

# Assessing and Improving Prompting Large Language Models for Software Vulnerability Analysis

YU NONG, University at Buffalo, SUNY, USA

GUANGBEI YI, Washington State University, USA

MOHAMMED ALDEEN, Clemson University, USA

LONG CHENG, Clemson University, USA

HONGXIN HU, University at Buffalo, SUNY, USA

HAIPENG CAI\*, University at Buffalo, SUNY, USA

Large language models (LLMs) have demonstrated potential in diverse domains including software analysis. Yet there is a lack of systematic *assessment* of how LLMs perform in comparison to various extant approaches, and how LLMs may be *improved*, for software vulnerability analysis, via prompt engineering. In this paper, we present a comprehensive, large-scale *empirical study* of ten LLMs with seven prompting strategies versus nine traditional (five code-analysis- and four deep-learning-based) techniques on three vulnerability analysis tasks (detection, classification, and repair) against five real-world datasets (8,000+ C/C++ samples, including a zero-day dataset). We show that, with existing prompting strategies, LLMs often struggle with practical vulnerability analysis and underperform the traditional approaches. Via in-depth case analysis, we reveal that the evaluated LLMs frequently suffer from incorrect reasoning. Based on these findings, we improve the prompting with a vulnerability-specific adaptation of chain-of-thought (CoT), named Vulnerability-Semantics-guided Prompting (VSP). Our results show that VSP improves the performance of some of the LLMs in certain configurations across the three tasks. VSP also helps mitigate the reasoning limitations for some of the evaluated LLMs. For vulnerability detection on unseen data, improvements are limited or marginal for some models. We further identify seven common challenges that led to the LLMs' incorrect answers in these tasks and provide actionable recommendations to help mitigate them.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: Vulnerability analysis, large language models, prompting, vulnerability semantics, vulnerability detection, vulnerability classification, vulnerability repair

## ACM Reference Format:

Yu Nong, Guangbei Yi, Mohammed Aldeen, Long Cheng, Hongxin Hu, and Haipeng Cai. 2026. Assessing and Improving Prompting Large Language Models for Software Vulnerability Analysis. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2026), 53 pages. <https://doi.org/10.1145/37xxxxx>

\*Haipeng Cai is the corresponding author.

---

Authors' Contact Information: Yu Nong, [yunong@buffalo.edu](mailto:yunong@buffalo.edu), University at Buffalo, SUNY, Buffalo, USA; Guangbei Yi, [guangbei.yi@wsu.edu](mailto:guangbei.yi@wsu.edu), Washington State University, Pullman, USA; Mohammed Aldeen, [mshujaa@g.clemson.edu](mailto:mshujaa@g.clemson.edu), Clemson University, Clemson, USA; Long Cheng, [lcheng2@clemson.edu](mailto:lcheng2@clemson.edu), Clemson University, Clemson, USA; Hongxin Hu, [hongxinh@buffalo.edu](mailto:hongxinh@buffalo.edu), University at Buffalo, SUNY, Buffalo, USA; Haipeng Cai, [haipengc@buffalo.edu](mailto:haipengc@buffalo.edu), University at Buffalo, SUNY, Buffalo, USA.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

1

## 1 INTRODUCTION

Software vulnerabilities are increasingly prevalent [41, 49, 52, 76] and consequential [23, 27, 40]. In response, numerous techniques for a range of defensive tasks (i.e., *detecting* the presence of vulnerabilities [13, 13, 22, 34, 48, 51, 65, 90, 91], *classifying* vulnerabilities of various types [12, 30, 37, 45, 50, 60, 64, 81], and *repairing* the vulnerabilities found [17, 29, 32, 58, 75, 102]) have been proposed. Tackling these *three essential software vulnerability analysis tasks*, existing approaches, including traditional ones based on code analysis [12, 22, 37, 45, 48, 64, 81] and more recent ones based on deep learning (DL) [17, 28, 29, 38, 53, 54, 60, 82, 95, 105], have been both successful and subject to various limitations [9, 15, 42, 67, 69, 71].

Lately, pre-trained large language models (LLMs) have demonstrated potential in diverse domains including software analysis [25, 43, 44, 57, 97, 103]. Some LLM-based techniques have also been proposed for vulnerability analysis [36, 46, 80, 83, 96]. However, these techniques require assistance from external information (e.g., error messages), which may not always be available [46, 80, 96]. Other works fine-tune the open-source LLMs [36, 83], which may require considerable resources and cannot use the most powerful commercial LLMs. These constraints limit the application of LLMs for vulnerability analysis. Therefore, a direct and easy-to-use LLM-based approach for vulnerability analysis is needed. Among various LLM-based strategies, prompt engineering stands out as a highly accessible and practical direction, especially given that pre/re-training or fine-tuning may not be feasible for most users, particularly with commercial LLMs. Yet there is still a lack of comprehensive assessment on how LLMs perform across different vulnerability analysis tasks with different prompting strategies and how to improve them.

Existing studies [21, 75, 84, 87, 104] have taken important steps in this regard. However, they are subject to one or more of the following limitations: (1) address single tasks only (almost all focusing on vulnerability detection), (2) use relatively small datasets (e.g., <400 samples in total [84, 87] or 30 real-world samples [87]), (3) dismiss traditional (i.e., non-LLM-based) approaches, including code analysis- and DL-based ones, in comparisons [75, 84, 104], (4) provide no evaluation on LLM reasoning correctness, and (5) ignore the data leakage/contamination issues [75, 84, 104] (i.e., the LLMs may have seen the testing samples' labels during their training). Moreover, while these studies generally conclude that current LLMs are not yet ready for vulnerability analysis, they (6) only directly use existing prompting strategies without providing or evaluating improvement strategies.

In this paper, we seek to fill these gaps through a comprehensive and large-scale assessment of LLMs with different prompting strategies, and exploration of how to improve them, for vulnerability analysis across three tasks (detection, classification, and repair) against more than 8,000 real-world code samples from four existing datasets and a zero-day dataset (where all the vulnerabilities were released after all the LLMs' training cut-off dates).

In this study, we seek answers to seven research questions:

**RQ1:** (How well) can current LLMs detect, classify, and repair vulnerabilities with existing prompting strategies?

**RQ2:** (How much) can we improve the prompting strategies for LLMs in vulnerability analysis?

**RQ3:** How do LLM configurations and system prompts affect the effectiveness of vulnerability analysis tasks?

**RQ4:** How do LLMs compare to traditional (DL- and code-analysis-based) approaches to the same three essential vulnerability analysis tasks?

**RQ5:** Are the vulnerability analysis results produced by LLMs based on correct reasoning about the vulnerabilities?

**RQ6:** Can we improve the reasoning of LLMs in vulnerability analysis via prompt engineering?

**RQ7:** What are the common challenges to LLMs in analyzing real-world software vulnerabilities?

To answer these questions, we assess ten LLMs with five prompting strategies (standard prompting, standard few-shot learning, retrieval-augmented generation (RAG), naive chain-of-thoughts (CoT) [92], and zero-shot completion [75]) against nine SOTA traditional (five code-analysis-based and four DL-based) approaches. Then, we examine whether the generated reasoning from LLMs captures the *vulnerability semantics* (i.e., behaviors of a program that make it vulnerable) and justifies their responses correctly. Based on the limitations found, we propose a vulnerability-specific CoT adaptation, named *Vulnerability-Semantics-guided Prompting* (VSP). It features task-specific prompting unified across the three tasks via vulnerability semantics, which is represented by the vulnerability-relevant code semantics of a program [7] that capture (i.e., account for) its vulnerable behaviors. Accordingly, VSP uses descriptions of vulnerability-relevant data and control dependencies as exemplars to encourage the model to focus its reasoning on the vulnerability behavior when producing an answer. We also assess the challenges faced by LLMs for vulnerability analysis which lead to incorrect predictions. We further provide actionable recommendations for mitigating these challenges such as leveraging traditional code analysis techniques for control/data flow analysis.

Among other findings, our study reveals that:

- Direct use of existing prompting strategies on LLMs does not work well for vulnerability analysis (with up to 47.06–54.55% F1 for detection, 25.40–41.11% micro F1 for classification, and 20.40–70.83% correct rate for repair) and sometimes *underperform* the traditional approaches (which achieve up to 52.31% F1, 36.23% micro F1, and 83.33% correct rate for detection, classification, and repair, respectively). Against the zero-day dataset, some LLMs perform much worse than they perform on existing datasets, indicating their potential subjection to data leakage/contamination issues. This underscores the critical need for better prompting strategies specific to vulnerability analysis tasks and for using zero-day datasets as the primary benchmark for unbiased evaluation to mitigate data leakage/contamination risks to evaluation validity.
- The LLMs suffer from incorrect reasoning, where only 4.03–14.07% of the reasoning is correct; even *among the correct (final) predictions*, 16.51–69.26% of the reasoning is incorrect, which limits LLMs’ usability and credibility for vulnerability analysis. Meanwhile, (i) the reasoning underlying incorrect responses is consistently incorrect, while (ii) for producing correct responses, different tasks depend on reasoning correctness to different extents—more challenging tasks (e.g., vulnerability repair) have stronger reliance on correct reasoning for producing correct predictions than less challenging tasks (e.g., vulnerability detection). In other words, more challenging tasks are less likely to suffer from *coincidental correctness* of their LLM responses, while LLMs can be more likely to produce coincidentally correct answers for less challenging vulnerability analysis tasks.
- With our proposed vulnerability-specific adaptation of Chain-of-Thought (CoT) prompting (i.e., VSP), some of the LLMs show improved performance for the three tasks in specific evaluated scenarios (achieving F1 scores of 48.70–62.68% for detection, micro F1 of 40.16–47.27% for classification, and correct rates of 30.25–83.33% for repair). VSP also enables certain top-performing LLMs to outperform the evaluated traditional baselines (by up to 25.80% for detection, 234.11% for classification, and 6.10% for repair) and helps mitigate incorrect reasoning (achieving 50.15–94.22% correct reasoning rates for correct final predictions). Our findings show that VSP is associated with improved performance in certain settings, consistent with its design goal of emphasizing vulnerability-focused semantic reasoning.
- Seven common causes can lead to LLMs’ *incorrect predictions* across the three tasks, including *insufficient context* (e.g., LLMs do not know about a self-defined/external function called) and *incomplete data/control flow analysis* (e.g., LLMs miss key data-flow facts as part of the vulnerability semantics) as primary causes; in other cases, LLMs suffer from not following exemplars, logic errors, and unreadable output issues. These novel insights into

Table 1. LLMs examined in our study

Model	Version	#Params	Max #Tokens	Vendor	Release date	Cutoff Date	License
Gemini	Gemini-Pro	50T	30,720	Google	Dec. 2023	Apr. 2023	Commercial
Claude3	Claude-3-Sonnet	70B	200,000	Anthropic	Mar. 2024	Aug. 2023	Commercial
GPT-3.5	gpt-3.5-turbo-16k	175B	16,385	OpenAI	June 2023	Jan. 2022	Commercial
GPT-4	gpt-4o	175B+	128,000	OpenAI	May 2024	Oct. 2023	Commercial
Llama2-7b	llama-2-7b-chat-hf	7B	4,096	Meta	July 2023	Sep. 2022	Open-Source
Llama2-13b	llama-2-13b-chat-hf	13B	4,096	Meta	July 2023	Sep. 2022	Open-Source
Falcon	falcon-7b-instruct	7B	2,048	TII	June 2023	Apr. 2023	Open-Source
CodeLlama	codellama-13b-instruct-hf	13B	4,096	Meta	July 2023	Sep. 2022	Open-Source
Qwen2.5-Coder	qwen2.5-coder-14b-instruct	14B	131,072	Alibaba	Sep. 2024	Oct. 2023	Open-Source
DeepSeek-Coder-V2	deepseek-coder-v2-lite-instruct	16B	131,072	DeepSeek	July 2024	Oct. 2023	Open-Source

why LLMs can fail in vulnerability analysis tasks provide informative guidance for developing future LLM-based vulnerability analysis techniques.

Based on these findings, we provide actionable suggestions for improving LLM-based vulnerability analysis. Among others, we recommend (1) using vulnerability-antics-guided prompting strategies, (2) using LLMs that are capable of following vulnerability-analysis-specific task instructions, especially the exemplars for effective in-context learning, as well as understanding code semantics for vulnerability reasoning, (3) providing necessary context information *on demand* during prompting to both meet the contextualization needs for vulnerability analysis and reduce the total prompt length, and (4) integrating traditional code analysis techniques for advanced LLM-based vulnerability analysis to mitigate LLMs’ insufficiency in data/control flow analysis essential for vulnerability-related code semantics reasoning.

To the best of our knowledge, our work is the first to systematically assess the prompting strategies of LLMs across three essential vulnerability analysis tasks at large scale and compare them with various SOTA traditional approaches to the same tasks, while exploring and evaluating an improving methodology. While LLMs face major challenges for analyzing (especially zero-day) vulnerabilities (i.e., undesirable effectiveness in both prediction accuracy and reasoning correctness), our findings highlight/point to a direction for improving the LLMs for that purpose. The substantial gaps in LLMs’ performance on existing vulnerability (i.e., with labels likely leaked) versus zero-day (i.e., vulnerability labels unseen by models) datasets point to the critical need for future LLM-based vulnerability analysis to be evaluated against testing samples labeled after the model’s cutoff date.

Our datasets and experimental results are available at <https://doi.org/10.5281/zenodo.20585652>.

## 2 METHODOLOGY

In this section, we describe our study setup, including the *LLMs* (§2.1) examined, the vulnerability analysis *tasks* (§2.2) targeted (i.e., goal/motivation and formulation of each task), the *datasets* (§2.3) utilized, the *prompting strategies* (§2.4) considered (existing ones and the improved one), and the *baseline traditional approaches* to vulnerability analysis (§2.6) as we compare LLMs with.

### 2.1 LLMs Assessed

Table 1 summarizes the ten LLMs chosen for our study to cover the models with a variety of sizes, vendors, and specializations. **Gemini** [85] and **Claude3** [4] are the LLMs from Google and Anthropic which can understand and generate natural language and code. We chose their most popular versions (Pro and Sonnet) for both effectiveness and efficiency. **GPT-3.5** [26] is a primary LLM from OpenAI for general purposes, and we chose the Turbo-16k version to allow more input/output tokens. We also consider **GPT-4** which is the more powerful LLM from OpenAI and use its most widely-used version GPT-4o. **Llama2** [86] is an open-source LLM from Meta, of which we chose to deploy 7b

and 13b versions locally due to our resource limitations. For the same reason, we chose the 7b-instruct version of **Falcon** [107], an open-source LLM from the Technology Innovation Institute (TII).

