VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection

Xin-Cheng Wen
Harbin Institute of Technology
Shenzhen, China
xiamenwxc@foxmail.com

Xinchen Wang
Harbin Institute of Technology,
Shenzhen, China
200111115@stu.hit.edu.cn

Yujia Chen
Harbin Institute of Technology,
Shenzhen, China
yujiachen@stu.hit.edu.cn

Ruida Hu
Harbin Institute of Technology,
Shenzhen, China
200111107@stu.hit.edu.cn

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

ABSTRACT

Deep Learning (DL)-based methods have proven to be effective for software vulnerability detection, with a potential for substantial productivity enhancements for detecting vulnerabilities. Current methods mainly focus on detecting single functions (i.e., intra-procedural vulnerabilities), ignoring the more complex inter-procedural vulnerability detection scenarios in practice. For example, developers routinely engage with program analysis to detect vulnerabilities that span multiple functions within repositories. In addition, the widely-used benchmark datasets generally contain only intra-procedural vulnerabilities, leaving the assessment of inter-procedural vulnerability detection capabilities unexplored.

To mitigate the issues, we propose a repository-level evaluation system, named **VulEval**, aiming at evaluating the detection performance of inter- and intra-procedural vulnerabilities simultaneously. Specifically, VulEval consists of three interconnected evaluation tasks: **(1) Function-Level Vulnerability Detection**, aiming at detecting intra-procedural vulnerability given a code snippet; **(2) Vulnerability-Related Dependency Prediction**, aiming at retrieving the most relevant dependencies from call graphs for providing developers with explanations about the vulnerabilities; and **(3) Repository-Level Vulnerability Detection**, aiming at detecting inter-procedural vulnerabilities by combining with the dependencies identified in the second task. VulEval also consists of a large-scale dataset, with a total of 4,196 CVE entries, 232,239 functions, and corresponding 4,699 repository-level source code in C/C++ programming languages. By evaluating 19 vulnerability detection methods on the data split randomly and by time respectively, we observe that the repository-level vulnerability detection framework outperforms the corresponding function-level methods, with an increase of 1.51% in F1 score and 2.63% in MCC on average. It indicates that incorporating vulnerability-related dependencies facilitates vulnerability detection. Our experimental results also demonstrate that the performance of program-analysis- and prompt-based methods are not affected when splitting the data by time. In addition, for the seven dependency retrieval methods studied, we find that lexical-based methods yield superior results than semantic-based methods for identifying vulnerability-related dependencies. Our analysis highlights the current progress and future directions for software vulnerability detection.

*Corresponding author.

1 INTRODUCTION

Software vulnerabilities, mostly caused by insecure code, can be exploited to attack software systems, and further cause the security issues such as system crash, data leakage, and even critical infrastructure damage. In the past ten years, the number of software vulnerabilities has increased more than five times, rising from 5,697 in 2013 to 29,065 in 2023 [48]. This increasing growth in both the quantity and type of software vulnerabilities has led to increasing economic losses [50]. For example, Clop ransomware has successfully extorted more than $500 million from various organizations [37]. Therefore, it is necessary to develop effective technologies for software vulnerability detection.

The existing vulnerability detection methods can be categorized into four types: program analysis-based, supervised learning-based, fine-tuning-based, and prompt-based methods. The traditional program analysis-based vulnerability detection techniques, such as INFER [15] and CheckMarx [25], rely on predefined rules to identify vulnerabilities. These approaches are labor-intensive and inefficient due to the diverse types of vulnerabilities and libraries. Deep learning (DL)-based approaches have emerged as effective solutions, exhibiting notable success by mitigating the reliance on domain expertise and enhancing the ability to detect a variety of software vulnerabilities [9]. Early DL-based approaches use the supervised learning-based methods, which leveraged Convolutional Neural Networks (CNNs) [60, 61], Recurrent Neural Networks (RNNs) [34, 47], and Graph Neural Networks (GNNs) [6, 65] for learning the vulnerability representation.

Nevertheless, the effectiveness of these supervised learning-based approaches is limited by the scarcity of vulnerability data [41]. The emergence of pre-trained models like CodeBERT [17] and UniXcoder [21], which are trained on large-scale open-source code repositories, has notably propelled this domain forward. These methods, equipped with extensive general programming knowledge, can be fine-tuned with vulnerability datasets to greatly enhance the vulnerability detection performance, denoted as fine-tuning-based methods. Nowadays, the prompt-based techniques have utilized Large Language Models (LLMs), such as LLaMA [51] and CodeLlama [46], for vulnerability detection, marking a trend of inclination towards unsupervised methodologies in the domain.

Despite substantial advancements in vulnerability detection through using fine-tuning and prompt techniques, evaluating the efficacy of these methods remains challenging. Specifically, there

```
1 void dd_close(struct dump_dir *dd)
2 {
3     if(!dd)
4         return;
5     dd_unlock(dd);
6     if(dd->next_dir)
7     {
8         closedir(dd->next_dir);
9     }
10    free(dd->dd_type);
11    free(dd->dd_dirname);
12    free(dd);
13}
```

(a) Source Code

```
1 int dd_delete(struct dump_dir *dd)
2 {
3     if(!dd->locked)
4     {
5         error msg("...",
                  dd->dd_dirname);
6     return -1;
7     int r= delete_file_dir
            (dd->dd_dirname, true);
8     dd->locked =0;
9     dd_close(dd);
10    return r;
11 }
```

(b) Callee Function

```
1 static void dd_unlock(struct dump_dir *dd)
2 {
3     if(dd->locked)
4     {
5         dd->locked = 0;
6         unsigned dirname_len =strlen(dd->dd_irname);
7         char lock_buf[dirname_len + sizeof("/.lock")];
8         strcpy(lock_buf, dd->dd_dirname);
9         strcpy(lock_buf + dirname_len, "/.lock");
10        xunlink(lock buf);
11        log_info("Unlocked '%s'",lock_buf);
12    }
13}
```

(c) Caller Function

Figure 1: An inter-procedural vulnerability example of the CWE-20. Lines highlighted in green denote the call relation (i.e., callee and caller), and red denotes the vulnerable statements.

exists a gap between the current evaluation scenarios and real-world vulnerability detection scenarios, embodied in the following two aspects:

(1) Lack of methods for detecting inter-procedural vulnerabilities. Despite the demonstrated efficacy of various methods for vulnerability detection, current evaluation frameworks primarily focus on the granularity of individual function or file, failing to fully account for the complexities of vulnerabilities that extend across multiple files or entire repositories. This narrow focus inadequately mirrors the complexity inherent in real-world vulnerability detection contexts, wherein developers routinely check with program analysis techniques to detect vulnerabilities that span multiple files in the repository level. For example, Figure 1 presents an inter-procedural vulnerability of CWE-20 (Improper Input Validation) [2]. Figure 1 (a), (b), and (c) illustrate the code snippet at the function level, the associated callee and caller functions, respectively. Specifically, the function `dd_close` assumes that the `dd` pointer is non-null without verification, and proceeds to invoke `dd_unlock` in Line 5 of Figure 1(a) and access member variables. It can cause the vulnerability (Lines 6-7 in Figure 1(c)). Similarly, `dd_delete` performs an operation contingent upon the locked status without ensuring that the `dd` pointer is valid (Lines 7-8 in Figure 1(b)). Such inter-procedural vulnerabilities across multiple functions are hard to be identified by existing methods.

