

Declaration of competing interest

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

Acknowledgment

This research is supported by Zhejiang Provincial Natural Science Foundation of China No. LZ25F020003.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al., 2023. GPT-4 technical report. arXiv:2303.08774

Bagheri, A., Hegedűs, P., 2021. A comparison of different source code representation methods for vulnerability prediction in python. In: International Conference on the Quality of Information and Communications Technology. Springer, pp. 267–281.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877–1901.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: are we there yet. IEEE Trans. Software Eng. 48, 3280–3296.

Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D., 2023. DiverseVul: a new vulnerable source code dataset for deep learning based vulnerability detection. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 654–668.

Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. DeepWukong: statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. 30 (3), 1–33.

Croft, R., Babar, M.A., Kholoosi, M.M., 2023. Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, pp. 121–133.

Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L., 2023. Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 423–435.

DeepSeek API pricing, 2025. https://api-docs.deepseek.com/quick_start/pricing.

Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y., 2024. Vulnerability detection with code language models: how far are we? arXiv:2403.18624

Doubao large model, 2024. https://www.doubao.com/.

Du, X., Zheng, G., Wang, K., Feng, J., Deng, W., Liu, M., Chen, B., Peng, X., Ma, T., Lou, Y., 2024. Vul-RAG: enhancing LLM-based vulnerability detection via knowledge-level RAG. arXiv:2406.11147

Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., Wu, Y., 2019. VulSniper: focus your attention to shoot fine-grained vulnerabilities. In: IJCAI, pp. 4665–4671.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al., 2024. The Llama 3 herd of models. arXiv:2407

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

Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., Tsang, R., Nazari, N., Wang, H., Homayoun, H., et al., 2024. Large language models for code analysis: do {LLMs} really do their job? In: 33rd USENIX Security Symposium (USENIX Security 24), pp. 829–846.

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

Fu, M., Tantithamthavorn, C.K., Nguyen, V., Le, T., 2023. ChatGPT for vulnerability detection, classification, and repair: how far are we? In: 2023 30th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 632–636.

2024. GPT-4o large model. https://openai.com/index/hello-gpt-4o/.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. UniXcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7212–7225.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al., 2025. DeepSeek-R1: incentivizing reasoning capability in LLMs via reinforcement learning. arXiv:2501.12948

openai simple-evals, https://github.com/openai/simple-evals.

https://github.com/anonymous-a-b-c/RLV/.

Jang-Jaccard, J., Nepal, S., 2014. A survey of emerging threats in cybersecurity. J. Comput. Syst. Sci. 80 (5), 973–993.

Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., Chatzigeorgiou, A., 2025. Transfer learning for software vulnerability prediction using transformer models. J. Syst. Softw. 227, 112448.

Kenton, J. D. M.-W.C., Toutanova, L.K., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of naacL-HLT. Vol. 1. Minneapolis, Minnesota, p. 2.

Ko, A.J., Uttl, B., 2003. Individual differences in program comprehension strategies in unfamiliar programming systems. In: 11th IEEE International Workshop on Program Comprehension, 2003.. IEEE, pp. 175–184.

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

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. 19, 2244–2258.

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

Liu, B., Wang, T., Zhang, X., Fan, Q., Yin, G., Deng, J., 2019. A neural-network based code summarization approach by using source code and its call dependencies. In: Proceedings of the 11th Asia-Pacific Symposium on Internetware, pp. 1–10.

Liu, Y., Tao, S., Meng, W., Wang, J., Ma, W., Chen, Y., Zhao, Y., Yang, H., Jiang, Y., 2024a. Interpretable online log analysis using large language models with prompt strategies. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, pp. 35–46.

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Tian, Z., Huang, Y., Hu, J., Wang, Q., 2024b. Testing the limits: unusual text inputs generation for mobile app crash detection with large language model. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–12.

Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A., 2007. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 529–540.

OpenAI API pricing, 2025. https://platform.openai.com/docs/pricing.

Peng, T., Chen, S., Zhu, F., Tang, J., Liu, J., Hu, X., 2023. PTLVD: program slicing and transformer-based line-level vulnerability detection system. In: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 162–173.

Qiu, F., Liu, Z., Hu, X., Xia, X., Chen, G., Wang, X., 2024. Vulnerability detection via multiple-graph-based code representation. IEEE Trans. Softw. Eng. 50, 2178–2199.

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

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al., 2023. Code Llama: open foundation models for code. arXiv:2308.12950

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

Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W., 2014. Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. 40 (10), 993–1006.

Siegmund, J., 2016. Program comprehension: past, present, and future. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 5. IEEE, pp. 13–20.

Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S., 2012. How do software engineers understand code changes? an exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1–11.

Theodoridis, T., Grosser, T., Su, Z., 2022. Understanding and exploiting optimal function inlining. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 977–989.

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

Wen, X.-C., Gao, C., Gao, S., Xiao, Y., Lyu, M.R., 2024a. Scale: constructing structured natural language comment trees for software vulnerability detection. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 235–247.

Wen, X.-C., Wang, X., Chen, Y., Hu, R., Lo, D., Gao, C., 2024b. VuleVal: towards repository-level evaluation of software vulnerability detection. arXiv:2404.15596

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al., 2025. Qwen3 technical report. arXiv:2505.09388

Yin, X., Ni, C., Wang, S., 2024. Multitask-based evaluation of open-source LLM on software vulnerability. IEEE Trans. Softw. Eng. 50, 3071–3087.

Yosinski, J., Clune, J., Bengio, Y., Lipson, H., 2014. How transferable are features in deep neural networks? Adv. Neural Inf. Process. Syst. 27, 3320–3328.

Yuan, X., Lin, G., Tai, Y., Zhang, J., 2022. Deep neural embedding for software vulnerability discovery: comparison and optimization. Secur. Commun. Netw. 2022 (1), 5203217.

Zhang, B., Haddow, B., Birch, A., 2023a. Prompting large language model for machine translation: a case study. In: International Conference on Machine Learning. PMLR, pp. 41092–41110.

Zhang, C., Liu, H., Zeng, J., Yang, K., Li, Y., Li, H., 2024a. Prompt-enhanced software vulnerability detection using chatgpt. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, pp. 276–277.

Zhang, J., Liu, Z., Hu, X., Xia, X., Li, S., 2023b. Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans. Softw. Eng. 49 (8), 4196–4212.

Zhang, T., Ladhak, F., Durmus, E., Liang, P., McKeown, K., Hashimoto, T.B., 2024b. Benchmarking large language models for news summarization. Trans. Assoc. Comput. Ling. 12, 39–57.

Zhou, X., Zhang, T., Lo, D., 2024. Large language model for vulnerability detection: emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, pp. 47–51.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32, 10197–10207.