To provide a comprehensive assessment of LLMs' capabilities in software vulnerability analysis, we also include three code-specific LLMs. **CodeLlama** [79], developed by Meta, is designed to generate and discuss code through text prompts, making it a valuable tool for various programming needs. **Qwen2.5-Coder** [39], from Alibaba, is a code-specific model for code generation, reasoning, and fixing. **DeepSeek-Coder-V2** [106] is an open-source model from DeepSeek that offers powerful performance in code-specific tasks.

Finally, to investigate the potential of LLMs beyond prompt engineering, we also evaluate the capability of fine-tuned LLMs. We specifically select Llama2-13b for fine-tuning, as it demonstrated the strongest overall performance among the evaluated open-source models in our preliminary experiments. By fine-tuning this top-performing open-source baseline on our datasets, we aim to rigorously compare the effectiveness of our proposed prompting strategy against the fine-tuning approach.

## 2.2 Tasks

Our study addresses three vulnerability analysis tasks: (1) **detection** (determining if a given code sample is vulnerable or not), (2) **classification** (classifying a vulnerable code sample into a vulnerability type/class (CWE)), and (3) **repair** (fixing a vulnerable sample based on the vulnerability type and location). These tasks span various levels of analysis, representing the key steps in holistic defenses against software vulnerabilities. Note that we aim to assess LLMs for these tasks separately—each assumes what it needs independently, not taking the output of a previous task as its input.

**2.2.1 Task 1: Vulnerability Detection.** In this task, we aim to check the presence of vulnerabilities in a given code sample, a typical defensive vulnerability analysis [15, 28, 105]. This task serves as a starting point for evaluating whether LLMs can recognize vulnerabilities in a broader sense. Note that in this task, the goal is not to detect a specific vulnerability but to determine whether the code contains a vulnerability of any kind. This mirrors real-world vulnerability scanning, where tools are expected to identify any security flaw in a piece of code, regardless of the category of that flaw.

We *formulate* this as a binary classification task. In the input (i.e., prompt) to LLMs, we start with the question “Q: Does the following code have exploitable vulnerabilities?”. We also request the model: “Please check whether the vulnerability, if any, can be practically exploited to cause real security consequences.” This guides the model to detect *exploitable* vulnerabilities rather than just potential/general bugs. The prompt ends with the code under analysis.

**2.2.2 Task 2: Vulnerability Classification.** The *goal* of this task is to, after determining a code sample as vulnerable, further identify the type of vulnerability (in the form of a CWE ID) in it. In practical vulnerability analysis, developers not only need to know whether given code is vulnerable but also the vulnerability type to inform their responses and countermeasure decisions [60].

We *formulate* this as a multi-class classification task. The input to LLMs starts with the question: “Q: The following code has been confirmed as vulnerable, which CWE does it have?”, followed by the code sample under analysis. We also provide the CWE options for the models to choose. That is, the models only need to classify a vulnerability into one of the five CWEs. The reason is that the CWE options are at different levels of abstraction and are related to one another (e.g., CWE-20 is a ‘super type’ of several specific input validation issues, such as SQL Injection CWE-89, XSS CWE-79, etc). By providing the options, we can avoid the conflicts on the CWE abstraction levels. Another reason is that state-of-the-art techniques (e.g., VulChecker [60]) also classify vulnerabilities into one of the given CWE options. To compare LLMs with these (non-LLM-based) techniques fairly, we follow the same approach to this *scoped* vulnerability

classification task. This scoped task formulation also facilitates automatically evaluating LLMs’ response correctness: given the scope of CWE choices, the LLMs never classified a vulnerable code sample into CWEs that are at higher/lower levels than the ground-truth labels.

**2.2.3 Task 3: Vulnerability Repair.** The goal of this task is to eliminate the vulnerability from given vulnerable code while preserving its functionalities. It is a crucial vulnerability-analysis step as it addresses security risks induced by the vulnerabilities hence protecting the respective software from potential threats/attacks [29, 75].

We *formulate* repair as a text-to-text transformation task. The LLM prompt starts with the question: “Q: The following code has a <CWE ID> vulnerability at line <line no.>. Please provide a valid patch.” We also request the model: “only show the code changes needed rather than the entire patched code, to conform to the conventional patch format.” The prompt then ends with the vulnerable code. We explicitly provide the vulnerability location (via the line no.) and type (via the CWE ID), which is common in bug/vulnerability repair [17, 29, 32].

## 2.3 Datasets

We evaluate the chosen LLMs on the three tasks using real-world C/C++ code as *testing* data. We focus on C/C++ language because (1) C/C++ is the most vulnerable language, and (2) many critical software systems are written in C/C++ (e.g., Linux kernel, Qemu, and FFmpeg). Our choice of C/C++ is also aligned with the availability of existing vulnerability datasets (with quality ground truth labels); such datasets for other languages are relatively rare compared to C/C++. We selected datasets containing low-level memory vulnerabilities (e.g., buffer overflows, use-after-free) which represent a class of vulnerabilities that are both highly prevalent and notoriously difficult to address with existing techniques. These types of vulnerabilities, which often involve intricate memory management and pointer arithmetic, are a significant challenge for both traditional tools and modern approaches, including LLMs. Specifically, within each dataset, we focused on five prevalent memory-related CWEs, with slight variations in the exact CWEs selected. The categories of these vulnerabilities that we target are mostly in the list of top-25 most dangerous CWEs [1].

**VulChecker Dataset.** For Tasks 1 and 2, we use the dataset from VulChecker [60] because the samples are compilable (hence allowing us to directly compare LLMs with VulChecker) and fairly diverse (covering 19 real-world projects, 35 CVEs, and 5 CWEs—CWE-121, 122, 190, 415, and 416). In this dataset, each vulnerable project version has one or more vulnerable points each labeled with one of the 5 CWEs. Given its source size, a project version cannot be fed as a whole to LLMs at once. Thus, we consider individual functions as input units as commonly adopted in DL-based vulnerability analysis [15, 28, 54, 55, 105]. To that end, we extract 85 vulnerable functions (11 CWE-121, 15 CWE-122, 40 CWE-190, 7 CWE-415, and 12 CWE-416). For Task 1, we also extract 87,353 non-vulnerable functions from these projects. Since there are too many non-vulnerable functions that cause extreme data imbalance (>1000:1), we follow VulChecker to downsample the non-vulnerable functions to 1%, resulting in 873 ones [60]. Nevertheless, we still kept the dataset imbalanced because non-vulnerable samples are still more prevalent than vulnerable samples in the real-world vulnerability distribution [15, 60], and such imbalanced dataset setting is widely used in previous works [15, 28, 54, 55].

**ExtractFix Dataset.** For Task 3, we use the dataset from ExtractFix [32] because the code therein is compilable and comes with test cases that allow us to validate patches. From this dataset, we collected 24 vulnerabilities that can be reproduced, for each of which we label the vulnerable line and the CWE ID based on the crash information from the test cases. These 24 also cover 5 CWEs (7 CWE-787, 9 CWE-125, 2 CWE-190, 4 CWE-369, and 2 CWE-476). Again, we use function-level samples.

**BigVul+CVFixes.** However, the two datasets are relatively small for a large-scale study. Thus, we further build a larger dataset where the samples are collected from BigVul [24] and CVFixes [11], two of the most widely used real-world vulnerability datasets [28, 29, 38, 68, 70]. We select the vulnerable samples with the same CWEs as in VulChecker and ExtractFix, resulting in 938 vulnerable samples (149 CWE-121, 159 CWE-122, 333 CWE-190, 112 CWE-415, and 185 CWE-416) for Tasks 1 and 2, as well as 236 vulnerability repair samples (126 CWE-787, 48 CWE-125, 43 CWE-190, 6 CWE-369, and 13 CWE-476) for Task 3. For Task 1, we also select the non-vulnerable samples from BigVul and downsample to 5,892 samples to avoid the extreme imbalance. We remove any overlap between BigVul and CVFixes to avoid duplicates.

**Zero-Day Dataset.** However, the datasets above include samples from many years ago. It is possible that these samples were used to train the LLMs, causing potential data leakage/contamination issues. To mitigate this, we constructed a Zero-Day dataset where all the vulnerabilities were reported after the latest cutoff date of the LLMs evaluated (October 2023 from GPT-4o). The construction process was as follows:

- (1) **CVE Collection:** We first developed a crawler to automatically collect the latest CVEs from the NVD website [63].
- (2) **Date and CWE Filtering:** We then filtered these collected CVEs to include only those with a publishing date after the LLMs' latest training cutoff date (October 2023) and that correspond to the most common five CWEs used in BigVul+CVFixes (CWE-787, CWE-125, CWE-190, CWE-416, and CWE-476).
- (3) **Commit Link Verification:** We further filtered for CVE reports that provided a direct commit link for patching the vulnerability. This step was essential to ensure the availability of both the vulnerable source code and the ground truth patches, which are necessary for our evaluation.
- (4) **Function-Level Sample Extraction:** Finally, from the collected commits, we manually extracted function-level samples written in C/C++. For each relevant commit, we checked the file types (e.g., .c, .cpp) of the changed code. We then manually filtered out changes irrelevant to the vulnerabilities (e.g., documentation updates, release notes) and extracted the vulnerable functions both before and after patching.

With the process, we collected a Zero-Day dataset with 103 vulnerabilities which also cover five CWEs (18 CWE-787, 26 CWE-125, 11 CWE-190, 24 CWE-416, 24 CWE-476) for the three tasks. The vulnerable and fixed versions of these samples are used for Task 1, while the vulnerable versions are used for Tasks 2 and 3.

**PrimeVul Dataset.** Recently, Ding et al. [21] point out the issue of low label accuracy in the previous datasets (e.g., BigVul and CVFixes), where up to 75% of the sample labels are inaccurate. To address this, they introduced PrimeVul, improving label accuracy to 92%. Given this higher quality, we employ PrimeVul to evaluate LLMs on vulnerability analysis tasks. We again select the five most common CWEs in BigVul+CVFixes (CWE-121, CWE-122, CWE-190, CWE-416, and CWE-476), and select 10% of the non-vulnerable samples from its testing set (considering the cost of LLMs and the balance of the dataset). As a result, the testing set we use from PrimeVul comprises a total of 2,635 function-level samples, of which 72 are vulnerable, which we call PrimeVul-Subset. The 72 vulnerable samples cover 1 CWE-121, 7 CWE-122, 14 CWE-190, 12 CWE-415, and 38 CWE-416. For Task 1, all 2,635 samples from the PrimeVul testing set are utilized. For Task 2 and Task 3, only the 72 vulnerable samples are used.

Besides, to address the limited representativeness of a restricted CWE subset and to facilitate broader cross-referencing, we also evaluate on the PrimeVul dataset with all the CWEs included in its testing dataset. This results in 435 vulnerable samples spanning 62 CWEs. To balance the dataset, we include 15,920 non-vulnerable samples. For Task 1, the respective vulnerable and non-vulnerable sets are utilized. For Tasks 2 and 3, only the vulnerable samples are used. We refer to this dataset as PrimeVul-Full in the evaluation. For the distribution of the CWEs, refer to the original PrimeVul paper [21].

```
// Line-by-line code summarization (used in Naive CoT)
The cycle0 function first initializes two integer variables `v` and
`w` and modifies `v` via a function call to `set`. Next, it
retrieves the number of offers from the hello object. Then, it
declares an array offered of size `max_offered + 1` and iterates
over the number of offers, fetching each offer and storing it in
the offered array while updating `w` by adding `v` in each
iteration. After that, it sends the final value of `w` using the
`send` function and calls the `reset` function to reset some state.
```

Fig. 1. Reasoning steps (describing the *functionality semantics*) in naive CoT (for the example in Fig. 2).

ensure the quality of the fine-tuning, we utilized all available samples for each task, resulting in training sets of 330,492 samples for vulnerability detection, 16,109 for classification, and 12,808 for repair.

## 2.4 Prompting strategies

In this study, we study different prompting strategies to evaluate the inherent capabilities of LLMs and their responsiveness to semantic guidance.

**2.4.1 Existing Prompting strategies.** We examine five popular/well-known strategies for prompting LLMs.

**Standard Prompting.** We ask the LLMs to solve a task without providing any exemplars or guidance, aiming to assess their raw/original capabilities for the task.

```
// Code to analyze
1 void cycle0(struct st *hello){
2   int v = 0;
3   int w = 1;
4   set(&v);
5   int num = hello->getOfferNum();
6   int offered[max_offered + 1];
7   for (int i = 0; i < num; i++) {
8     offered[i] = hello->getOffer(i);
9     w += v;
10  send(w);
11  reset();
}

// Vulnerability semantics description (used in VSP)
First, `num` is obtained from the input variable `hello`
via `hello->getOfferNum()`, we thus should be careful what
values `num` might take. Next, the size of the `offered`
array is fixed at `max_offered + 1` and the loop iterates
`num` times. If `num` is greater than `max_offered + 1`,
then `offered[i]` will access indices beyond the bounds of
the array, the key security-related behavior of this code.
```

Fig. 2. Illustration of *vulnerability semantics* (as opposed to *functionality semantics* shown in Fig. 1) on a buffer-overflow code sample, with the approximation (slice) highlighted at the top and description at the bottom. The sensitive statement is marked in orange; the relevant contextual statements which make the sensitive statement reachable from the external inputs are marked in yellow, and those it data-depend on in cyan. The approximating data/control dependencies are denoted with  $\rightarrow$ .

to provide the LLM with highly relevant in-context examples by dynamically retrieving them based on the input query, thereby potentially mitigating issues like insufficient context and improving the quality of few-shot learning [33]. For our RAG experiments, given a testing sample, we first use the BM-25 algorithm [77] to find the most similar samples from a predefined exemplar set, as the BM-25 algorithm is very fast with good performance for code comparison [20, 31, 62]. Similar to standard few-shot learning, we select a total of 10 exemplars for each prompt. These retrieved samples are then used as exemplars for standard few-shot learning, meaning that we provide the questions, code samples, and final

**DiverseVul Dataset.** To evaluate the capabilities of fine-tuned LLMs, we utilize the DiverseVul [16] dataset, a large-scale collection of real-world vulnerable code. To prevent data leakage and ensure the validity of our evaluation, we filtered the dataset to remove any samples that overlap with our testing datasets (VulChecker, ExtractFix, BigVul+CVEFixes, Zero-Day, and PrimeVul). To en-

**Standard Few-Shot (Learning).** In each prompt, we provide exemplars which include the questions, code samples, and the final answers without any reasoning steps. This is to assess whether a simple instantiation of few-shot learning can help LLMs with vulnerability analysis. In this study, we manually construct exemplars with the same CWEs as the testing samples. In particular, for each of the 5 CWEs in our datasets, we provide two exemplars. This results in a total of 10 exemplars for prompting—we target 10 as in [92]. We curate the exemplar code samples using the demonstrative examples (with appropriate modification) on the official CWE website [1]. For Task 1, we also provide 10 non-vulnerable exemplars to balance.