(2) Lack of a comprehensive evaluation system for vulnerability detection. The existing work generally conducts the evaluation on randomly split function-/file-level datasets, without considering different scenarios separately and the timeliness. The previous datasets [16, 35] only use the vulnerability patches to construct the dataset, which ignores the corresponding dependencies (e.g., callee and caller) in the repository. In addition, due to the large number of dependencies from the call graph, it is necessary to retrieve vulnerability-related dependencies for developers. Furthermore, given the substantial vulnerabilities identified every year, the utilization of historical vulnerability data for detecting future vulnerabilities emerges as a critical need. However, the existing random-split setting may lead to risks of data leakage and the potential for inflated performance, which ultimately compromises the reliability of vulnerability detection methods and reflects the challenges present in real-world software development environments.

To mitigate the issues, in this paper, we propose a holistic evaluation system, named **VulEval**, designed for evaluating inter- and intra-procedural vulnerabilities simultaneously. Specifically, we perform three interconnected tasks to construct the evaluation system: **(1) Function-level Vulnerability Detection**, where the task is to predict the given code snippet whether it is vulnerable or not, aims at detecting intra-procedural vulnerability; **(2) Vulnerability-Related Dependency Prediction**, where the task is retrieving the vulnerability-related dependency from the call graph, thereby providing developers with explanations about the vulnerabilities; and **(3) Repository-level Vulnerability Detection**, aiming at detecting inter-procedural vulnerabilities. To explore the current vulnerability detection methods' performance in the third task, we propose a repository-level vulnerability detection framework by combining dependencies identified in the second task.

We collect a large-scale repository-level source code for each vulnerability patch to provide repository information. It consists of 4,196 CVE entries, 232,239 functions, and corresponding 4,699 repository-level source code in C/C++ programming languages. We also extract 347,533 function dependencies (i.e., Callee and Caller) and 9,538 vulnerability-related dependencies from the repository to detect the inter-procedural vulnerability.

Based on the proposed evaluation system, we empirically study the performance of the four types of vulnerability detection methods (i.e., 19 baselines) on VulEval for function- and repository-level vulnerability detection. We also evaluate the three types of retrieval methods (i.e., seven baselines) for vulnerability-related dependency prediction. During the evaluation, we analyze the effectiveness in two settings (i.e., random split and split by time). Furthermore, we highlight the current progress and shed light on future directions.

Key Findings. Based on the extensive experiments, our study reveals several key findings:

(1) Incorporating contexts related to vulnerabilities in repository-level vulnerability detection enhances the performance compared with function-level vulnerability detection.
(2) Supervised learning- and fine-tuning-based methods exhibit performance degradation within the time-split setting; while the performance of program-analysis- and prompt-based methods are not affected.

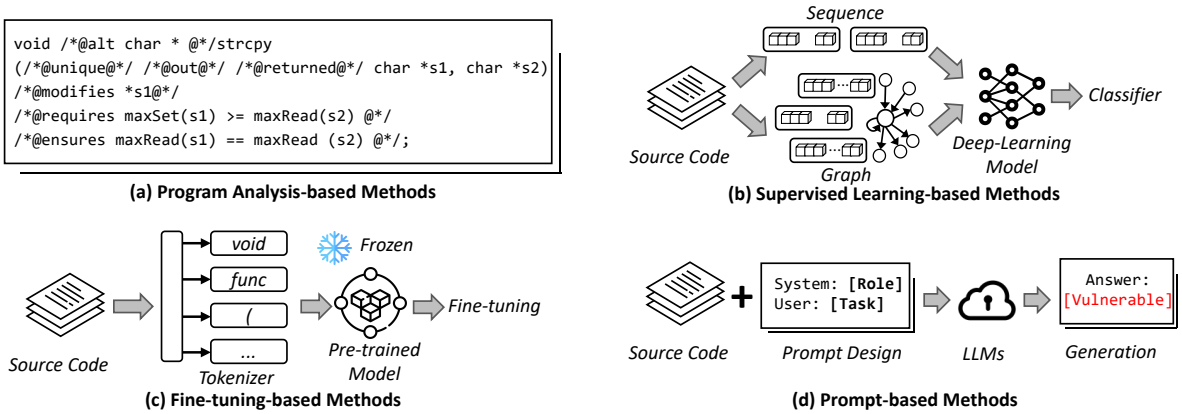


Figure 2: The four types of vulnerability detection methods.

(3) Lexical-based methods yield superior results than semantic-based methods in identifying vulnerability-related dependency. It is essential to develop more effective retrieval techniques for retrieving vulnerability-related dependencies.

Contributions. In summary, the major contributions of this paper are summarized as follows:

(1) To the best of our knowledge, we are the first to propose a holistic evaluation system for evaluating inter and intra-procedural vulnerabilities simultaneously.
(2) We collect a large-scale repository-level source code and extract corresponding dependencies that provide repository-level information. We extract 347,533 dependencies and 9,538 vulnerability-related dependencies to detect the inter-procedural vulnerability.
(3) We perform an extensive evaluation of 19 vulnerability detection methods and seven dependency retrieval methods in two settings. Our analysis highlights the current progress and future directions for software vulnerability detection.

2 BACKGROUND

In this section, we introduce the existing vulnerability detection methods, including program analysis-based, supervised learning-based, fine-tuning-based and prompt-based methods, as illustrated in Figure 2.

2.1 Program Analysis-based Methods

Numerous program analysis-based methods have been proposed and widely used in the industry, such as CheckMarx [25], FlawFinder [58], PCA [29] and RATs [3]. These methods leverage pre-defined rules or patterns designed by experts to identify specific types of vulnerabilities, such as stack-based buffer overflow, heap-based buffer overflow, and so on.

Figure 2 (a) shows an example from Splint [14]. It represents a formal specification designed to express the expected behavior of the `strcpy` function and concurrently provides a rule for detecting potential buffer overflow vulnerabilities. Specifically, it checks the call complies with the condition `maxSet(s1) >= maxRead(s2)`. If the Splint identifies any invocation that contravenes these conditions, it will alert the developer to a possible vulnerability. The advantage of these methods lies in their independence from extensive vulnerability datasets. Moreover, they explain the detected vulnerabilities by reporting the vulnerability-triggering path [9]. This path comprises a sequence of code snippets, thereby facilitating developers' verification processes. However, designing well-defined vulnerability rules or patterns is time-consuming and laborious [30, 31], making it challenging to cover all vulnerabilities.

2.2 Supervised Learning-based Methods

In recent years, many supervised-learning-based methods have been proposed that utilize representation learning techniques to capture vulnerability patterns. It mainly includes the sequence-based [34, 35] and graph-based [6, 33, 65] approaches. Figure 2 (b) illustrates the process of these methods. The sequence-based methods typically use source code as input and learn the corresponding representations for determining whether the given code snippet is vulnerable or not. For instance, SySeVR [34] extracts the code gadget and then uses the bidirectional Long Short-Term Memory network for vulnerability detection. VulCNN [61] transforms source code into images and uses the CNNs to detect vulnerabilities.

Recent studies have shown that graph-based methods ascend in prominence due to their superior interpretability and effectiveness. Compared to sequence-based approaches, these methods extract structured representations from source code, including Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graphs (DFG), and Code Property Graphs (CPG) [55]. Subsequently, GNNs are utilized to learn the graph representations for vulnerability detection. In contrast to program analysis-based methods, these methods can automatically capture vulnerability patterns, thereby mitigating the expenditure of human resources and time-consuming. Nevertheless, the effectiveness of these approaches highly depends on the availability of large and high-quality datasets for training.

2.3 Fine-tuning-based Methods

Although supervised learning-based methods have demonstrated effectiveness for vulnerability detection, Croft et al. [12] have pinpointed that existing vulnerability datasets often lack in quality

and accuracy. It is challenging to apply them in real-world scenarios [57].

Figure 2 (c) shows the process of fine-tuning-based methods [18, 22, 64]. These methods commence with pre-training on a vast corpus of code and textual data, and then fine-tune the pre-trained model for a specific task. For instance, CodeBERT [17] employs the Transformer architecture, utilizing an encoder for its training process. Similarly, CodeT5 [56] and UniXcoder[21] are specifically designed to provide both encoder and decoder in code-related tasks. Through the exploitation of knowledge encapsulated within pre-trained models, these approaches have been shown to excel in vulnerability detection. EPVD [63] introduces an algorithm for the selection of execution paths and leverages a pre-trained model to learn path representations. PILOT [57] proposes a positive and unlabeled framework and uses the pre-trained model to construct the classifier. However, these approaches are limited to the length of input code and exhibit a deficiency in interpretability.

2.4 Prompt-based Methods

In recent years, LLMs have demonstrated superior performance in the fields of Software Engineering (SE) [23] due to their broad generalization and reasoning abilities. Prominent among these developments is the series of generative pre-trained transformer models, developed by OpenAI, including ChatGPT [7] and GPT-4 [40], as well as the LLaMA models unveiled by Meta, comprising both LLaMA [51] and LLaMA2 [52]. Figure 2 (d) presents the process of prompt-based methods [7, 19]. It takes source code as input, subsequently constructs a prompt tailored for vulnerability detection, and feeds this prompt into the LLMs. Then, the LLMs generate a response to detect whether the source code is vulnerable or not. However, these LLMs face notable challenges in software vulnerability detection [19], which primarily stems from two aspects. First, the code snippets often lack enough contextual information for effectively detecting vulnerabilities. Second, LLMs lack the specific domain knowledge required for vulnerability detection, which significantly hampers their performance.

3 VULEVAL SYSTEM

In this section, we introduce the evaluation system of VulEval. It mainly includes two parts: data collection, and evaluation task.

3.1 Data Collection

3.1.1 Data Source. Following the previous work [54], the raw data used to build VulEval consists of a vast collection of CVE entries from the Mend [59]. The dataset consists of a total of 4,196 CVE entries, 4,699 vulnerability patches, and 164 vulnerability types in C/C++ programming languages.

3.1.2 Repository Code Collection. For evaluating the inter-procedural vulnerabilities, we further collect the repository source code via three steps: (1) We select the repositories from which we can retrieve complete source code and commit logs via GitHub, Chrome, and Android. (2) For each vulnerability patch, we gather the repository-level source code corresponding to the commit time of the vulnerability patches. (3) For each file in the vulnerability patch, we use Tree-sitter [1] to slice it as function-level code snippets, where each function-level code snippet contains the corresponding repository-level source code separately.

Table 1: Statistics of the dataset.

Set	# Function	# Repository	# Dependency	# Vul-Dependency
Train	185,791/185,656	3,537/2,872	277,408/253,063	7,580/6,848
Valid	23,224/23,312	2,970/349	37,176/40,619	957/813
Test	23,224/23,271	2,984/331	32,949/53,851	1,001/1,877
All	232,239	4,699	347,533	9,538

As shown in Table 1, we collect 4,699 repository-level source code for vulnerability detection. In repository-level vulnerability detection, we also utilize the function-level label of the target function as the repository-level label (i.e., "1" for vulnerability and "0" for non-vulnerability). The target function and the corresponding dependencies are used as a whole sample to serve as the input for the repository-level sample.

3.1.3 Contextual Dependency Extraction. One of the major contributions of VulEval is that VulEval considers the target code snippet's contextual dependency, which refers to the external code functions that are essential for vulnerability detection.

We extract the contextual dependencies of a code snippet through program analysis of its belonging repository with two steps. (1) Before the extraction process, we first construct the repository database for each vulnerability patch, which includes the corresponding repository source code with different header files (i.e., *.h*) and source code files (i.e., *.c* and *.cpp*). (2) Then, we select the code changed file in the vulnerability patch and employ static program analysis tool [42] to extract the dependency elements. We classify them into the "Callee" and "Caller" dependencies. Specifically, "Callee" represents the user-defined function being invoked or executed by the vulnerability patch. The "Caller" denotes the user-defined function of the repository source code responsible for invocating the function in the vulnerability patch.

As shown in Table 1, we extract 347,533 dependencies in the repository-level source code. We also label 9,538 vulnerability-related dependencies (i.e., denoted as "Vul-Dependency"), which are directly involved in code changes of vulnerability patches. All the other dependencies are considered unrelated to the vulnerability.

3.2 Evaluation Task

VulEval involves three evaluation tasks: function-level vulnerability detection, vulnerability-related dependency prediction, and repository-level vulnerability detection, with details as below.

3.2.1 Function-level Vulnerability Detection (Detector). This task aims to predict whether the function contains a vulnerability or not. As shown in Figure 3, Function vulnerability detection focuses solely on the source code of the target prediction function as input, abstaining from incorporating any inter-procedure information beyond the function itself. The goal of this task is to learn a detector f that can be illustrated as follows:

$$f : \mathcal{X} \mapsto \mathcal{Y}, \mathcal{Y} = \{0, 1\} \tag{1}$$

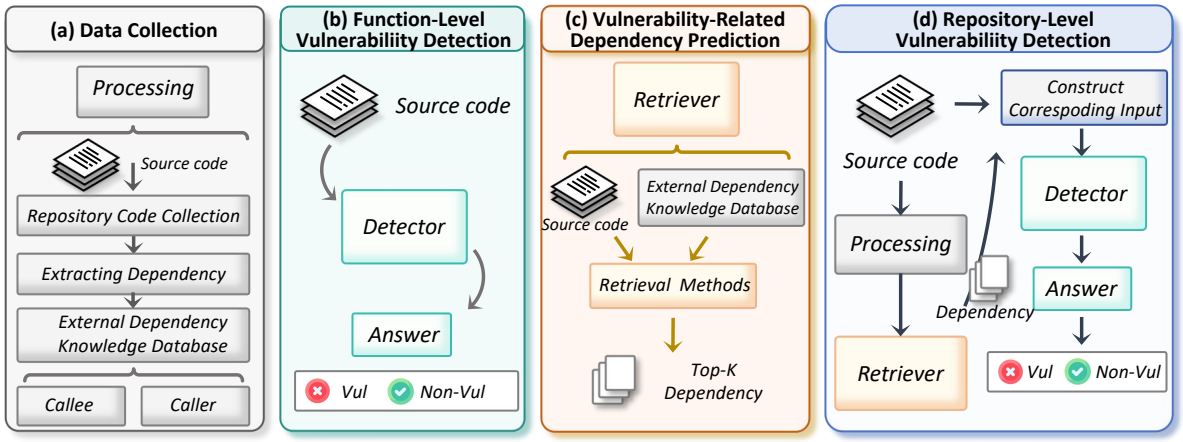


Figure 3: The overview of VulEval. Figure (a), (b), (c), and (d) denote the process of data collection, function-level vulnerability detection, vulnerability-related dependency prediction, and repository-level vulnerability detection, respectively.

where $\mathcal{X}$ denotes the input of function-level code snippet and $\mathcal{Y}$ denotes the label which is set as 1 for vulnerable code snippets and 0 otherwise.