**Retrieval-Augmented Generation (RAG).** To address concerns about insufficient code contexts and to explore more advanced few-shot techniques, we incorporate Retrieval Augmented Generation (RAG). RAG aims

answers of the exemplars in the prompt. To ensure diversity and avoid data leakage within the same dataset during retrieval, we cross-use our datasets (VulChecker, ExtractFix, BigVul+CVEFixes, Zero-Day, and PrimeVul) for exemplar retrieval: when testing on one dataset, samples from the other datasets are used as the pool for the exemplar selection.

**Naive CoT (based Few-Shot Learning).** This strategy extends standard few-shot learning by incorporating a basic form of Chain-of-Thought (CoT) [92]. In addition to the question, code, and final answer, each exemplar includes a line-by-line analysis of the code, mainly describing its functionality semantics without focusing on security-relevant code. As illustrated in Figure 1, this analysis proceeds sequentially through the code, explaining what each significant line or block of code does. The rationale behind this 'naive' approach is to evaluate whether a straightforward, generic CoT, without deep vulnerability-specific guidance, can prompt LLMs to generate more structured and accurate reasoning. This is to evaluate whether LLMs perform better on vulnerability analysis by giving exemplars with line-by-line code semantics analysis. We used the same number and source of exemplars as in standard few-shot learning.

To minimize subjectivity and ensure replicability, the "line-by-line analysis" used in Naive CoT follows a fixed and constrained construction procedure rather than free-form explanation. Specifically, each Naive CoT exemplar is generated by applying the following deterministic rules:

- (1) Sequential Coverage Rule: Each statement or contiguous block of statements in the function body is described in source-code order without reordering or omission.
- (2) Syntactic Description Rule: Each description describes the syntactic functionality (e.g., variable initialization, function calls, loop iteration, data assignment) derived directly from the source code, and tries to identify the abnormal behaviors.
- (3) No Dependency Inference Rule: Data-flow or control-flow relationships are not inferred unless explicitly encoded in the syntax (e.g., loop conditions), and no backward or forward reasoning is performed.
- (4) Under these constraints, Naive CoT serves as a baseline that reflects generic program summarization. This design allows us to evaluate whether generic chain-of-thought structure alone can improve LLM performance.

```
// Vulnerability Detection
Q: Does the following code have exploitable vulnerabilities? Please
check whether the vulnerability, if any, can be practically exploited
to cause real consequences.

{Code to analyze}

A: Step 1: Vulnerability semantics approximation:
The function has an input "hello". The number of offers "hello" can store
is assigned to "num" at line 5. At lines 7-8, The offers from "hello" are
assigned to a stack-based buffer "offered" which is defined at line 6.
Therefore, the vulnerability semantics is:
1 void cycle(struct st *hello)
5   int num = hello->getOfferNum();
6   int offered[max_offered + 1];
7   for (int i = 0; i < num; i++)
8     offered[i] = hello->getOffer(i);
Step 2: Vulnerability semantics reasoning:
{Vulnerability semantics description}
The security-related behavior may cause crashes or exploitable conditions.
Conclusion: the code is vulnerable.
```

Fig. 3. VSP vulnerability-detection exemplar (for the example in Fig. 2).

We do not optimize or iteratively refine prompt phrasing at the lexical level (e.g., specific adjectives or sentence structures), as doing so would introduce uncontrolled variability and confound comparisons across prompting strategies. Instead, we fix prompt wording once per strategy and treat the presence or absence of reasoning structure and semantic guidance as the primary independent variable. This design choice aligns with our goal of evaluating prompting strategies as methodological categories rather than handcrafted prompt instances.

**Zero-Shot (Code) Completion.** Specifically for vulnerability analysis (repair in particular), Pearce et al. [75] propose a prompting strategy that removes or comments out vulnerable code fragments and lets LLMs complete the code. They explored several options and found that the "s.2" version worked most effectively for repairing vulnerabilities in real-world projects, which simply removes the vulnerable code and uses a comment `/*fix [error name] bug*/` to prompt the LLMs for the task. In our study, `[error name]` is the CWE ID of the vulnerability. We refer to this prompting strategy as *zero-shot compl.* and only consider it in Task 3.

```

// Vulnerability Classification
Q: The following code has been confirmed as vulnerable,
which CWE does it have?
{Code to analyze}

A: Step 1: Vulnerability semantics approximation:
..... // same as Task 1
Step 2: Vulnerability semantics reasoning:
{Vulnerability semantics description}
The security-related behavior may cause stack-based buffer
overflow (CWE-121).
Conclusion: the code has a CWE-121 vulnerability.

```

Fig. 4. VSP vulnerability-classification exemplar (for the example in Fig. 2).

```

// Vulnerability Repair
Q: The following code has a CWE-125 out-of-bound read vulnerability at
line 8. Please provide a patch, only showing the code changes needed
rather than the entire patched code, to conform to the conventional
patch format.
{Code to analyze}

A: Step 1: Vulnerability semantics approximation:
..... // same as Task 1
Step 2: Vulnerability semantics reasoning:
{Vulnerability semantics description}
The security-related behavior may cause stack-based buffer overflow
(CWE-121).
Step 3: Repair strategy identification and application:
The vulnerability occurs because the loop does not ensure that `i` is
within the bounds of the allocated array size `max_offered + 1`.
Proper bounds checking should be added to ensure `i` does not exceed
`max_offered + 1`. Thus, the patch is:
- for (int i = 0; i < num; i++)
+ for (int i = 0; i < num && i < max_offered + 1; i++)
..

```

Fig. 5. VSP vulnerability-repair exemplar (for the example in Fig. 2).

on Transformer [88], which is known to exhibit diminished performance when handling extensive text [69], and (3) although LLMs may be good at language understanding tasks, they may not be as good at reasoning complex logic, including code analysis [61]—thus, concentrating on vulnerability semantics aids in curtailing the length of exemplars/prompts/answers and focusing on the core reasoning of vulnerabilities, mitigating potential LLM limitations.

**Vulnerability Semantics.** We define vulnerability semantics of a program as its *code semantics that makes it vulnerable*, which can be approximated by (a) sensitive (i.e., likely vulnerable) statements (e.g., those performing memory operations, numeric computation, and pointer manipulation) and (b) their relevant contextual statements (e.g., those enabling control reachability to the operations/computation). This can be computed with a union of a forward slice and a backward slice. Figure 2 illustrates the vulnerability semantics of a code example. To approximate the semantics, we start with the inputs of the program (i.e., the parameter at line 1). Then, we identify the statements that have (forward) data and control dependencies on the input parameter (lines 5, 7) until a sensitive statement (line 8). Finally, we include line 6 which has backward data dependency of the buffer `offered` used at line 8. Therefore, lines 1 and 5-8 constitute the (approximate) vulnerability semantics.

Note that the vulnerability semantics is either manually identified (for exemplars) or approximated by LLMs during their reasoning (for testing samples), without any pre-extracted data/control-flow information. The manually identified vulnerability semantics in the exemplars is for demonstrating to the LLM the reasoning (i.e., how to identify and focus on vulnerability-relevant behaviors) we expect it to perform. During the testing phase, LLMs are expected to generate their own reasoning and predictions based on the provided code and the patterns learned from the VSP exemplars.

While more complex prompting strategies exist, such as Tree of Thoughts [99], they often involve multiple LLM calls, complex parsing of intermediate outputs, or integration with external tools/agents. Our study specifically focuses on direct prompt engineering to derive actionable insights that are broadly applicable to a wide range of users without requiring significant computational resources or specialized model training expertise. Therefore, we exclude these strategies.

#### 2.4.2 Vulnerability Semantics guided Prompting.

After preliminary evaluation, we observed undesirable performance of LLMs with existing prompting strategies. Thus, we improve Naive CoT into a vulnerability-specific prompting strategy—*Vulnerability-Semantics-guided Prompting (VSP)*. It hones in on *vulnerability semantics*, the pivotal element of effective vulnerability analysis. This improvement is underpinned by three key insights: (1) while a program may encompass numerous lines of code, only a small fraction may be susceptible to vulnerabilities; (2) mainstream LLMs are based

Table 2. Traditional analysis approaches compared

Class	Tool	Year	Technique	Tasks
<i>Code analysis based</i>	CodeQL [3]	2024	static analysis/code querying	1,2
	FlawFinder [2]	2022	static analysis/pattern matching	1,2
	CppCheck [59]	2024	static analysis/symbolic execution	1,2
	ExtractFix [32]	2021	dynamic analysis/symbolic execution	3
	VulnFix [101]	2022	dynamic analysis/inductive inference	3
<i>DL based</i>	VulChecker [60]	2023	graph neural network/training	1,2
	LineVul [28]	2022	Transformer (BERT)/fine-tuning (FT)	1
	VulRepair [29]	2022	Transformer (CodeT5)/pre-training+FT	3
	Devign [105]	2019	(gated) graph neural network/training	1

**VSP.** Guided by vulnerability semantics, we improve Naive CoT to realize VSP. Specifically, each VSP prompt includes two parts. The first part consists of exemplars that provide examples of the reasoning steps for the model to follow CoT. As shown in Figures 3, 4, and 5, each exemplar consists of a question for the task, a code sample, and the answer for the question which includes the reasoning steps. Importantly, these steps describe vulnerability semantics by focusing on code relationships that are relevant to explaining the vulnerable behavior, rather than on general functionality.

Such reasoning steps not only help LLMs focus on the important code fragments for vulnerability analysis but also guide LLMs to focus on the vulnerability behaviors. We provide the same number of exemplars as in standard few-shot learning as discussed earlier—the example code samples are the same. Following these exemplars, the second part of a VSP prompt is the testing sample and the same question as in the exemplars. With the whole VSP prompt as input, LLMs should output the vulnerability analysis results following the vulnerability-semantics-guided reasoning steps in the exemplars. While we provide exemplars with the same CWEs as the testing samples, we note that VSP is not limited to the specific CWEs we evaluate—it can be easily adapted to any CWE with demonstrative examples on the official CWE website [1].

We note that VSP does not enable LLMs to perform sound or complete data/control-flow analysis comparable to traditional program analysis techniques. Instead, VSP is a prompt-level strategy that biases the model’s reasoning focus toward vulnerability-relevant code relationships, as illustrated in the exemplars. The intent is to reduce attention to general functionality and encourage consideration of code semantics that are commonly associated with vulnerability manifestation, without assuming that the resulting reasoning corresponds to precise or exhaustive dependency analysis.

While both Naive CoT and VSP adopt a chain-of-thought structure, they differ fundamentally in their reasoning objectives. Naive CoT enforces a functionality-driven summarization of code behavior, whereas VSP explicitly guides models to identify and reason over vulnerability-relevant semantics. As a result, Naive CoT is intentionally limited to evaluating whether functionality-based reasoning alone is sufficient, while VSP evaluates whether task-specific semantic guidance is necessary for effective vulnerability analysis.

**Zero-Shot VSP.** To assess whether exemplars are essential for improving LLMs’ capabilities, we also consider zero-shot VSP, a VSP variant where we replace the exemplars with a description of VSP steps in the prompt: “(1) approximate the code semantics that make it vulnerable and (2) reason about its underlying code dependencies to analyze vulnerabilities.”

## 2.5 Decoding Strategies and System Prompts

In this study, we investigate how the configuration of the LLMs—specifically decoding strategies and system prompts—impacts their effectiveness in vulnerability analysis.

**Decoding Strategies.** To assess the sensitivity of LLMs to generation parameters, we examine two key decoding dimensions: temperature and Top-P (nucleus sampling). We prioritize these two parameters as they are the most ubiquitous hyperparameters governing the trade-off between output diversity and determinism, directly influencing whether the model adheres to the most probable reasoning paths or explores alternative possibilities. Temperature controls the stochasticity of the model’s output, where 0 ensures deterministic results and 1 allows for maximum diversity. Top-P restricts token generation to the top subset of tokens with a cumulative probability of P. We observe how constraining the probability space impacts the accuracy of vulnerability semantics analysis.

**System Prompts.** We investigate the impact of defining a specific persona for the LLM through system-level instructions. We compare three distinct settings: No System Prompt, which provides the VSP with input directly without behavioral constraints; Helpful Assistant, which applies the generic instruction "You are a helpful assistant"; and Security Expert, which primes the model with the persona "You are an expert in software security and vulnerability analysis." We selected these specific prompts to bracket the potential impact of persona adoption: the "Helpful Assistant" serves as a baseline representing the default alignment behavior of modern LLMs, while the "Security Expert" tests the hypothesis that domain-specific priming activates latent security knowledge and enforces stricter reasoning standards.

## 2.6 Traditional Approaches Compared

To assess where LLMs stand among extant vulnerability analysis solutions, we consider nine traditional approaches for the three tasks, as summarized in Table 2. The year shown is that of the latest release of respective tools accessible by Sep. 2024. The choices are justified by their coverage of (1) the latest SOTA, popular, and/or most effective tools/techniques in respective classes for the tasks they are capable of, (2) both static and dynamic analysis techniques, and (3) various deep learning (DL) models and training schemes. For instance, CodeQL, CppCheck, and FlawFinder are among the best static application security testing (SAST) tools [56] widely used in practice, while VulChecker represents the most recent advance in DL-based vulnerability detection/classification. We used their default or most widely used configuration to test their performance. Specifically, for ExtractFix and VulnFix, we used their default setting that allowed us to reproduce the results reported in their respective original papers. For CppCheck and FlawFinder, we also use their default setting without special arguments/options, which enabled the full range of memory error checking. For CodeQL, we use the security query pack for C/C++ to detect/classify the vulnerabilities. For DL-based models, to ensure a fair comparison and mitigate potential performance degradation due to distribution shifts, we evaluated them in two settings: (1) using their originally provided pre-trained models, and (2) re-training them on the large-scale DiverseVul dataset. This ensures that the DL-based baselines are optimized for the vulnerability analysis tasks on a dataset distribution comparable to what the fine-tuned LLMs experienced, providing a robust "best-case" baseline for traditional DL approaches.

## 2.7 Implementation and Experiment Settings

For Gemini, Claude3, GPT-3.5, and GPT-4, we subscribe to the APIs from respective companies to process the input and generate the results. For Llama2, Falcon, CodeLlama, Qwen2.5-Coder, and DeepSeek-Coder-V2, we download the models from HuggingFace [93], deploy locally, and use their APIs to conduct the experiments. For fine-tuning the Llama2-13b model, we utilized the Unsloth framework [35] to perform efficient instruction tuning. We employed the standard prompting strategy (as described in §2.4.1) to format the input and output for the fine-tuning samples. We perform our experiments on a workstation with an AMD Ryzen Threadripper Pro 5595WX (4.5GHz) CPU with 64 Cores, four Nvidia GeForce RTX A6000 GPUs, and 512GB memory.

<pre> 1 static void esp_do_dma(ESPState *s){ 2   uint32_t len; 3   int to_device; 4   if (!s) 5     return; 6   len = s-&gt;dma_left; 7   if (s-&gt;do_cmd){ 8     trace_esp_do_dma(s-&gt;cmdlen, len); 9     s-&gt;dma_memory_read(s-&gt;dma_opaque, &amp;s-&gt;cmdbuf[s-&gt;cmdlen], len); 10    return; 11  } 12  return; } </pre>
<p><b>Answer with VSP prompt:</b></p> <p>A: Step 1: Find vulnerability semantics: The function has an input "s". The function reads from "s" and writes to "s-&gt;cmdbuf". Therefore, the vulnerability semantics is:</p> <p>Step 2: Vulnerability semantics analysis: At line 6, the value of "s-&gt;dma_left" is assigned to "len". At lines 8-9, the function reads from "s-&gt;dma_memory_read" and writes to "&amp;s-&gt;cmdbuf[s-&gt;cmdlen]". However, the size of "&amp;s-&gt;cmdbuf[s-&gt;cmdlen]" may be less than "len", causing a heap-based buffer overflow (CWE-122). Conclusion: the code has an exploitable vulnerability.</p>
<p><b>Answer with Standard Prompt:</b></p> <p>Based on the provided code, it is difficult to determine if there are any exploitable vulnerabilities or if the inputs can be maliciously used to crash the program. The code appears to be a function that performs a DMA (Direct Memory Access) operation, but without additional context or information about the input validation and usage, it is not possible to definitively determine if there are any vulnerabilities.</p>
<p><b>Answer with Naive CoT Prompt:</b></p> <p>A: 1. The code defines a static function named "esp_do_dma" that takes a pointer to an "ESPState" struct named "s" as a parameter. ..... 11. The function returns. This line is not necessary as it is unreachable code. This code does not have any obvious vulnerability as it is a simple function that performs some operations based on the provided "ESPState" struct. However, without the implementation details of the "ESPState" struct and the "dma_memory_read" function, it is not possible to determine if there are any exploitable vulnerabilities. Conclusion: the code does not have an exploitable vulnerability based on the provided information.</p>

Fig. 6. GPT-3.5 detects a CWE-122 vulnerability with VSP but not with Standard Prompting or naive CoT.

comprehensive comparison: using their originally provided pre-trained models, and using models fine-tuned on the large-scale DiverseVul dataset. This allows us to assess both their off-the-shelf capabilities and their optimized performance when adapted to the latest vulnerability datasets.

### 3 RQ1/RQ2: EFFECTIVENESS OF LLMs

Tables 3, 5, and 6 summarize the effectiveness results of LLMs with existing prompting strategies versus VSP on the three tasks. On each dataset, results of VSP are highlighted in green, with lighter depth for zero-shot VSP.

#### 3.1 Task 1: Vulnerability Detection

We measure the effectiveness of vulnerability detection in terms of recall, precision, and F1, which are standard metrics for binary classification. Additionally, we report the Matthews Correlation Coefficient (MCC) and the Area Under the Precision-Recall Curve (AuPRC). MCC provides a balanced measure of the correlation between predicted and actual classifications, which is particularly useful for imbalanced datasets, while AuPRC evaluates the trade-off between precision and recall across different decision thresholds.

We first look at LLMs' performance with existing prompting strategies. As shown in Table 3, the performance varies widely. While F1 scores range up to 54.55%, the MCC values reveal deeper reliability issues. For instance, with Standard Prompting, Gemini and Llama2-13b exhibit negative MCCs on the Zero-Day dataset (-0.0501 and -0.0588 respectively) and VulChecker (-0.2845 for Llama2-7b), indicating their predictions are often worse than or equivalent to random

To ensure the reproducibility and consistency of our primary evaluations (RQ1, RQ2, and RQ4–RQ7), we utilized a fixed default configuration for all LLMs. Specifically, we set the temperature to 0 to minimize stochasticity and maximize the determinism of the outputs. We set the Top-P (nucleus sampling) to 1, allowing the model to consider the full vocabulary probability distribution, though the generation is effectively constrained by the zero temperature. Furthermore, regarding system instructions, we employed the No System Prompt setting as the default; this means the models received only the task-specific prompts (as detailed in §2.4) without any additional persona definitions or behavioral instructions (e.g., "You are a helpful assistant"). We only deviate from these fixed defaults in RQ3, where we explicitly vary these parameters to assess the sensitivity of LLMs to decoding strategies and system prompts.

For the traditional baseline techniques, we first follow their instructions to reproduce the results reported in their papers. Then, we use the same configurations to evaluate them against our datasets. For DL-based techniques, we evaluate them in two settings to ensure a

Table 3. Vulnerability Detection (Task 1) effectiveness of LLMs with different prompting strategies. SP: Standard Prompting; SF: Standard Few-Shot; NC: Naive CoT; RAG: Retrieval-Augmented Generation; ZVSP: Zero-Shot VSP; VSP: Vulnerability-Semantics-guided Prompting. QC-2.5: Qwen2.5-Coder; DSC-V2: DeepSeek-Code; FT-Llama2: Fine-Tuned Llama2-13b

LLM	PS	VulChecker				BigVul+CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full																												
		Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	Rec	Prec	F1	MCC	
Gemini	SP	36.07%	16.42%	22.56%	0.2008	31.82%	20.19%	24.71%	0.2158	0.2732	16.50%	44.73%	24.11%	-0.5501	0.5149	28.13%	27.11%	27.61%	0.2560	0.2868	60.69%	23.59%	33.98%	0.3526	0.4266	60.69%	23.59%	33.98%	0.3526	0.4266	60.69%	23.59%	33.98%	0.3526	0.4266	60.69%	23.59%	33.98%	0.3526	0.4266	60.69%	23.59%	33.98%	0.3526	0.4266	
	SF	55.42%	25.84%	35.25%	0.2893	0.4261	54.47%	43.31%	48.25%	0.4626	0.4977	28.15%	45.31%	34.73%	-0.6030	0.5469	46.01%	38.15%	41.72%	0.4016	0.4288	54.10%	21.52%	30.79%	0.3183	0.3842	54.10%	21.52%	30.79%	0.3183	0.3842	54.10%	21.52%	30.79%	0.3183	0.3842	54.10%	21.52%	30.79%	0.3183	0.3842	54.10%	21.52%	30.79%	0.3183	0.3842
	NC	9.21%	9.59%	3.99%	0.0077	0.1343	11.01%	25.00%	15.29%	0.1444	0.1972	3.92%	26.67%	6.84%	-0.1314	0.3932	8.05%	20.42%	11.54%	0.1136	0.1559	49.36%	19.72%	28.19%	0.2832	0.3322	49.36%	19.72%	28.19%	0.2832	0.3322	49.36%	19.72%	28.19%	0.2832	0.3322	49.36%	19.72%	28.19%	0.2832	0.3322	49.36%	19.72%	28.19%	0.2832	0.3322
	RAG	27.05%	56.09%	36.51%	0.3511	0.4481	25.88%	16.42%	20.09%	0.1660	0.2258	22.33%	57.50%	32.17%	0.0736	0.5933	25.09%	43.34%	31.78%	0.3161	0.3532	11.65%	46.15%	18.60%	0.2222	0.3008	11.65%	46.15%	18.60%	0.2222	0.3008	11.65%	46.15%	18.60%	0.2222	0.3008	11.65%	46.15%	18.60%	0.2222	0.3008	11.65%	46.15%	18.60%	0.2222	0.3008
	ZVSP	81.93%	10.88%	19.21%	0.1002	0.4721	81.03%	15.38%	25.85%	0.3023	0.4857	32.03%	54.09%	40.24%	0.0531	0.6005	65.00%	26.78%	37.93%	0.3938	0.4641	52.82%	18.49%	27.40%	0.2822	0.3629	52.82%	18.49%	27.40%	0.2822	0.3629	52.82%	18.49%	27.40%	0.2822	0.3629	52.82%	18.49%	27.40%	0.2822	0.3629	52.82%	18.49%	27.40%	0.2822	0.3629
Claude3	SP	80.00%	26.12%	39.38%	0.3705	0.5395	74.19%	40.05%	52.62%	0.5214	0.5762	33.33%	54.55%	41.88%	0.0604	0.6061	62.51%	40.24%	48.96%	0.4649	0.5193	63.75%	31.00%	40.72%	0.4327	0.4786	63.75%	31.00%	40.72%	0.4327	0.4786	63.75%	31.00%	40.72%	0.4327	0.4786	63.75%	31.00%	40.72%	0.4327	0.4786	63.75%	31.00%	40.72%	0.4327	0.4786
	SF	71.43%	14.15%	23.62%	0.1672	0.4406	63.63%	20.00%	30.43%	0.3136	0.4252	37.36%	50.65%	43.33%	0.0100	0.5979	57.64%	28.27%	37.93%	0.3511	0.4358	17.13%	19.31%	18.18%	0.1611	0.1935	17.13%	19.31%	18.18%	0.1611	0.1935	17.13%	19.31%	18.18%	0.1611	0.1935	17.13%	19.31%	18.18%	0.1611	0.1935					
	NC	53.01%	7.43%	13.04%	-0.0666	0.3230	33.18%	8.25%	13.21%	0.0979	0.2200	29.41%	56.60%	38.71%	0.0782	0.6065	38.53%	24.09%	29.65%	0.2808	0.3222	14.11%	16.67%	15.28%	0.1321	0.1653	14.11%	16.67%	15.28%	0.1321	0.1653	14.11%	16.67%	15.28%	0.1321	0.1653	14.11%	16.67%	15.28%	0.1321	0.1653					
	RAG	8.93%	12.82%	10.53%	0.0356	0.1492	10.62%	60.00%	18.05%	0.2417	0.3703	24.21%	52.27%	33.09%	0.0249	0.5719	14.59%	41.70%	21.61%	0.2351	0.2940	14.72%	20.87%	17.27%	0.1565	0.1893	14.72%	20.87%	17.27%	0.1565	0.1893	14.72%	20.87%	17.27%	0.1565	0.1893	14.72%	20.87%	17.27%	0.1565	0.1893					
	ZVSP	10.64%	35.71%	16.39%	0.1555	0.2714	21.33%	58.50%	31.26%	0.3389	0.4143	3.64%	33.33%	6.56%	-0.0801	0.4258	11.87%	45.25%	18.56%	0.2142	0.2849	30.67%	25.00%	27.55%	0.2550	0.2876	30.67%	25.00%	27.55%	0.2550	0.2876	30.67%	25.00%	27.55%	0.2550	0.2876	30.67%	25.00%	27.55%	0.2550	0.2876					
GPT-3.5	SP	57.57%	45.24%	50.67%	0.4563	0.5329	51.28%	76.95%	61.54%	0.6164	0.6504	11.76%	46.15%	18.75%	-0.0294	0.5102	48.24%	25.38%	33.26%	0.3258	0.3757	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165					
	SF	4.71%	4.44%	4.57%	-0.0502	0.0880	2.14%	80.00%	4.6%	0.1271	0.4295	12.91%	60.00%	5.56%	0.0315	0.5573	3.25%	48.15%	6.09%	0.2100	0.2713	3.11%	6.55%	4.27%	0.0288	0.0495	3.11%	6.55%	4.27%	0.0288	0.0495	3.11%	6.55%	4.27%	0.0288	0.0495	3.11%	6.55%	4.27%	0.0288	0.0495					
	NC	32.94%	84.85%	47.45%	0.5047	0.6187	16.90%	95.18%	28.70%	0.3938	0.5764	4.85%	55.56%	8.93%	0.0238	0.5399	18.23%	78.53%	29.59%	0.3717	0.4559	8.70%	16.87%	11.48%	0.1041	0.1400	8.70%	16.87%	11.48%	0.1041	0.1400	8.70%	16.87%	11.48%	0.1041	0.1400	8.70%	16.87%	11.48%	0.1041	0.1400					
	RAG	23.53%	28.17%	25.64%	0.1920	0.2924	17.29%	48.20%	25.45%	0.2728	0.3434	23.81%	83.33%	37.04%	0.0721	0.7262	12.54%	53.23%	30.67%	0.3719	0.4955	13.04%	36.84%	19.27%	0.2610	0.3123	13.04%	36.84%	19.27%	0.2610	0.3123	13.04%	36.84%	19.27%	0.2610	0.3123	13.04%	36.84%	19.27%	0.2610	0.3123					
	ZVSP	47.05%	15.00%	37.50%	0.3623	0.5549	16.00%	54.55%	24.74%	0.2816	0.3689	27.18%	57.14%	36.84%	0.0798	0.6037	42.97%	45.23%	44.07%	0.4260	0.4494	32.65%	22.61%	26.72%	0.2478	0.2853	32.65%	22.61%	26.72%	0.2478	0.2853	32.65%	22.61%	26.72%	0.2478	0.2853										
GPT-4	SP	57.57%	45.24%	50.67%	0.4563	0.5329	51.28%	76.95%	61.54%	0.6164	0.6504	11.76%	46.15%	18.75%	-0.0294	0.5102	48.24%	25.38%	33.26%	0.3258	0.3757	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165	13.50%	27.50%	18.11%	0.1775	0.2165					
	SF	6.87%	16.67%	9.52%	0.0526	0.1581	6.81%	25.00%	10.71%	0.1132	0.1770	26.31%	65.22%	37.50%	0.0530	0.6419	13.26%	35.33%	19.33%	0.2949	0.2573	19.02%	19.25%	17.14%	0.1694	0.2021	19.02%	19.25%	17.14%	0.1694	0.2021	19.02%	19.25%	17.14%	0.1694	0.2021	19.02%	19.25%	17.14%	0.1694	0.2021					
	NC	23.33%	63.63%	34.14%	0.3531	0.4688	11.36%	83.33%	20.00%	0.3001	0.4905	21.05%	83.33%	33.61%	0.1194	0.5719	18.58%	76.76%	29.92%	0.3707	0.4887	16.59%	86.50%	27.84%	0.3731	0.5265	16.59%	86.50%	27.84%	0.3731	0.5265	16.59%	86.50%	27.84%	0.3731	0.5265	16.59%	86.50%	27.84%	0.3731	0.5265					
	RAG	60.00%	50.00%	54.55%	0.4993	0.5677	40.91%	60.00%	48.64%	0.4792	0.5159	34.95%	63.16%	45.00%	0.0628	0.6532	45.29%	57.72%	50.75%	0.4996	0.5231	7.36%	17.65%	10.39%	0.0987	0.1374	7.36%	17.65%	10.39%	0.0987	0.1374	7.36%	17.65%	10.39%	0.0987	0.1374	7.36%	17.65%	10.39%	0.0987	0.1374					
	ZVSP	36.67%	5.16%	9.05%	-0.1706	0.2372	43.18%	9.13%	15.79%	0.1295	0.2725	32.25%	44.99%	37.50%	-0.0814	0.5536	37.37%	19.63%	25.74%	0.2435	0.2942	31.14%	16.36%	21.45%	0.1966	0.2466	31.14%	16.36%	21.45%	0.1966	0.2466	31.14%	16.36%	21.45%	0.1966	0.2466	31.14%	16.36%	21.45%	0.1966	0.2466					
Llama2-7b	SP	70.00%	56.75%	62.68%	0.5903	0.6471	56.81%	49.01%	52.63%	0.5073	0.5374	44.36%	54.09%	48.70%	0.0822	0.6564	60.39%	49.95%	54.68%	0.5357	0.5576	39.13%	32.47%	35.49%	0.3372	0.3661	39.13%	32.47%	35.49%	0.3372	0.3661	39.13%	32.47%	35.49%	0.3372	0.3661	39.13%	32.47%	35.49%	0.3372	0.3661					
	SF	16.67%	2.42%	4.22%	-0.2945	0.1324	14.87%	6.15%	11.58%	0.0879	0.2776	43.14%	46.67%	46.31%	0.0000	0.6079	33.56%	19.85%	23.33%	0.2857	0.3574	19.06%	10.85%	13.83%	0.1127	0.1603	19.06%	10.85%	13.83%	0.1127	0.1603	19.06%	10.85%	13.83%	0.1127	0.1603	19.06%	10.85%	13.83%	0.1127	0.1603					
	NC	28.13%	11.53%	3.63%	0.0491	0.2302	25.33%	10.05%	14.39%	0.1056	0.1913	22.54%	63.89%	33.33%	0.1286	0.6258	25.33%	28.49%	28.82%	0.2498	0.2801	16.54%	7.31%	10.14%	0.0731	0.1033	16.54%	7.31%	10.14%	0.0731	0.1033	16.54%	7.31%	10.14%	0.0731	0.1033	16.54%	7.31%	10.14%	0.0731	0.1033					
	RAG	12.50%	7.11%	8.98%	-0.0284	0.1364	10.00%	6.25%	7.69%	0.0319	0.0986	45.49%	48.42%	46.70%	-0.0295	0.6048	22.53%	20.56%	21.50%	0.1928	0.2629	23.98%	14.06%	17.72%	0.1544	0.2003	23.98%	14.06%	17.72%	0.1544	0.2003	23.98%	14.06%	17.72%	0.1544	0.2003	23.98%	14.06%	17.72%	0.1544	0.2003					
	ZVSP	31.25%	4.41%	7.41%	-0.2042	0.2088	53.13%	8.13%	14.11%	0.1289	0.3153	20.																																		