3.2.2 Vulnerability-Related Dependency Prediction (Retriever). The task aims at providing developers with explanations about the vulnerabilities. Table 1 shows that the dataset has 347,533 dependencies, but only 9,538 dependencies are related to vulnerabilities. Thus, it is necessary to retrieve vulnerability-related dependencies from the large number of dependencies in the repository source. As shown in Figure 3 (c), the process of dependency prediction generally involves the "Callee" (*Callee*) and "Caller" (*Caller*) dependency extracted from the input function $\mathcal{X}$, followed by the calculation of the degree of vulnerability-related between the input code snippet and each candidate dependency. The general retrieval function g for identifying vulnerability-related dependency can be formulated as follows:

$$\underset{i,j\in\{1,2,..,m+n\}}{\overset{k}{max}} g(\mathcal{X}, Callee^i, Caller^j) \tag{2}$$

where $Callee^i$ and $Caller^j$ represent the i-th and j-th candidate dependency, respectively, and m and n are the number of "Callee" and "Caller" candidate dependencies, respectively. k denotes the top k relevant dependencies to be retrieved in this task.

3.2.3 Repository-level Vulnerability Detection. Repository-level vulnerability detection is our proposed task, which integrates dependencies identified in the second task for vulnerability detection, as shown in Figure 3 (d). It first uses the "Retriever" to retrieve the associated dependency of the given code snippet. Then, the identified dependencies (i.e., retrieved by "Retriever'), are concatenated with the target function as input. Then, it uses the "Detector" to determine whether the input is vulnerable or not. The definition of repository-level vulnerability detection h can be represented as follows:

$$h : (\mathcal{X}, Callee_{\mathcal{X}}, Caller_{\mathcal{X}}) \mapsto \mathcal{Y}, \mathcal{Y} = \{0, 1\} \tag{3}$$

where the $Callee_{\mathcal{X}}$ and $Caller_{\mathcal{X}}$ denote the retrieved "Callee" and "Caller" dependency from code snippet $\mathcal{X}$, respectively.

4 EXPERIMENTAL SETUP

4.1 Research Questions

Our experiment intends to answer the following research questions:

- **RQ1**: How do program analysis-, supervised learning-, fine-tuning- and prompt-based methods perform in function-level vulnerability detection?
- **RQ2**: How do the retrieval methods perform in identifying the vulnerability-relevant dependency?
- **RQ3**: How do these methods perform in repository-level vulnerability detection?
- **RQ4**: How do these vulnerability detection methods perform for each CWE type?

4.2 Experimental Methodology

4.2.1 Comparison on Vulnerability Detection Approaches. To evaluate the efficacy of vulnerability detection across function-level and repository-level contexts, our benchmark compares four types of vulnerability detection approaches: **(1) Program analysis-based methods**: Following the previous works [5, 61], we select four popular program analysis-based vulnerability detectors, i.e., Cppcheck [11], Flawfinder [58], RATS [3], and Semgrep [44]. These methods leverage predefined rules and patterns to discern potentially improper operations within source code. **(2) Supervised learning-based methods**: We use Devign [65] and Reveal [6] as representative supervised baselines, which are widely adopted as baselines in recent works [4, 33, 57]. These methods construct graphs from source code and then perform vulnerability detection using features obtained through Gated Graph Neural Networks [32]. **(3) Fine-tuning-based methods**: The fine-tuning-based methods consists three general pre-trained models and four state-of-the-art approaches specialized for vulnerability detection. We select three general pre-trained models, CodeBERT [17], CodeT5 [56], and

UniXcoder [21], for their widespread adoption in code-related tasks and further fine-tune these models for vulnerability detection. In addition, we choose four state-of-the-art models designed specifically for vulnerability detection, including PILOT [57], EPVD [63], LineVul [18] and PDBERT [36]. **(4) Prompt-based methods**: We choose two open-source LLMs: LLaMA [51] and CodeLlama [46] for their proficiency in text and code generation, respectively. Additionally, we also incorporate two closed-source LLMs: ChatGPT (i.e., GPT-3.5-*turbo*) and GPT-3.5-instruct, developed by OpenAI, which produce text with 175 billion parameters. For these LLMs, we utilize the process described in Section 2 (d) to assess their effectiveness in vulnerability detection.

4.2.2 Comparison on Dependency Prediction Approaches. We first extract all functions from the call graph as dependency candidates. Then, three types and seven baselines are employed for the vulnerability-related dependency prediction task: **(1) Random method:** This method retrieves code snippets randomly, serving as a foundational baseline for evaluating other prediction methods. To mitigate sampling bias, we repeat this randomized process 100 times and report the average results. **(2) Lexical-based methods:** We evaluate the relevance of vulnerability dependencies using two primary metrics as baselines: Jaccard Similarity and Edit Similarity [49]. Additionally, we use BM25 [45] and BM25+ [53] as lexical-based baselines weighting functions to rank dependencies by their relevance to specific code snippets. **(3) Semantic-based methods:** We leverage pre-trained models as backbone, specifically CodeBERT [17] and UniXcoder [21]) to obtain feature embeddings and then employ Cosine Similarity [62] to measure the semantic relevance between code and dependency snippets.

4.3 Evaluation Metrics

4.3.1 Metrics for Vulnerability Detection Task. We use the following four widely-used performance metrics for vulnerability detection:

Precision: It is calculated as the ratio of true positives (TP) to the sum of true positives and false positives (FP), expressed as Precision = $\frac{\text{TP}}{\text{TP+FP}}$. It signifies the proportion of correctly identified vulnerabilities among all retrieved vulnerabilities.

Recall: Recall is computed as the ratio of TP to the sum of TP and false negatives (FN), given by Rec = $\frac{\text{TP}}{\text{TP+FN}}$. It represents the proportion of vulnerabilities detected by baselines out of all vulnerabilities.

F1 Score (F1): F1 score is defined as the harmonic mean of precision and recall, calculated using the formula F1 = $2 \times \frac{\text{Pre}\times\text{Rec}}{\text{Pre+Rec}}$. It serves as a combined measure of precision and recall, providing insight into the balance between them.

Matthews Correlation Coefficient (MCC): MCC is a measure of binary classification, particularly useful in imbalanced datasets, computed as MCC = $\frac{\text{TP}\times\text{TN}-\text{FP}\times\text{FN}}{\sqrt{(\text{TP+FP})(\text{TP+FN})(\text{TN+FP})(TN+\text{FP})}}$, where TN denotes the true negatives.