Table 4. Statistical significance and size of differences in effectiveness between different prompting strategies. SP: Standard Prompting; SF: Standard Few-Shot; NC: Naive CoT; ZSC: Zero-Shot Completion; RAG: Retrieval-Augmented Generation; ZVSP: Zero-Shot VSP; VSP: Vulnerability-Semantics-guided Prompting. —: Not applicable; ZSC cannot be applied to detection or classification tasks

Compared Strategies	Detection						Classification						Repair	
	p-value			Effect Size			p-value			Effect Size			Correct	Correct
	F1	MCC	AuPRC	F1	MCC	AuPRC	micF1	MCC	micAu	micF1	MCC	micAu		
SP:SF	0.3414	<b>0.0158</b>	<b>0.0971</b>	-0.1104	-0.1775	-0.1379	0.1556	<b>0.0385</b>	0.9900	-0.0651	-0.0823	-0.0413	<b>0.0059</b>	-0.2124
SP:NC	0.3745	0.5352	0.3269	0.0413	-0.0542	0.0188	<b>0.0045</b>	<b>0.0023</b>	0.0641	-0.1540	-0.1825	-0.1554	<b>0.0003</b>	-0.2336
SP:ZSC	-	-	-	-	-	-	-	-	-	-	-	-	<b>0.0306</b>	0.0680
SP:RAG	0.6034	0.3459	0.7566	-0.0334	-0.0972	-0.0083	0.9554	0.4438	0.2212	-0.0298	0.0294	0.0628	<b>0.0009</b>	-0.2136
SP:ZVSP	0.8397	0.3745	0.3524	-0.0311	-0.0486	-0.0760	<b>0.0002</b>	<b>0.0298</b>	0.2611	-0.1002	-0.0767	-0.0952	<b>0.0037</b>	-0.1556
SP:VSP	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.6413</b>	<b>-0.5577</b>	<b>-0.4856</b>	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.4436</b>	<b>-0.4655</b>	<b>-0.4387</b>	<b>0.0000</b>	<b>-0.4628</b>
SF:NC	0.1496	0.1381	0.0703	0.1094	0.1160	0.1458	0.0859	<b>0.0458</b>	<b>0.0081</b>	-0.0850	-0.1243	-0.1481	0.1577	-0.0704
SF:ZSC	-	-	-	-	-	-	-	-	-	-	-	-	<b>0.0003</b>	0.3060
SF:RAG	0.3836	0.1114	<b>0.0125</b>	0.0618	0.0797	0.1207	0.1941	<b>0.0282</b>	0.1999	0.0288	0.1127	0.1104	0.7881	0.0220
SF:ZVSP	0.4868	0.1828	0.6329	0.0721	0.1369	0.0668	0.4559	0.8406	0.1720	-0.0284	0.0000	-0.0740	0.5932	0.0124
SF:VSP	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.5587</b>	-0.4050	-0.3256	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	-0.3904	-0.4165	-0.4093	<b>0.0000</b>	-0.3252
NC:ZSC	-	-	-	-	-	-	-	-	-	-	-	-	<b>0.0000</b>	0.3372
NC:RAG	0.3438	0.9466	0.5463	-0.0638	-0.0486	-0.0393	<b>0.0104</b>	<b>0.0003</b>	<b>0.0026</b>	0.1084	0.2288	0.2483	0.2005	0.0648
NC:ZVSP	0.5243	0.9399	0.1643	-0.1071	-0.0043	-0.1306	0.5297	0.2598	0.4433	0.0588	0.1051	0.0400	<b>0.0174</b>	0.0952
NC:VSP	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.6066</b>	<b>-0.5015</b>	<b>-0.4744</b>	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	-0.3150	-0.3236	-0.2820	<b>0.0000</b>	-0.2496
RAG:ZSC	-	-	-	-	-	-	-	-	-	-	-	-	<b>0.0004</b>	-0.2888
RAG:ZVSP	0.5575	0.7061	0.2687	0.0281	0.0631	-0.0565	<b>0.0370</b>	<b>0.0146</b>	<b>0.0071</b>	-0.0598	-0.1293	-0.1719	<b>0.0001</b>	0.0280
RAG:VSP	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.5722</b>	<b>-0.4486</b>	<b>-0.4555</b>	0.0000	<b>0.0000</b>	<b>0.0000</b>	-0.4026	<b>-0.5131</b>	<b>-0.4952</b>	<b>0.0000</b>	-0.3180
ZVSP:ZSC	-	-	-	-	-	-	-	-	-	-	-	-	0.3121	-0.2600
ZVSP:VSP	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	<b>-0.6605</b>	<b>-0.5213</b>	<b>-0.4592</b>	<b>0.0000</b>	<b>0.0000</b>	<b>0.0000</b>	-0.3577	-0.3997	-0.3345	<b>0.0000</b>	-0.3516
ZSC:VSP	-	-	-	-	-	-	-	-	-	-	-	-	<b>0.0000</b>	<b>-0.5712</b>

34.48% (VulChecker), 40.56% (BigVul+CVEFixes), 47.06% (Zero-Day), 42.44% (PrimeVul-Subset), and 20.23% (PrimeVul-Full), compared to Llama2-7b’s 16.36%, 22.95%, 46.70%, 26.82%, and 17.72% respectively. Similarly, GPT-4 consistently outperforms GPT-3.5. This trend holds across all datasets.

We also notice that the commercial LLMs (i.e., Gemini, Claude3, GPT-3.5, GPT-4) outperform the open-source LLMs (i.e., Llama2-7b, Llama2-13b, Falcon, CodeLlama, Qwen2.5-Coder, DeepSeek-Coder-V2), where the commercial LLMs achieve 23.62%-54.55% F1 while the open-source LLMs achieve 7.69%-47.06% F1 against the four datasets. This indicates the advantages of commercial LLMs for vulnerability analysis.

When examining the performance of the code-specific LLMs (CodeLlama, Qwen2.5-Coder, and DeepSeek-Coder-V2) with existing prompting strategies, these models generally exhibit lower F1 scores compared to the leading general-purpose commercial LLMs like GPT-4 and Claude3. For instance, CodeLlama’s best F1 across all datasets is 11.13% (RAG on PrimeVul-Subset), Qwen2.5-Coder’s is 33.71% (Standard Prompting on BigVul+CVEFixes), and DeepSeek-Coder-V2’s is 11.56% (Standard Prompting on BigVul+CVEFixes). While their performance varies across datasets and prompting strategies, their peak F1 scores remain considerably below the 54.55% achieved by GPT-4. This suggests that despite their code-centric training, these code-specific LLMs are not yet as effective for complex software vulnerability detection tasks as the more powerful, larger general-purpose LLMs. A potential reason for this could be that while code-specific LLMs are optimized for tasks like code generation or simple bug fixing, their training objectives might not sufficiently emphasize the deep semantic understanding of code that is crucial for identifying subtle security vulnerabilities. General-purpose LLMs, with their broader training on diverse text and code, might develop a more robust understanding of logic and abstract reasoning that translates more effectively to complex vulnerability analysis, even without explicit code-focused pre-training.

The limitations of existing prompting strategies are further highlighted by the MCC and AuPRC metrics. A significant number of models exhibit negative MCC scores, particularly on the Zero-Day dataset, indicating a correlation worse than random guessing. For instance, on the Zero-Day dataset with standard prompting, Gemini scores -0.0501 MCC, DeepSeek-Coder-V2 scores -0.7360, and CodeLlama drops to -0.8944. Even the high-performing GPT-4 only achieves a

marginal 0.0530 MCC on the Zero-Day dataset with standard prompting. The AuPRC scores mirror this instability; while some models like GPT-4 maintain a relatively high AuPRC (0.6419 on Zero-Day with standard prompting), smaller models like Falcon struggle (0.5641 AuPRC). These metrics reveal that without specific guidance, LLMs often fail to discriminate reliably between vulnerable and non-vulnerable samples, leading to poor predictive stability.

```

1 static String HHVM_FUNCTION(bcsb, const String &left,
2 const String &right, int64_t scale) {
3     if (scale < 0) scale = BCG(bc_precision);
4     bc_num first, second, result;
5     bc_init_num(&first);
6     bc_init_num(&second);
7     bc_init_num(&result);
8     php_str2num(&first, (char *)left.data());
9     php_str2num(&second, (char *)right.data());
10    bc_sub(first, second, &result, scale);
11    if(result->n_scale>scale) result->n_scale = scale;
12    String ret(bc_num2str(result), AttachString);
13    bc_free_num(&first);
14    bc_free_num(&second);
15    bc_free_num(&result);
16    return ret; }

```

// Gemini output with VSP  
Step 1: Find vulnerability semantics:  
The function has String inputs "left" and "right". At lines 9-10, "left" and "right" are converted into numbers "first" and "second" which are declared at line 6. At line 11, "first" and "second" are subtracted into "result" with scale "scale". Therefore, the vulnerability semantics is:  
1 static String HHVM\_FUNCTION(bcsb, const String &left, const String &right, int64\_t scale)  
9 php\_str2num(&first, (char \*)left.data());  
10 php\_str2num(&second, (char \*)right.data());  
11 bc\_sub(first, second, &result, scale);  
Step 2: Vulnerability semantics analysis:  
At line 11, "first" and "second" are subtracted into "result" without checking. This may cause an integer overflow (CWE-190) vulnerability.  
Conclusion: the code has a CWE-190 vulnerability.

// Gemini output with naive CoT  
A: 1. The code defines a static function named "HHVM\_FUNCTION(bcsb)" that takes three parameters: a constant reference to an object of type "String" named "left", a constant reference to an object of type "String" named "right", and an int64\_t variable named "scale".  
2. The function begins.  
17. It calls the "bc\_free\_num" function, passing the address of the "result" object. This line is potentially vulnerable to a CWE-416 vulnerability if "result" was already freed.  
18. The function returns the "ret" variable.  
19. The function ends.  
Conclusion: The code has a CWE-416 vulnerability if any of the "bc\_free\_num" calls are performed on a previously freed object.

// Llama2-7b output with VSP  
A: Step 1: Find vulnerability semantics:  
The function has inputs "left", "right", and "scale". The function performs a subtraction operation on two bc\_num objects, which may cause a stack-based buffer overflow (CWE-121) vulnerability if the input values are not properly validated.  
Step 2: Vulnerability semantics analysis:  
At line 9, the "php\_str2num" function is called with the input "left" and "right". The function may cause a buffer overflow if the input values are not properly validated.  
Conclusion: The code has a CWE-121 vulnerability.

Fig. 7. An example where Gemini correctly classifies a CWE-190 vulnerability with VSP while it fails with naive CoT, while Llama2-7b fails even with VSP.

we also evaluate the performance of the fine-tuned Llama2-13b model. With standard prompting, the fine-tuned model generally outperforms its base (i.e., pre-trained) version and other open-source models on the existing datasets. For instance, on the VulChecker dataset, it achieves an F1 score of 34.13%, significantly surpassing the base Llama2-13b (14.11%) and even exceeding GPT-4 with standard prompting (9.52%). This indicates that fine-tuning effectively adapts the model to the task format and distribution of the training data. However, on the Zero-Day dataset, the fine-tuned model with standard prompting (46.30%) does not show a significant advantage over the base model (47.06%), suggesting that fine-tuning on historical data does not automatically guarantee better generalization to unseen vulnerabilities. Furthermore, we observe that the choice of prompting strategy remains crucial for the fine-tuned model. Advanced strategies consistently boost performance over standard prompting: Standard Few-Shot and RAG improve the F1 score on VulChecker to 44.32% and 46.81%, respectively, while Naive CoT reaches 55.76%.