4.3.2 Metrics for Dependency Prediction Task. We propose the following two metrics for identifying dependency:

Precision@K (Pre@K): It is the proportion of correctly predicted dependency amongst the Top-K predicted dependency, calculated as follows: $Pre@K = \frac{MATCH_k}{k}$. where $MATCH_k$ denotes the count of correctly predicted dependencies among the Top-K predicted dependency.

Recall@K (Rec@K): It is the proportion of correctly predicted dependency amongst the ground-truth dependency, which is computed as $Rec@K = \frac{MATCH_k(m)}{GT}$, where GT represents the total count of ground-truth, vulnerability-related dependencies.

4.4 Data Split

In this paper, we experiment under the following two settings: **(1) Random Split**: Following the previous work [6, 65], we randomly split the datasets into disjoint training, validation, and test sets in a ratio of 8:1:1. **(2) Time Split**: To mitigate the risk of data leakage and effectively evaluate the methods' ability to identify emerging vulnerabilities, we adopt a time-split setting based on the "commit date" of vulnerability patches. We divide the dataset into training, validation, and test sets in an 8:1:1 ratio. Specifically, patches before 2018-03-21 are designated for training, those before 2022-07-21 constitute the validation set, and patches after this date are used for the test set.

4.5 Implementation Details

For program analysis-, supervised-, and fine-tuning-based methods, we directly use the replication packages and hyper-parameters that have been made publicly accessible. For prompt-based methods, we downloaded LLaMA (i.e., 7B and 13B) and CodeLlama (i.e., 7B and 13B) from the HuggingFace Hub [24] and deploy them locally by the vLLM [26] framework. For ChatGPT ("gpt-3.5-*turbo-0301*") and GPT-3.5-instruct ('gpt-3.5-*turbo-instruct*"), we use the public APIs and initial parameters setting provided by OpenAI. All evaluations are conducted on a server equipped with four NVIDIA A100-SXM4-40GB.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness in Function-level Vulnerability Detection

5.1.1 Effectiveness in Random Split Setting. To answer RQ1, we compare the four types of vulnerability detection methods, including program analysis-, supervised learning-, fine-tuning-, and prompt-based methods. The results are shown in the middle column of Table 2.

Fine-tuning-based methods demonstrate superior performance compared to other methods in the random split. Specifically, these methods yield average results of 51.80% in precision, 38.97% in F1 score, and 39.03% in MCC in the random setting. We also observe that fine-tuning-based methods fall short in terms of recall, with an average of 32.13%, compared to prompt-based methods, which achieve an average result of 55.80% in terms of recall. In the broader evaluation for Top-3 performance across four metrics (comprising 12 instances), these methods demonstrate superiority in 9 out of 12 cases, achieving the highest precision, F1 and MCC, with a score at 63.64%, 42.47% and 41.80%, respectively.

The program analysis- and supervised learning-based methods consistently exhibit inferior performance across all metrics. Program analysis-based methods often target only specific vulnerability types and consequently yield poor results in

Table 2: The experimental results of function-level vulnerability detection in random and time split settings. Bold text cells represent the best performance. The cells in grey represent the performance of the top-3 best methods, with darker colors representing better performance.

Split Methods		Random				Time			
Type	**Baseline**	**Precision**	**Recall**	**F1**	**MCC**	**Precision**	**Recall**	**F1**	**MCC**
Program Analysis	Cppcheck	12.12	1.79	3.12	3.61	19.43	4.34	7.10	7.45
	Flawfinder	6.54	24.78	10.35	7.65	8.55	32.52	13.54	9.73
	RATS	7.06	12.54	9.04	5.80	11.18	20.13	**14.38**	10.14
	Semgrep	10.36	7.76	8.87	6.64	8.37	6.67	7.42	7.45
Supervised Learning	Devign	38.36	24.26	29.72	28.88	9.41	5.26	6.75	3.95
	Reveal	5.95	33.35	10.08	7.96	7.20	24.68	10.99	5.83
Fine-tuning	CodeBERT	51.45	31.79	39.30	39.09	13.85	2.86	4.74	4.56
	CodeT5	51.83	35.97	**42.47**	**41.80**	17.23	5.40	8.23	7.58
	UniXcoder	**63.64**	18.81	29.03	33.66	13.36	4.13	6.31	5.31
	PILOT	49.01	33.28	39.64	38.96	4.26	**91.63**	8.15	2.74
	EPVD	46.84	35.33	40.28	39.18	12.76	4.41	6.55	5.39
	LineVul	47.95	34.93	40.41	39.44	12.79	2.97	4.82	4.31
	PDBERT	51.89	34.78	41.64	41.11	**34.97**	5.30	9.20	**12.32**
Prompt	LLaMA-7B	2.73	68.66	5.25	-1.51	4.17	70.66	7.88	0.87
	LLaMA-13B	2.87	57.46	5.47	0.00	3.91	57.84	7.33	-0.90
	CodeLlama-7B	0.88	**72.09**	1.74	-2.94	2.48	75.65	4.81	-3.33
	CodeLlama-13B	2.26	50.45	4.33	-4.98	3.28	53.60	6.18	-5.51
	GPT-3.5-instruct	4.02	53.58	7.48	5.37	5.10	46.93	9.20	4.09
	ChatGPT	7.38	32.55	12.03	10.44	9.69	26.69	14.22	10.13

general vulnerability detection. The pre-trained models utilized in fine-tuning and prompt-based methods learn more general knowledge during the pre-training phase, thereby endowing them with superior efficacy than supervised learning-based methods.

5.1.2 Effectiveness in Time Split Setting. We also evaluate all baseline methods in the time split setting to comprehensively verify their effectiveness against real-world scenarios without data leakage. The results are shown in the right column of Table 2.

Degradation in performance of supervised learning- and fine-tuning-based methods within time split. Analyzing the results in Table 2, we observe that fine-tuning-based methods exhibit a substantial decrease in all four metrics, showcasing an average decrement of 36.20% in precision, 15.46% in recall, 32.11% in F1 score, and 33.00% in MCC. Similarly, the supervised learning-based methods also demonstrate a decline in performance across four metrics by 13.85%, 13.84%, 11.03%, and 13.53%, respectively. This can be attributed to their heavy reliance on extracting semantics from historical data, rather than directly capturing vulnerability patterns. However, most vulnerabilities are discovered much later than they are introduced. Consequently, we can achieve that these methods struggle to identify new emerging vulnerabilities in real-world scenarios.

The performance of program analysis- and prompt-based methods are not influenced under the time-split setting. Our empirical results indicate that program analysis achieved superior performance due to the predefined rules, with average F1 score and MCC from 7.85%, 5.93% in the random setting to 10.61%, 8.69% in the time-split setting, respectively. Besides, ChatGPT demonstrates near-optimal performance on both F1 score and MCC metrics at 14.22% and 10.13%, respectively. Notably, ChatGPT is trained solely on data up to September 2021, which avoids the data leakage problem. The ChatGPT's performance can be attributed to its vast training corpus containing general knowledge, which enables it to maintain consistent performance across diverse data distributions.

Summary for RQ1: Experiment results reveal that fine-tuning-based methods exhibit superior performance in the random split setting. We also observe a performance degradation in supervised learning and fine-tuning-based baselines within a time-split setting. In addition, the program analysis and prompt-based methods are not influenced within the time-split setting, thereby preserving efficacy across real-world scenarios.