Among the four existing prompting strategies, standard few-shot achieves the best performance on many cases (e.g., Gemini, GPT-3.5, Llama2-7b, and Llama2-13b achieve the best F1 on the either VulChecker or BigVul+CVEFixes datasets with standard few-shot, as marked red). Yet naive CoT does not work well on the majority of these LLMs. It only achieves the best F1 for GPT-4 against VulChecker, BigVul+CVEFixes, and PrimeVul-Subset, Llama2-7b and CodeLlama against Zero-Day, as well as Falcon against BigVul+CVEFixes. As Figure 6 shows, with naive CoT, GPT-3.5 just simply scans the code and analyzes the major functionality. It is even worse than standard prompting on Gemini (6.84%-15.29% versus 22.56%-27.61% F1 against the four datasets) and Claude3 (10.53%-33.09% versus 23.62%-43.33% F1 against the four datasets). RAG achieves the best effectiveness for several LLMs against some datasets (e.g., Gemini against VulChecker, Claude3 against BigVul+CVEFixes, GPT-3.5 against Zero-Day and PrimeVul-Subset), indicating that RAG has the potential to mitigate the insufficient context and improve the few-shot learning.

Across different datasets, the effectiveness does not consistently correlate to the LLMs' performance. For instance, seven models (Gemini, Claude3, Llama2-7b, Llama2-13b, CodeLlama, Qwen2.5-Coder, and DeepSeek-Coder-V2) detect vulnerabilities in BigVul+CVEFixes more accurately than in the VulChecker dataset (up to 8.94%-48.25% versus 7.69%-36.51% F1 among different prompting strategies). The other three show the opposite trend (up to 37.93%-54.55% versus 21.82%-48.64% F1 against VulChecker and BigVul+CVEFixes respectively). Notably, performance on existing datasets

(VulChecker, BigVul+CVFixes, PrimeVul-Subset) may be subject to data leakage, as evidenced by the performance drops on the Zero-Day dataset in many cases. (e.g., from 48.25% to 34.73%, 47.45% to 36.84%, 54.55% to 46.07% for Gemini, GPT-3.5, and GPT-4 respectively). The PrimeVul-Subset dataset also presents varied performance, with F1 scores ranging from 3.33% to 50.75%. Its effectiveness is not consistently superior or inferior to the other three datasets. For example, the best F1 of Gemini on PrimeVul-Subset (41.72%) is higher than the ones on VulChecker (36.51%) and Zero-Day (34.73%) but lower than the one on BigVul+CVFixes (48.25%). This indicates that LLMs can achieve comparable effectiveness on datasets with higher label accuracy.

When evaluated on the PrimeVul-Full dataset, which includes a much wider variety of 62 CWEs compared to the 5 targeted in other datasets, we observe a general decline in performance for most models using existing strategies. For example, Claude3 drops from 37.93% to 18.18% with standard prompting. This indicates that while LLMs may be reasonably effective at detecting common high-profile vulnerabilities (like buffer overflows), they struggle significantly when tasked with identifying a broader spectrum of security flaws without specific guidance. However, Gemini remains relatively robust, achieving the highest F1 of 33.98% with Standard Prompting on this diverse dataset.

**Finding 1 (RQ1):** In vulnerability detection, large general-purpose LLMs outperform specialized code models. Fine-tuning aids historical analysis but fails to improve Zero-Day generalization. Naive CoT lags behind standard methods, while RAG shows promise. Performance drops on Zero-Day data—suggesting possible leakage in older datasets—and on diverse sets like PrimeVul-Full, highlighting current struggles with varied vulnerability types.

Now let us look at LLMs' performance with **VSP strategies**. As shown in Table 3, VSP improves the F1 scores for some of the LLMs in certain evaluated settings. However, the extent of this improvement is variable; while gains are observed on existing datasets, they are often marginal on the unbiased Zero-Day dataset for some models. As Figure 6 shows, VSP helps address some limitations of existing prompting strategies and detects vulnerabilities more effectively in certain cases. For example, with VSP, GPT-3.5 focuses on vulnerability semantics and notices that the write of `&s->cmdbuf[s->cmdlen]` is unchecked and may have a buffer overflow issue. It pushes their F1 up to 62.68% (on GPT-4), 61.54% (on Claude3), 48.70% (on GPT-4), and 54.68% (on GPT-4)—from 54.55% (on GPT-4), 48.64% (on GPT-4), 47.06% (on Llama2-13b), and 50.75% (on GPT-4)—for the four datasets, respectively. However, the improvement on the Zero-Day dataset, the dataset least likely to suffer from contamination, is relatively modest (1.64 percentage points, or 3.48% relative increase), indicating that VSP may not provide a significant benefit for vulnerability detection. Among individual LLMs, VSP achieves performance improvement by up to 25.62 points (or 111.6%) in F1 on Llama2-7b against the BigVul+CVFixes dataset, compared to its best performance with existing strategies. On Claude3, the improvements are 27.05 and 28.37 points (or 114.5% and 85.52%) in F1 for the VulChecker and BigVul+CVFixes datasets.

VSP also yields improvements in predictive reliability as measured by MCC and AuPRC for certain models. For example, on the Zero-Day dataset, VSP boosts Claude3's MCC from 0.0782 (Standard Few-Shot) to 0.2198, and improves its AuPRC to 0.6812, reinforcing that VSP helps models distinguish the positive class more effectively. On the challenging PrimeVul-Full dataset, VSP enables Gemini to achieve the highest MCC of 0.4237 and AuPRC of 0.4786, demonstrating that the vulnerability semantics focused guidance provided by VSP translates into more robust and correlated predictions across diverse vulnerability types.

Overall, VSP improves less on Gemini (up to 7.24 points against the PrimeVul-Subset dataset) and GPTs (up to 16.64% against BigVul+CVFixes for GPT-3.5) than the other LLMs. In comparison, zero-shot VSP (i.e., VSP without exemplars) does not help as much. It underperforms the best existing strategies on most of the LLM (e.g., GPT-4 achieves 9.05%-37.50% F1 with zero-shot VSP, while it achieves 46.07%-54.55% F1 with RAG and Naive CoT, against the four

datasets), even often worse than standard prompting (e.g., Llama2-13b achieves 10.24%-25.71% F1 with zero-shot VSP, while with standard prompting, it achieves 14.11%-47.06% F1 against the four datasets). This indicates that the exemplars in VSP are essential for improving the LLMs for vulnerability detection.

```

1 static xmlChar *
2 xmlParseAttValueComplex(xmlParserCtxtPtr ctxt, int *attlen, int normalize){
3     xmlChar limit = 0;
4     xmlChar *buf = NULL;
5     size_t len = 0;
6     size_t buf_size = 0;
7     int c, i, in_space = 0;
8     .....
28    buf_size = XML_PARSER_BUFFER_SIZE;
29    buf = (xmlChar *) xmlMallocAtomic(buf_size);
30    .....
186   if ((in_space) && (normalize)) {
187       while (buf[len - 1] == 0x20) len--;
188   }
190   buf[len] = 0;

```

**Answer with VSP:**  
Step 1: Find vulnerability semantics:  
The function has input "len" and "buf". "len" is used to access the "buf" at line 187. Therefore, the vulnerability semantics is:  
187 while (buf[len - 1] == 0x20) len--;

Step 2: Vulnerability semantics analysis:  
At line 187, "buf" is accessed with "len". However, it is possible that "len" is less than or equal to 0, causing an out-of-bound read (CWE-125) vulnerability.  
Step 3: Patching strategy:  
To avoid the out-of-bound read (CWE-125) vulnerability at line 187, we should ensure that the "len" is greater than 0, which can be done by adding a condition at line 187. Therefore, the patch should be:  
Patch:  
- while (buf[len - 1] == 0x20) len--;  
+ while (len > 0 && buf[len - 1] == 0x20) len--;

**Answer with Standard Prompt:**  
Patch:  
- if ((in\_space) && (normalize)) {  
- while (buf[len - 1] == 0x20) len--;  
- }  
+ if ((in\_space) && (normalize) && len > 0) {  
+ while (buf[len - 1] == 0x20 && len > 0) len--;  
+ };

**Answer with Naive CoT Prompt:**  
Step 1: Analyze the code line by line:  
1. This line declares a static function named "xmlParseAttValueComplex" that takes in three parameters: a pointer to a structure of type "xmlParserCtxtPtr" named "ctxt", a pointer to an integer "attlen", and an integer "normalize". The function returns a pointer to an "xmlChar".  
2. The function's body begins on this line.  
3. This line declares an "xmlChar" variable named "limit" and initializes it to 0.  
.....  
Step 2: Vulnerability semantics analysis:  
At line 187, the "buf" array is accessed at index "len - 1". However, it is possible that "len" is 0, causing an out-of-bound read (CWE-125) vulnerability.  
Step 3: Patching strategy:  
To avoid the out-of-bound read (CWE-125) vulnerability at line 187, we should add a condition to check if "len" is greater than 0 before accessing the "buf" array. If "len" is 0, we can skip the loop and proceed to the next line. Therefore, the patch should be:  
Patch:  
- while (buf[len - 1] == 0x20) len--;  
+ if (len > 0) {  
+ while (buf[len - 1] == 0x20) len--;  
+ };

Fig. 8. GPT-3.5 fixed a CWE-125 vulnerability with VSP but not with Standard Prompting or naive CoT.

To validate these improvements statistically, we employed Wilcoxon signed-rank tests [89] to compute the  $p$ -value at the 0.95 confidence level, considering differences with  $p \leq 0.05$  as significant. We used Cliff's Delta [19] to measure effect size, interpreting the strength as negligible if  $|d| \leq 0.147$ , small if  $0.147 < |d| \leq 0.33$ , medium if  $0.33 < |d| \leq 0.474$ , and large if  $|d| > 0.474$  [78]. As shown in Table 4, the performance differences observed with VSP are statistically confirmed ( $p < 0.001$ ) against existing prompting strategies. Specifically, in the aggregate analysis, VSP demonstrates a large effect size when compared to Standard Prompting ( $d = -0.6413$  for F1) and Naive CoT ( $d = -0.6066$  for F1). Even against RAG, VSP maintains a large effect size on F1 scores ( $d = -0.5722$ ) and a medium effect size on MCC and AuPRC ( $d = -0.4486$  and  $d = -0.4555$ ), confirming that the vulnerability-semantics guidance offers a statistical advantage over general-purpose prompting strategies for detection.

Furthermore, when applying VSP to the fine-tuned model, we also observe performance gains, achieving F1 scores of 60.38%, 43.67%, 48.37%, and 46.43% across the four datasets. Notably, the base Llama2-13b model with VSP achieves an F1 score of 48.48% on the Zero-Day dataset, which is comparable to, and slightly higher than, the fine-tuned model with either standard prompting (46.30%) or VSP (48.37%). This finding is pivotal: it demonstrates that our proposed VSP strategy applied to an off-the-shelf open-source model can achieve performance parity with, or even surpass, a more resource-intensive fine-tuning approach on unseen vulnerabilities. On the PrimeVul-Full dataset, VSP demonstrates strong generalization capabilities. Despite the increased difficulty of detecting 62 different CWEs, VSP still raise performance in some cases. For instance, GPT-4 improves from 28.26% (RAG) to 35.49% (VSP). Notably, Gemini achieves the highest performance on this diverse dataset with an F1 of 41.72% using VSP. This suggests that the vulnerability semantics reasoning approach used in VSP is not limited to specific memory-related errors but helps LLMs reason more effectively about a wide variety of security flaws.

Table 5. Vulnerability Classification (Task 2) effectiveness of LLMs with different prompting strategies. SP: Standard Prompting; SF: Standard Few-Shot; NC: Naive CoT; RAG: Retrieval-Augmented Generation; ZVSP: Zero-Shot VSP; VSP: Vulnerability-Semantics-guided Prompting. QC-2.5: Qwen2.5-Coder; DSC-V2: DeepSeek-Coder-V2; FT-Llama2: Fine-Tuned Llama2-13b. MacF1: Macro F1, MicF1: Micro F1, MacAu: Macro AuPRC; MicAu: Micro AuPRC

LLM	PS	VulChecker Dataset					BigVul-CVEfixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full				
		MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu
Gemini	SP	21.1%	24.3%	0.150	0.161	0.499	29.8%	29.9%	0.251	0.404	0.570	16.6%	18.1%	0.076	0.287	0.468	24.8%	27.1%	0.167	0.352	0.465	1.5%	8.2%	0.194	0.709	0.445
	SF	26.3%	28.6%	0.160	0.321	0.452	28.3%	26.9%	0.168	0.473	0.543	14.9%	20.1%	0.075	0.298	0.458	30.1%	33.7%	0.219	0.347	0.459	1.7%	10.5%	0.160	0.689	0.420
	NC	26.2%	27.9%	0.162	0.350	0.461	26.4%	32.6%	0.237	0.447	0.593	12.0%	16.3%	0.074	0.275	0.442	27.9%	34.1%	0.215	0.343	0.457	1.7%	10.2%	0.191	0.707	0.441
	RAG	28.1%	28.7%	0.185	0.373	0.452	20.6%	26.9%	0.166	0.290	0.475	12.0%	18.0%	0.074	0.280	0.430	18.3%	26.9%	0.146	0.304	0.458	1.8%	4.3%	0.047	0.177	0.104
	ZVSP	22.7%	24.0%	0.157	0.307	0.475	31.5%	31.1%	0.252	0.449	0.580	13.6%	17.4%	0.072	0.278	0.474	26.0%	29.6%	0.163	0.349	0.450	2.7%	16.3%	0.170	0.695	0.448
	VSP	28.2%	31.4%	0.201	0.397	0.525	39.4%	47.2%	0.354	0.452	0.593	21.3%	27.8%	0.192	0.296	0.510	28.4%	35.2%	0.228	0.351	0.460	4.2%	23.2%	0.174	0.136	0.454
Claude3	SP	21.0%	25.3%	0.170	0.305	0.454	23.6%	26.6%	0.170	0.305	0.454	13.6%	25.4%	0.075	0.195	0.352	25.3%	30.2%	0.145	0.308	0.407	0.9%	7.1%	0.023	0.190	0.706
	SF	26.0%	28.4%	0.159	0.340	0.498	29.8%	25.5%	0.159	0.340	0.498	12.9%	20.4%	0.084	0.212	0.401	23.8%	29.6%	0.159	0.340	0.498	1.1%	8.7%	0.095	0.055	0.405
	NC	21.8%	27.1%	0.194	0.344	0.537	21.6%	24.5%	0.163	0.324	0.449	12.0%	23.7%	0.087	0.226	0.439	23.8%	34.7%	0.167	0.349	0.470	3.0%	13.0%	0.107	0.028	0.404
	RAG	28.9%	31.9%	0.188	0.352	0.457	23.5%	25.9%	0.170	0.306	0.452	12.9%	24.4%	0.075	0.194	0.361	24.8%	32.7%	0.148	0.294	0.398	0.4%	1.7%	0.055	0.065	0.049
	ZVSP	28.2%	31.8%	0.203	0.391	0.525	28.3%	31.1%	0.203	0.421	0.452	15.2%	25.7%	0.129	0.261	0.487	22.7%	33.9%	0.146	0.301	0.407	0.9%	6.9%	0.027	0.026	0.041
	VSP	29.9%	32.6%	0.220	0.450	0.559	31.4%	36.9%	0.274	0.438	0.453	37.1%	40.1%	0.345	0.467	0.610	25.9%	34.8%	0.242	0.350	0.492	3.4%	14.5%	0.109	0.067	0.416
GPT3.5	SP	24.6%	25.3%	0.123	0.203	0.474	23.2%	26.5%	0.125	0.203	0.474	9.0%	19.6%	0.035	0.146	0.380	21.9%	27.7%	0.203	0.307	0.510	1.5%	4.2%	0.029	0.168	0.071
	SF	29.9%	27.0%	0.178	0.408	0.477	23.2%	25.9%	0.183	0.358	0.488	12.1%	15.2%	0.067	0.238	0.376	20.7%	23.6%	0.117	0.313	0.419	0.6%	5.1%	0.025	0.024	0.034
	NC	32.0%	36.2%	0.244	0.398	0.506	22.3%	25.9%	0.126	0.378	0.487	8.5%	14.2%	0.047	0.299	0.373	38.3%	31.0%	0.202	0.317	0.509	4.8%	16.5%	0.104	0.061	0.416
	RAG	29.9%	32.2%	0.235	0.378	0.502	25.8%	24.7%	0.137	0.359	0.441	12.2%	19.2%	0.069	0.237	0.360	27.0%	28.5%	0.157	0.339	0.441	1.0%	6.7%	0.025	0.028	0.072
	ZVSP	30.1%	31.8%	0.216	0.374	0.493	31.3%	32.5%	0.220	0.456	0.525	15.3%	24.4%	0.127	0.210	0.509	22.2%	28.8%	0.160	0.329	0.547	1.4%	8.9%	0.169	0.094	0.445
	VSP	35.6%	37.2%	0.470	0.410	0.575	27.4%	28.0%	0.149	0.423	0.496	14.2%	19.2%	0.046	0.245	0.391	30.3%	39.1%	0.257	0.393	0.574	3.9%	16.9%	0.104	0.062	0.412
GPT4	SP	31.7%	32.4%	0.252	0.511	0.525	26.9%	34.4%	0.223	0.464	0.579	11.2%	21.9%	0.189	0.150	0.517	18.9%	22.7%	0.070	0.293	0.388	1.1%	5.6%	0.022	0.023	0.073
	SF	38.2%	41.1%	0.312	0.497	0.558	30.3%	34.7%	0.224	0.460	0.481	11.4%	20.4%	0.179	0.149	0.519	21.3%	27.2%	0.254	0.328	0.503	0.7%	7.2%	0.026	0.020	0.046
	NC	35.2%	35.7%	0.292	0.442	0.521	28.4%	31.0%	0.221	0.426	0.587	14.8%	22.7%	0.181	0.203	0.486	30.3%	38.9%	0.250	0.391	0.569	5.0%	18.7%	0.129	0.070	0.445
	RAG	29.9%	36.3%	0.228	0.352	0.466	27.8%	29.8%	0.146	0.417	0.448	12.2%	19.2%	0.067	0.243	0.379	27.0%	28.5%	0.146	0.317	0.418	1.7%	5.7%	0.020	0.030	0.080
	ZVSP	39.1%	39.8%	0.316	0.504	0.549	31.3%	32.5%	0.220	0.456	0.525	15.3%	24.4%	0.127	0.210	0.509	22.2%	28.8%	0.160	0.329	0.547	1.4%	8.9%	0.169	0.094	0.445
	VSP	44.4%	42.1%	0.313	0.517	0.572	33.0%	34.9%	0.227	0.478	0.575	35.9%	39.4%	0.278	0.412	0.595	30.7%	39.5%	0.265	0.406	0.585	5.9%	20.9%	0.160	0.139	0.457
Llama2-7b	SP	15.0%	15.6%	0.043	0.294	0.320	13.4%	14.2%	0.043	0.294	0.320	4.5%	5.0%	0.016	0.202	0.365	9.9%	9.9%	-0.058	0.194	0.379	0.3%	1.8%	0.060	0.011	0.529
	SF	19.0%	4.3%	0.039	0.293	0.375	12.6%	14.0%	0.059	0.219	0.430	6.7%	8.3%	-0.014	0.161	0.263	20.7%	24.4%	0.097	0.264	0.358	0.6%	2.9%	0.027	0.176	0.685
	NC	18.6%	18.5%	0.063	0.273	0.408	19.9%	21.3%	0.074	0.296	0.416	12.1%	17.2%	0.074	0.267	0.452	25.5%	28.3%	0.178	0.365	0.429	0.3%	1.6%	0.066	0.078	0.584
	RAG	17.6%	19.5%	0.029	0.297	0.342	14.4%	15.0%	0.046	0.298	0.425	3.2%	5.8%	0.017	0.215	0.367	22.8%	26.4%	0.269	0.329	0.496	0.3%	1.2%	0.060	0.071	0.532
	ZVSP	19.4%	19.9%	0.067	0.283	0.374	19.8%	20.9%	0.067	0.283	0.374	4.0%	5.6%	0.016	0.204	0.368	15.1%	15.7%	0.044	0.249	0.423	0.3%	1.2%	0.066	0.071	0.529
	VSP	20.5%	20.5%	0.076	0.298	0.392	22.2%	22.6%	0.110	0.382	0.440	13.2%	18.6%	0.075	0.309	0.462	26.8%	30.7%	0.231	0.442	0.580	1.0%	4.3%	0.054	0.181	0.104
Llama2-13b	SP	29.6%	31.4%	0.222	0.448	0.595	25.6%	26.6%	0.148	0.316	0.412	11.0%	19.7%	0.170	0.182	0.452	8.0%	12.7%	0.036	0.256	0.362	0.3%	2.9%	0.024	0.182	0.705
	SF	22.7%	20.3%	0.266	0.286	0.370	23.1%	22.3%	0.104	0.290	0.476	13.9%	15.9%	0.040	0.242	0.484	22.9%	26.3%	0.120	0.324	0.435	0.7%	3.6%	0.029	0.091	0.742
	NC	23.9%	24.0%	0.162	0.429	0.612	25.7%	26.8%	0.181	0.359	0.473	7.5%	11.0%	-0.023	0.184	0.348	22.6%	26.8%	0.220	0.343	0.539	0.8%	3.1%	0.028	0.209	0.813
	RAG	17.6%	21.5%	0.057	0.266	0.402	13.0%	16.0%	0.063	0.269	0.387	8.1%	13.3%	0.036	0.218	0.357	23.9%	29.3%	0.141	0.301	0.390	0.9%	1.4%	0.055	0.064	0.482
	ZVSP	29.2%	31.1%	0.249	0.449	0.606	27.1%	27.9%	0.159	0.472	0.501	9.1%	17.3%	0.060	0.276	0.400	19.5%	21.5%	0.040	0.278	0.356	0.3%	1.4%	0.057	0.087	0.594
	VSP	32.4%	38.1%	0.292	0.458	0.628	31.1%	34.9%	0.257	0.403	0.519	27.1%	32.0%	0.259	0.452	0.593	27.7%	31.4%	0.269	0.452	0.593	1.0%	4.7%	0.027	0.020	0.818
Flan	SP	10.1%	15.5%	0.047	0.239	0.383	7.1%	14.9%	0.047	0.239	0.383	9.5%	12.8%	0.047	0.239	0.383	12.4%	8.6%	0.029	0.297	0.304	0.0%	0.0%	0.000	0.035	0.165
	SF	13.3%	19.4%	0.071	0.282	0.444	10.8%	15.0%	0.071	0.265	0.410	3.4%	4.4%	0.062	0.042	0.097	9.2%	14.1%	0.031	0.245	0.390	1.3%	1.6%	0.064	0.076	0.566
	NC	12.0%	14.6%	0.178	0.375	0.566	12.9%	18.1%	0.078	0.282	0.460	7.4%	9.2%	-0.029	0.192	0.312	13.8%	14.6%	0.037	0.239	0.393	0.9%	1.0%	0.066	0.077	0.581
	RAG	20.1%	20.0%	0.074	0.311	0.413	19.8%	23.1%	0.167	0.324	0.469	6.2%	8.9%	-0.020	0.206	0.314	11.0%	12.1%	0.033	0.248	0.394	0.8%	0.9%	0.066	0.078	0.585
	ZVSP	11.9%	18.0%	0.053	0.249	0.397	8.7%	13.7%	0.026	0.228	0.366	11.3%	19.4%	0.026	0.228	0.366	10.4%	11.5%	0.026	0.228	0.366	0.5%	0.6%	0.056	0.067	0.498
	VSP	30.2%	28.5%	0.249	0.489	0.650	27.2%	24.8%	0.165	0.474	0.564	18.3%	19.4%	0.074	0.300	0.463	10.7%	15.0%	0.039	0.248	0.402	1.3%	1.6%	0.062	0.073	0.549
CodLlama	SP	16.8%	18.3																							

the F1-score from these global counts, effectively giving more weight to larger classes [8]. In addition to F1 scores, we report the Matthews Correlation Coefficient (MCC) and both Macro and Micro Area Under the Precision-Recall Curve (AuPRC). MCC offers a robust measure of the quality of multiclass classifications, while AuPRC evaluates the models' ability to rank correct CWEs higher across different thresholds. Table 5 summarizes these metrics across all the CWEs. As micro F1 focuses on the global performance of the models, we mainly discuss micro F1 in the table.

```

1 void ice_debugfs_module_write(struct file *filp,
2 const char __user *buf, size_t count, loff_t *ppos){
3 struct ice_pf *pf = file_inode(filp)->i_private;
4 struct dentry *dentry = file_dentry(filp);
5 struct device *dev = ice_pf_to_dev(pf);
6 char user_val[16], *cmd_buf;
7 int module, log_level, cnt;
.....
// array user_val is safely used without
overflowing the size 16.
zeroday_0229_detect/CWE125__CVE-2024-36935__linux
_vul.c:7: [2] (buffer) char: Statically-sized arrays can
be improperly restricted, leading to potential overflows
or other issues (CWE-119/CWE-120). Perform bounds
checking, use functions that limit length, or ensure that
the size is larger than the maximum possible length.

```

Fig. 9. An example where traditional static analyzer FlawFinder does not conduct comprehensive checks on the whole program but directly reports the potential vulnerability based on its predefined patterns, causing the false positive.

the best micro F1 (22.75%) against the Zero-Day dataset with Naive CoT.

Overall, the differences between the three existing datasets, VulChecker, BigVul+CVEFixes, and PrimeVul-Subset, are not substantial (e.g., Llama2-13b achieves up to 31.48%, 26.83% and 29.33% micro F1 against the VulChecker, BigVul+CVEFixes, and PrimeVul-Subset datasets, which are similar). However, we consistently notice performance drops against the Zero-Day dataset for most LLMs (e.g., GPT-4 achieves only up to 22.75% micro F1 against the Zero-Day dataset, while it achieves 31.92%-41.11% micro F1 against the three other datasets). These drops suggest data contamination in the non-zero-day datasets, where the Zero-Day dataset is most important for evaluating LLMs' classification effectiveness.

We also notice the challenge of classifying vulnerabilities across a wide range of types on the PrimeVul-Full dataset. With 62 different CWEs to distinguish between, most models struggle significantly compared to the 5-CWE classification task in other datasets. For instance, GPT-4's Micro F1 drops from 41.92% on PrimeVul-Subset (using Naive CoT) to just 18.76% on PrimeVul-Full. This sharp decline underscores that while LLMs can differentiate between a small set of distinct vulnerability types, their ability to accurately classify a broad spectrum of security flaws remains limited without specialized guidance.

Besides, we have similar observations to those on Task 1. For instance, between models from the same vendors, larger/newer ones again perform better (e.g., Llama2-13b achieved 19.73%-31.48% micro F1 while Llama2-7b only achieves 17.72%-28.37% micro F1). Yet generally the performance is not correlated consistently with model size, age, and capacity across vendors (e.g., GPT-3.5 achieved up to 36.26% micro F1 against the VulChecker dataset, while Claude3, the newer LLM achieved only up to 34.78% micro F1). Code-specific LLMs (CodeLlama, Qwen2.5-Coder, and DeepSeek-Coder-V2) generally exhibit lower micro F1 scores compared to the leading general-purpose commercial LLMs like GPT-4 and Claude3 (e.g., DeepSeek-Coder-V2 achieves up to 25.44% micro F1 while GPT-4 achieves 34.44% against BigVul+CVEFixes).

With **existing prompting strategies**, the LLMs again see largely varying performance, from 4.39% (Llama-7b against VulChecker) to 41.11% (GPT-4 against VulChecker) in micro F1, across models of different vendors. GPT-4 again achieves the best performance against the VulChecker and BigVul+CVEFixes, with 41.11% and 34.44% micro F1. Among the four strategies, RAG performs the best on the VulChecker dataset, where four (Gemini, Claude3, Llama2-7b, Falcon) of the ten LLMs achieve the best micro F1 with RAG (28.74%, 31.97%, 19.51%, and 20.00% respectively), indicating that RAG mitigated insufficient context and improved the quality of few-shot learning. On the other datasets, naive CoT and standard prompting perform better in some cases. For example, GPT-4 achieves the best micro F1 (34.44%) against BigVul+CVEFixes with standard prompting, while it achieves

We also evaluate the capabilities of the fine-tuned Llama2-13b model. Overall, fine-tuning yields a substantial performance boost over the base Llama2-13b model across all datasets. For instance, on the Zero-Day dataset, the micro F1 score with standard prompting increases from 19.73% to 31.71%. When comparing different prompting strategies, we observe that the fine-tuned model is remarkably robust; the performance differences between Standard Prompting, Standard Few-Shot, Naive CoT, and RAG are minimal (e.g., ranging only from 31.41% to 31.90% on the Zero-Day dataset, and 32.47% to 33.72% on BigVul+CVEFixes). This indicates that the model has effectively internalized the classification task definition during training, reducing its reliance on in-context exemplars or retrieval mechanisms found in strategies like Few-Shot or RAG.