5.2 RQ2: Effectiveness in Vulnerability-related Dependency Prediction

To answer RQ2, we evaluate the performance of three types of methods under both random-split and the time-split settings. We present the top-1,3,5 Pre@k and Rec@k in Table 3.

Superior performance of lexical-based methods for identifying dependency. The experimental results show that both lexical and semantic-based techniques enhances performance, yielding the averagely improvements of 10.21% in Pre@1 and 5.74% in Rec@1 for identifying dependency. Notably, the lexical-based retrieval techniques contribute the largest improvements, with the consistent improvements 3.44% ∼ 18.83% and 3.45% ∼ 18.91% of

Table 3: The experimental results of vulnerability-related dependency prediction in the random and time-split settings. The shaded cells represent the performance of the best methods in each metric. Bold text cells represent the best performance.

Split Methods		Random						Time					
Type	Baseline	Pre@1	Pre@3	Pre@5	Rec@1	Rec@3	Rec@5	Pre@1	Pre@3	Pre@5	Rec@1	Rec@3	Rec@5
Random	Random	56.90	68.60	75.26	30.48	55.46	68.67	55.36	68.11	78.67	32.26	62.42	75.77
Lexical	Jaccard Similarity	**69.70**	70.68	**77.94**	**37.34**	57.14	**71.10**	**68.04**	72.33	81.33	**39.65**	66.29	78.33
	Edit Similarity	67.27	**73.09**	76.51	36.04	**59.09**	69.81	67.77	**76.18**	**86.17**	39.49	**69.82**	**82.99**
	BM25	62.42	70.68	76.87	33.44	57.14	70.13	63.91	71.45	80.50	37.24	65.49	77.53
	BM25+	67.27	70.28	77.22	36.04	56.82	70.45	67.22	71.45	80.33	39.17	65.49	77.37
Semantic	CodeBERT	68.48	72.69	75.44	36.69	58.77	68.83	64.19	72.33	78.00	37.4	66.29	75.12
	UnixCoder	61.82	69.48	76.87	33.12	56.17	70.13	**68.04**	73.03	81.00	**39.65**	66.93	78.01

Pre@k and Rec@k ($k = 1, 3, 5$), respectively. When $k = 1$, the performance benefits more from the knowledge associated with retrieval techniques. For instance, all retrieval methods show an average increase of 16.27% in Pre@1, 3.72% in Pre@3, and 2.06% in Pre@5 compared to random method. Moreover, the semantic-based retrieval methods show moderate performance. This may be attributed to pre-trained models' focus on general semantics rather than domain-specific vulnerability knowledge, indicating that incorporating vulnerability-specific characteristics in retrieval methods is beneficial.

Jaccard Similarity and Edit Similarity outperform in random and time split settings, respectively. As shown in Table 3, Jaccard Similarity is the most effective method under the random split setting, demonstrating superiority in 4 out of 6 cases. It achieves the best performance 69.70% of Pre@1 and 37.34% of Rec@1, respectively. The Edit Similarity performs best in the time split setting, which outperforms the Jaccard Similarity by 3.85% and 3.53%, with respect to Pre@3 and Rec@3, respectively. This finding implies that the retrieving common tokens between code snippets and vulnerability-related dependencies is effective on identifying dependency.

Summary for RQ2: Our empirical analysis indicates that lexical-based methods yield superior results in identifying dependency. Specifically, the Jaccard Similarity and Edit Similarity achieve the best performance in the random and time-split settings, respectively.

5.3 RQ3: Effectiveness in Repository-level Vulnerability Detection

This research question aims to investigate whether integrating vulnerability-related dependencies can enhance the existing vulnerability detection methods' performance. We employ two strategies: "Upper" and "Prediction" to evaluate the baselines performance. "Upper" refers to incorporate the vulnerability-related dependency as input for vulnerability detection. "Prediction" represents the most effective retrieve method as identified in RQ2 (i.e., Jaccard Similarity in random split setting and Edit Similarity in time-split setting). For the repository-level vulnerability detection, due to the input length limited, we only evaluate the fine-tuning- and prompt-based methods. The experimental results are presented in Table 4.

The incorporation of vulnerability-related dependency contexts improves vulnerability detection performance. We observe that repository-level approaches that utilize the "Upper" strategy generally outperform function-level methods previously mentioned. Specifically, when applying the "Upper" strategy in fine-tuning-based methods, we can observe performance enhancements in five out of six baselines in VulEval. Except for PILOT, these repository-level methods demonstrate an average improvement over the corresponding baselines of 7.43% in precision, 3.38% in recall, 4.91% in F1 score, and 5.24% in MCC. This suggests that incorporating vulnerability-related dependencies provides additional contextual information, which allows the model to leverage a more comprehensive understanding of the code repository. The observed performance decline in PILOT may be attributed to its weakly supervised learning, which can be a consequence of an excess of unlabeled samples in the dataset.

Larger models benefit more from the repository-level vulnerability-related knowledge. Our experimental findings indicate that the benefits derived from repository-level information are marginal for the LLaMA and CodeLlama. It may be due to the limitations of the model's abilities for capturing vulnerability patterns. In contrast, ChatGPT exhibits performance improvements across all four evaluation metrics in combing repository-level dependencies, with increases of 11.60%, 24.43%, 14.48%, and 26.35%, respectively. These results suggest that models with a larger foundational architecture possess superior comprehension abilities when dealing with extensive textual input.

It is imperative to explore more effective retrieval methods for identifying dependency. Despite utilizing the most effective retrieval approach identified in RQ2 for identifying dependencies, combining it with existing repository-level vulnerability detection techniques does not greatly enhance the performance. For instance, under a random setting, ChatGPT employing Edit Similarity yields improvements of 4.07%, 10.63%, 5.24%, and 8.43% respectively across the four metrics. However, using Edit Similarity under the time-split setting is not effective. These observations underscore the need for advancements in retrieval strategies to better capture and leverage vulnerability-related dependency.

Table 4: The experimental results of repository-level vulnerability detection in the two split settings. The dark and light shaded cells represent the best performance by using vulnerability-related and predicted dependency, respectively.

Split Methods			Random				Time			
Type	Baseline	Strategy	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
Fine-tuning	CodeBERT	Upper	56.75	33.88	42.43	42.60	26.63	5.61	9.27	10.63
		Prediction	50.51	29.61	37.34	37.32	20.88	4.03	6.75	7.57
	CodeT5	Upper	52.82	37.76	**44.04**	**43.29**	23.59	9.75	13.79	12.93
		Prediction	49.66	32.14	39.02	38.54	17.03	5.72	8.56	7.73
	UniXcoder	Upper	57.53	31.34	40.48	41.25	32.68	7.10	11.66	13.68
		Prediction	54.39	28.57	37.46	38.17	25.22	6.14	9.88	10.72
	PILOT	Upper	**69.78**	14.48	23.98	31.01	41.09	11.23	17.64	21.56
		Prediction	68.00	12.65	21.33	28.57	20.00	2.33	4.17	5.57
	LineVul	Upper	57.63	32.69	41.71	42.18	20.81	6.57	9.98	9.67
		Prediction	55.71	29.02	38.16	38.98	18.68	5.40	8.38	8.08
	PDBERT	Upper	57.72	34.03	42.82	43.09	**47.08**	11.97	**19.09**	**22.26**
		Prediction	54.57	28.42	37.38	38.14	26.43	3.92	6.83	8.82
Prompt	LLaMA-7B	Upper	1.71	5.37	2.59	-2.22	2.14	4.45	2.89	-2.94
		Prediction	1.55	4.91	2.36	-2.54	1.70	3.60	2.31	-3.65
	LLaMA-13B	Upper	2.04	15.22	3.60	-2.65	2.32	11.12	3.84	-4.32
		Prediction	2.06	15.33	3.63	-2.60	2.35	11.12	3.88	-4.21
	CodeLlama-7B	Upper	2.35	29.70	4.36	-2.41	2.67	23.83	4.80	-5.31
		Prediction	2.11	26.79	3.91	-3.56	2.65	23.20	4.76	-5.29
	CodeLLama-13B	Upper	2.19	27.16	4.05	-3.09	2.83	24.05	5.06	-4.51
		Prediction	2.05	24.85	3.79	-3.69	2.94	24.79	5.25	-4.10
	GPT-3.5-instruct	Upper	3.92	**49.70**	7.27	4.70	5.18	**56.46**	9.49	5.07
		Prediction	3.78	48.07	7.00	4.02	5.03	42.27	8.98	3.53
	ChatGPT	Upper	8.61	41.19	14.25	13.69	10.44	32.52	15.80	12.30
		Prediction	7.68	36.01	12.66	11.32	8.83	26.59	13.26	9.03

Summary for RQ3: The experiment results reveal that incorporating contexts related to vulnerabilities enhances the performance of vulnerability detection. It is noteworthy that larger models particularly gain improvement from the integration of vulnerability knowledge at the repository level. In addition, it becomes essential to develop more effective retrieval techniques for identifying vulnerability-related dependencies.

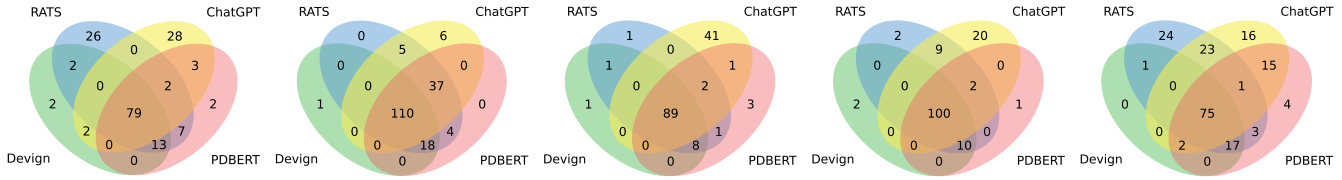
5.4 RQ4: Effectiveness in Each CWE Type Vulnerability Detection

To answer RQ4, we select four methods from different types of methods (i.e., RATS, Devign, PDBERT, and ChatGPT), which perform the best overall performance in their types. We then evaluate these methods in CWE-190, CWE-400, CWE-415, CWE-416, and CWE-787. These vulnerability types represent the most recurrent vulnerabilities, highlighting their elevated potential for software damage. For each type, we deliberately choose 200 representative samples within the time-split setting to avoid the problems of data leakage. Figure 4 shows the number of correctly predicted samples by the four baselines on each of the vulnerability type.

The superior performance of ChatGPT for each CWE vulnerability detection. In the domain of singular vulnerability detection, our analysis reveals that ChatGPT has demonstrated superior performance, correctly identifying 668 samples and achieving 47.53% F1 score averagely. For example, within the CWE-416, ChatGPT can correctly detect 131 samples and achieve 45.67% F1 score respectively, demonstrating effectiveness superior to general vulnerability detection. Moreover, ChatGPT exclusively identified 20 samples, while only 2, 2, and 1 samples can be detected by RATS, Devign, and PDBERT respectively. Therefore, it is practical to leverage LLMs to design a detector for specific CWE type vulnerability in real-world scenarios.

It is worthwhile to explore how to combine the vulnerability detection capabilities of different baselines. Among the 200 samples in CWE-787, the four baselines correctly detect 122 samples in average. The capabilities of the four models are evidenced by their ability to correctly predict 181 samples, showcasing their complementary strengths. Concurrently, there exists a subset of 75 samples that are detectable by any of the four baselines, illustrating the overlap in their detection capabilities. In the future, it is worthwhile to explore how to combine the vulnerability detection capabilities of different methods.

Summary for RQ4: The experimental results reveal that the superior performance of ChatGPT for each CWE type vulnerability detection. Besides, it is worthwhile to explore how to combine different methods' capabilities for software vulnerability detection in the future.



(a) CWE-190 (b) CWE-400 (c) CWE-415 (d) CWE-416 (e) CWE-787

Figure 4: The experimental results of several vulnerability types, including CWE-190, CWE-400, CWE-415, CWE-416, and CWE-787. The green, blue, red, and yellow circles denote the results of Devign, RATS, PDBERT, and ChatGPT, respectively.

6 DISCUSSION

6.1 Implications of Findings

In this section, we discuss the implications of our work for software vulnerability detection. Our experimental results also show potential research directions in the era of software vulnerability detection. Specifically:

(1) For RQ1, fine-tuning-based methods exhibit superior performance in the random split setting. They need to consider time factors to be more effective in real-world scenarios. The program-analysis- and prompt-based methods effectively are not affected by time-split setting. Leveraging LLMs and prompt techniques can be a solution for alleviating performance degradation by time-split setting, thereby enhancing the applicability in real-world scenarios.
(2) For RQ2, using lexical-based methods for identifying vulnerability-related dependency leads to relatively better performance than other semantic-based methods. However, the number of dependencies is not consistent across the code samples. Therefore, how to automatically identify the dependencies needs to be further investigated.
(3) For RQ3, incorporating contexts related to vulnerabilities in repository-level vulnerability detection enhances the performance compared with function-level vulnerability detection. Moreover, the larger LLMs benefit more from the repository-level vulnerability-related knowledge. The retrieval techniques for predicting vulnerability-related dependency are one major bottleneck for improving the performance of current repository-level approaches and remain unsolved.
(4) For RQ4, ChatGPT is more effective than other vulnerability detection methods for detecting the specific CWE vulnerability types. In addition, it is worthwhile to explore how to combine different methods' capabilities for software vulnerability detection in the future.

6.2 Threats to Validity

Representativeness of Baselines Selection. A potential threat to the validity of our study arises from the representativeness of the baselines employed in our experiments for vulnerability detection. Owing to the constraints posed by computational resources and the excessively expensive costs associated with the usage of APIs, we refrained from conducting experiments involving a 34B model size, as well as several other contemporary models including CodeGeex [64], StarCoder [28], and GPT-4 [40]. Future research will conduct more comprehensive experiments across broader baselines.

Generalizability on Other Programming Languages. In this paper, our experimental analysis focuses solely on C/C++ programming languages, excluding other popular languages such as Java and Python. However, the system of VulEval can be generalized to other programming languages because the approach does not rely on language-specific features. In future research, we intend to evaluate the efficacy of VulEval in the context of a broader range of programming languages.

Implementation of baselines. To replicate the baselines, we meticulously use the methodologies delineated in the open-source codes and the original papers. However, owing to the unavailability of the implementation details and hyper-parameters for Devign [65], our reproduction is guided by Reveal's implementation [6].

7 RELATED WORK

We have elaborated on the vulnerability detection methods in Section 2, and focus on illustrating the vulnerability datasets in this section. These datasets can be broadly classified into three groups: function-, slice- and file-level. The function-level datasets [8, 16, 65] utilize cases crafted artificially or extract the functional source code from the real-world scenarios code snippets. For example, SARD [39] constructs samples by manual checking and industrial production. Reveal [6] collects patches from open-source repositories and extracts function-level data from code changes in the patches. The primary limitation of these function-level datasets is constrained by the context-provided code segments. The slice-level datasets [10, 43] typically employ pre-defined rules or static tools to extract DFG and CFG from the source code. For example, Vuldeepecker [35] constructs Code Gadget Database (CGD) by generating code gadgets, consisting of CWE-119 and CWE-399 vulnerabilities. μVuldeepecker [66] expands upon this approach by collecting the 40 types of vulnerabilities and their corresponding labels. The file-level datasets [13, 20, 27] generally provide entire files from vulnerability patches. Such datasets present a comprehensive snapshot of the source code, which is beneficial in detecting the broader context within which vulnerabilities may exist. For example, CrossVul [38] constructs the dataset spanning 40 programming languages and 1,675 projects, while it only provides file-level source code.

However, the previous works more focus on collecting function/file-level data, and ignore repository-level dependencies which are important for detecting inter-procedural vulnerability.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a holistic multi-level evaluation system VulEval, aiming at evaluating the software vulnerability detection performance of inter- and intra-procedural vulnerabilities simultaneously. Specifically, VulEval consists of three evaluation tasks: function-level vulnerability detection, vulnerability-related dependency prediction and repository-level vulnerability detection. VulEval also consists of a large-scale vulnerability dataset. By evaluating 19 vulnerability detection methods on the data split randomly and by time respectively, we observe that incorporating vulnerability-related dependencies facilitates repository-level vulnerability detection performance compared with function-level vulnerability detection. Our analysis highlights the current progress and future directions for software vulnerability detection. In the future, we will explore more aspects of repository-level vulnerability detection such as designing retrieval methods for identifying vulnerability-related dependencies and integrating dependency information in prompts.

REFERENCES

[1] 2023. *Tree-sitter.* https://tree-sitter.github.io/tree-sitter/

[2] [n.d.]. *CWE-20: Improper Input Validation.* https://cwe.mitre.org/data/definitions/20.html

[3] [n.d.]. Rough Audit Tool for Security. https://code.google.com/archive/p/rough-auditing-tool-for-security.

[4] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. *BGNN4VD*: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[5] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1456–1468.

[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? *CoRR* abs/2009.07235 (2020).

[7] ChatGPT. 2022. ChatGPT. https://chat.openai.com/.

[8] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *RAID*. ACM, 654–668.

[9] Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wangand Zheng Zheng, and Yulei Sui. 2022. How About Bug-Triggering Paths?-Understanding and Characterizing Learning-Based Vulnerability Detectors. *IEEE Transactions on Dependable and Secure Computing*.

[10] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 38:1–38:33.

[11] Cppcheck-team. [n.d.]. "Cppcheck". http://cppcheck.sourceforge.net/..

[12] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. *CoRR* abs/2301.05456 (2023).

[13] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John C. Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *MSR*. IEEE / ACM, 46–57.

[14] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (2002), 42–51.

[15] Facebook. [n.d.]. Infer. https://fbinfer.com/.

[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 508–512.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.

[18] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *MSR*. ACM, 608–620.

[19] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? *CoRR* abs/2310.09810 (2023). https://doi.org/10.48550/ARXIV.2310.09810 arXiv:2310.09810

[20] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2021. Neural software vulnerability analysis using rich intermediate graph representations of programs. *Inf. Sci.* 553 (2021), 189–207.

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 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), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225.

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR* abs/2308.10620 (2023).

[24] Huggingface hub. 2023. HuggingFace. https://huggingface.co/.

[25] Israel. [n.d.]. Checkmarx. https://www.checkmarx.com/.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626.

[27] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. 2017. Software Defect Prediction via Convolutional Neural Network. In *QRS*. IEEE, 318–328.

[28] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023).

[29] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1621–1625.

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 141:1–141:29.

[31] Yue Li, Tian Tan, Anders Moller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40.

[32] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016*.

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/SIGSOFT FSE*. ACM, 292–303.

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[36] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. *CoRR* abs/2402.00657 (2024).

[37] mimecast. [n.d.]]. The history of Clop ransomware. https://www.mimecast.com/content/clop-ransomware/

[38] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit

data. In *ESEC/SIGSOFT FSE*. ACM, 1565–1569.
[39] NIST. 2022. SARD: Software assurance reference dataset. https://samate.nist.gov/SRD/index.php.
[40] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).
[41] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. *CoRR* abs/2307.09163 (2023).
[42] Sergey Poznyakoff. 2005. "GNU cflow". https://www.gnu.org/software/cflow/.
[43] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 147:1–147:25.
[44] r2c. 2021. "Semgrep". https://semgrep.dev.
[45] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389.
[46] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).
[47] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *ICMLA*. IEEE, 757–762.
[48] Statista. 2024. Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2024 YTD. https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/
[49] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443.
[50] Rahul Telang and Sunil Wattal. 2007. An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price. *IEEE Transactions on Software Engineering* 33, 8 (2007), 544–557. https://doi.org/10.1109/TSE.2007.70712
[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).
[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).
[53] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and Language Models Examined. In *Proceedings of the 2014 Australasian Document Computing Symposium, ADCS 2014, Melbourne, VIC, Australia, November 27-28, 2014*, J. Shane Culpepper, Laurence Anthony F. Park, and Guido Zuccon (Eds.). ACM, 58.
[54] Chaozheng Wang, Zongjie Li, Yun Peng, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, and Michael R. Lyu. 2023. REEF: A Framework for Collecting Real-World Vulnerabilities and Fixes. In *ASE*. IEEE, 1952–1962.
[55] Xiaomeng Wang, Tao Zhang, Runpu Wu, Wei Xin, and Changyu Hou. 2018. CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning. In *10th International Conference on Advanced Infocomm Technology, ICAIT 2018, Stockholm, Sweden, August 12-15, 2018*. IEEE, 184–188.
[56] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708.
[57] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When Less is Enough: Positive and Unlabeled Learning Model for Vulnerability Detection. *CoRR* abs/2308.10523 (2023).
[58] David A. Wheeler. [n.d.]. Flawfinder. https://dwheeler.com/flawfinder/
[59] WhiteSource. 2023. "Mend bolt". https://www.mend.io/free-developer-tools/.
[60] Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. 2017. Vulnerability detection with deep learning. In *2017 3rd IEEE international conference on computer and communications (ICCC)*. IEEE, 1298–1302.
[61] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2365–2376.
[62] Peipei Xia, Li Zhang, and Fanzhang Li. 2015. Learning similarity with cosine similarity ensemble. *Inf. Sci.* 307 (2015), 39–52.
[63] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability Detection by Learning From Syntax-Based Execution Paths of Code. *IEEE Trans. Software Eng.* 49, 8 (2023), 4196–4212.
[64] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR* abs/2303.17568 (2023).
[65] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*. 10197–10207.
[66] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* 18, 5 (2021), 2224–2236.