The MCC and AuPRC results for existing prompting strategies reveal significant instability in classification reliability, often more severe than F1 scores suggest. On the Zero-Day dataset, MCC values are frequently close to zero, indicating near-random classification capabilities for many models. For instance, Llama2-7b achieves a negligible MCC of 0.0116 with Standard Prompting, while even the sophisticated GPT-4 only reaches 0.1849. Furthermore, on the PrimeVul-Subset dataset, Llama2-7b even exhibits a negative MCC (-0.0058), implying a negative correlation between predictions and ground truth. The Micro AuPRC scores mirror this struggle; on the PrimeVul-Full dataset, which requires distinguishing among 62 classes, most models collapse, with Claude3 achieving a Micro AuPRC of just 0.0706 and Falcon reaching 0.1065. This highlights that without careful task-specific guidance, LLMs struggle to form distinct decision boundaries between vulnerability types.

**Finding 3 (RQ1):** For classification, with existing prompting strategies, the LLMs achieve up to 41.11% micro F1 (on GPT-4). Fine-tuning yields improvements for Llama2-13b and makes it robust to variations in prompting strategies. Data contamination in the non-zero-day datasets still exists, indicating the importance of using zero-day data for evaluation. Yet neither a consistently winning strategy, nor a consistent correlation between LLMs' size/age/max #tokens and their performance, is observed. The significant performance drop on PrimeVul-Full highlights the difficulty of broad-spectrum vulnerability classification.

With **VSP strategies**, the results in Table 5 reveal improvements brought by VSP for some of the LLMs and datasets studied. For example, with VSP, the classification performance is pushed to 47.27% micro F1 with Gemini, against BigVul+CVEFixes, where the best number from existing prompting strategies is 34.44% with GPT-4 (Standard Prompting). Against the Zero-Day dataset, the classification performance is pushed from 25.40% (Claude3 with standard prompting) to 40.16% (Claude3), indicating that VSP helps improve the effectiveness of vulnerability classification. As Figure 7 shows, Gemini correctly classifies a CWE-190 vulnerability with VSP while it fails with naive CoT: without focusing on vulnerability semantics, the LLMs may be easily distracted by the long source code contents and ignore the core part for vulnerability classification. However, Llama2-7b still fails on this case. This indicates that the model's capability still matters (in terms of size, as shown in Table 1): the model cannot reason about the vulnerability semantics correctly, causing the failure.

On the PrimeVul-Full dataset, absolute scores of VSP are lower due to the 62-class problem, VSP however achieves better effectiveness in some cases. For example, Gemini achieves a Micro F1 of 27.14% with VSP, compared to 10.56% with Standard Few-Shot. GPT-4 improves from 18.76% (Naive CoT) to 20.95% (VSP). This indicates that the structured reasoning provided by VSP helps models navigate the larger decision space of diverse CWEs more effectively than unguided prompting.

Furthermore, while the fine-tuned model is generally stable across existing strategies, VSP still manages to push the performance boundary further. On the BigVul+CVEFixes dataset, the fine-tuned model with VSP achieves a micro

F1 of 35.53%, surpassing all other strategies. Similarly, on the Zero-Day dataset, VSP achieves the highest score of 32.02%. This demonstrates that even for a model well-tuned to the task, guiding the reasoning process via vulnerability semantics provides a tangible benefit over standard fine-tuning approaches.

```

1 void exif_data_load_data (ExifData *data, const unsigned char *d_orig, unsigned int ds){
2 unsigned int l;
3 ExifLong offset;
4 ExifShort n;
5 const unsigned char *d = d_orig;
6 unsigned int len, fullds;
7
8 .....
9
10 while (ds >= 3) {
11     while (ds && (d[0] == 0xff)) {
12         d++;
13         ds--;
14     }
15     if (ds && d[0] == JPEG_MARKER_SOI) {
16         d++;
17         ds--;
18         continue;
19     }
20     if (ds && d[0] == JPEG_MARKER_APP1)
21         break;
22     if (ds >= 3 && d[0] >= 0xe0 && d[0] <= 0xef) {
23         d++;
24         ds--;
25         l = {d[0] << 8} | d[1]; // Access of d has potential CWE-121 vulnerability
26         if (l > ds)
27             return;
28         d += l;
29         ds -= l;
30         continue;
31     }
32     .....
33 }

```

A: Step 1: Find vulnerability semantics:  
The function has inputs "data", "d\_orig", and "ds". At line 10, the inputs are checked whether they are NULL or 0. At line 13, the size of the EXIF data is printed. At lines 20-23, if the size of the EXIF data is less than 6, an error message is printed and the function returns. At lines 24-27, if the EXIF header is not found at the start of the data, the function searches for the EXIF marker.

Therefore, the vulnerability semantics is:  
1 exif\_data\_load\_data (ExifData \*data, const unsigned char \*d\_orig, unsigned int ds)  
..... // Listing all the lines for the code  
Conclusion: the code does not have any exploitable vulnerabilities.

Fig. 10. GPT-3.5 with VSP outputs vague interpretation of code semantics on a vulnerability detection sample.

maintain clearer decision boundaries even when the classification space is larger and more complex.

In 30 out of 40 cases, zero-shot VSP did not improve the LLM over the existing strategies, indicating that the exemplars in VSP were essential for improving vulnerability classification.

These observations are further supported by the statistical analysis in Table 4, which indicates that VSP achieves statistically distinguishable results ( $p < 0.001$ ) compared to existing prompting strategies for classification. The effect sizes are predominantly medium to large; for instance, compared to Standard Prompting, VSP shows a medium effect size on Micro F1 ( $d = -0.4436$ ) and Micro AuPRC ( $d = -0.4387$ ). Against RAG, VSP exhibits a large effect size on both MCC ( $d = -0.5131$ ) and Micro AuPRC ( $d = -0.4952$ ). This suggests that for the multi-class classification task, the structured reasoning in VSP provides a robust advantage over retrieval-based or simple few-shot approaches.

**Finding 4 (RQ2):** VSP improves the highest vulnerability classification accuracy to 47.27% (on Gemini) in micro F1. VSP raises classification performance metrics for some of the evaluated LLMs, demonstrating improvements against the Zero-Day and the PrimeVul-Full datasets compared to standard prompting, though the magnitude is smaller for some cases. VSP also enhances the fine-tuned model. However, the improvement is much less if without exemplars (i.e., Zero-Shot VSP).

### 3.3 Task 3: Vulnerability Repair

For the ExtractFix dataset, we apply the LLM-generated patches to the vulnerable functions and use the test cases to validate if the vulnerabilities are removed and the programs are still functional. If the program passes the test cases

VSP demonstrates a marked ability to improve the correlation and ranking quality of classification predictions as evidenced by MCC and AuPRC. On the Zero-Day dataset, VSP improves Claude3's MCC from 0.0887 (Naive CoT) to 0.3345, and its Micro AuPRC from 0.4349 to 0.6160. This indicates that VSP not only helps the model select the correct CWE more often but also improves the calibration of its confidence scores. Furthermore, on the more challenging PrimeVul-Full dataset, VSP raises the Micro AuPRC for Gemini to 0.4544 (compared to 0.4455 with Standard Prompting) and GPT-4 to 0.4587, suggesting that the vulnerability semantics focused guidance provided by VSP helps

Table 6. Vulnerability Repair (Task 3) effectiveness of LLMs with different prompting strategies

LLM	PS	ExtractFix Dataset					BigVul+CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full					
		Valid	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct	
Gemini	SP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.42%	0.42%	0.00%	0.83%	0.00%	0.00%	0.00%	0.00%	1.89%	0.00%	1.89%	0.00%	1.84%	3.68%	3.52%		
	SF	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	0.00%	2.12%	2.91%	3.88%	3.88%	9.68%	1.89%	1.89%	5.66%	9.43%	0.84%	3.73%	3.83%	8.40%	
	NC	29.17%	16.67%	4.17%	8.33%	0.00%	29.17%	5.93%	5.08%	1.69%	12.71%	2.91%	5.83%	1.94%	10.68%	1.89%	5.66%	9.43%	16.98%	1.84%	4.90%	4.47%	11.21%	
	ZSC	33.33%	4.17%	0.00%	0.00%	0.00%	29.16%	1.27%	0.85%	0.42%	2.54%	2.91%	0.00%	3.88%	6.80%	3.77%	5.66%	5.66%	15.09%	0.00%	3.07%	5.78%	8.84%	
	RAG	25.00%	4.17%	12.50%	8.33%	0.00%	25.00%	0.42%	5.51%	2.12%	8.06%	0.97%	5.83%	3.88%	10.68%	3.77%	3.77%	5.66%	13.21%	0.21%	3.16%	3.56%	6.93%	
	ZVSP	0.00%	0.00%	0.00%	4.17%	0.00%	4.17%	4.66%	2.12%	1.69%	8.47%	0.97%	2.91%	2.91%	6.80%	1.89%	3.77%	7.55%	13.21%	0.63%	4.84%	6.14%	11.61%	
Claude	VSP	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	1.27%	13.14%	2.91%	5.83%	6.80%	15.53%	3.77%	3.77%	11.32%	18.87%	0.42%	3.94%	7.90%	12.26%	
	SP	29.17%	8.33%	12.50%	8.33%	0.00%	29.17%	7.63%	8.47%	3.81%	19.91%	4.95%	9.71%	0.97%	15.53%	3.77%	3.77%	7.55%	15.09%	0.00%	4.90%	4.99%	9.90%	
	SF	54.17%	12.50%	20.83%	20.83%	0.00%	54.16%	8.47%	5.08%	7.20%	20.76%	5.83%	7.76%	1.94%	15.53%	3.77%	5.66%	5.66%	15.09%	0.42%	5.06%	6.63%	12.12%	
	NC	50.00%	12.50%	16.67%	20.83%	0.00%	50.00%	7.20%	3.81%	2.12%	13.14%	5.83%	8.74%	3.88%	18.45%	1.89%	5.66%	7.55%	15.09%	0.00%	4.90%	4.99%	9.90%	
	ZSC	29.16%	0.00%	4.17%	4.17%	20.83%	8.33%	2.97%	1.27%	2.54%	6.78%	8.74%	0.00%	5.83%	14.56%	1.89%	3.77%	3.77%	9.43%	0.63%	2.04%	5.64%	8.31%	
	RAG	50.00%	8.33%	20.83%	20.83%	0.00%	50.00%	0.42%	8.47%	7.63%	16.52%	5.83%	7.77%	5.83%	19.43%	1.89%	5.66%	9.43%	16.98%	1.84%	4.90%	3.42%	10.16%	
GPT-3.5	ZVSP	33.33%	12.50%	8.33%	12.50%	0.00%	33.33%	8.90%	4.24%	1.69%	14.83%	6.80%	7.76%	0.00%	14.56%	1.89%	5.66%	7.55%	15.09%	0.84%	6.26%	5.88%	12.98%	
	VSP	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.32%	6.36%	22.88%	11.65%	10.80%	8.74%	31.07%	3.77%	5.66%	9.43%	18.87%	3.68%	3.88%	6.04%	13.40%	
	SP	20.84%	4.17%	12.50%	4.17%	0.00%	20.84%	6.78%	6.78%	6.36%	19.92%	5.83%	2.91%	8.74%	17.48%	3.77%	9.43%	9.43%	22.64%	0.21%	9.07%	10.76%	20.04%	
	SF	50.00%	16.67%	20.83%	12.50%	0.00%	50.00%	8.47%	7.64%	3.81%	19.92%	2.91%	7.76%	2.91%	13.59%	3.77%	7.55%	9.43%	20.75%	0.21%	3.88%	3.23%	11.81%	
	NC	41.67%	4.17%	25.00%	12.50%	0.00%	41.67%	8.05%	6.78%	3.39%	19.92%	4.95%	3.88%	8.74%	16.57%	7.55%	5.66%	15.09%	28.30%	0.21%	5.06%	9.07%	14.35%	
	ZSC	19.16%	0.00%	12.50%	4.17%	25.00%	16.67%	3.39%	0.85%	3.39%	7.62%	4.85%	0.00%	0.97%	5.83%	3.77%	3.77%	11.34%	16.98%	0.42%	2.53%	14.14%	17.09%	
GPT-4	RAG	29.16%	8.33%	16.67%	4.17%	0.00%	29.16%	3.39%	8.92%	10.47%	22.88%	1.94%	5.83%	10.68%	18.45%	5.66%	9.43%	7.55%	22.64%	0.00%	1.90%	2.53%	4.43%	
	ZVSP	37.50%	8.33%	12.50%	16.67%	0.00%	37.50%	6.78%	6.36%	6.36%	19.50%	5.83%	2.91%	5.83%	14.56%	5.66%	9.43%	7.55%	22.64%	0.21%	4.85%	10.97%	16.03%	
	VSP	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.75%	8.47%	12.29%	30.71%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	16.98%	32.08%	0.22%	15.38%	13.88%	29.06%	
	SP	54.17%	16.67%	20.83%	16.67%	0.00%	54.17%	8.06%	9.19%	8.06%	25.31%	8.74%	4.86%	8.93%	19.43%	5.66%	5.66%	7.55%	18.87%	0.00%	6.33%	14.35%	20.66%	
	SF	54.17%	8.33%	37.50%	8.33%	0.00%	54.17%	9.30%	10.47%	10.47%	30.24%	5.83%	4.86%	5.83%	16.52%	5.66%	7.55%	7.55%	20.75%	0.21%	4.22%	8.23%	12.66%	
	NC	62.50%	20.83%	29.17%	12.50%	0.00%	62.50%	11.62%	2.32%	10.47%	24.42%	5.83%	1.94%	7.76%	15.53%	7.55%	7.55%	16.98%	32.08%	0.00%	5.91%	14.35%	20.25%	
ZSC	50.00%	4.16%	12.50%	4.16%	16.67%	20.83%	3.37%	3.37%	4.49%	11.24%	4.85%	0.07%	4.85%	10.68%	3.77%	3.77%	5.66%	13.20%	0.21%	5.06%	11.60%	16.88%		
Llama-2-7b	RAG	70.83%	16.67%	20.83%	33.33%	0.00%	70.83%	5.45%	10.47%	10.47%	26.39%	4.85%	3.90%	11.65%	20.40%	3.77%	18.87%	12.11%	33.60%	0.00%	4.43%	5.06%	9.49%	
	ZVSP	58.33%	16.67%	37.50%	4.17%	0.00%	58.33%	11.62%	9.30%	6.97%	27.89%	7.76%	2.91%	4.85%	15.52%	5.66%	9.43%	7.55%	22.64%	0.21%	8.95%	17.40%	26.50%	
	VSP	83.33%	25.00%	41.67%	16.67%	0.00%	83.33%	17.02%	12.77%	19.15%	48.73%	16.50%	6.80%	20.39%	43.69%	7.55%	9.43%	20.75%	37.74%	1.84%	7.36%	22.07%	31.26%	
	SP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.97%	0.00%	0.00%	3.77%	3.77%	0.00%	0.00%	3.68%	3.68%	
	SF	8.33%	0.00%	4.17%	4.17%	0.00%	8.33%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	
	NC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.97%	0.97%	1.89%	1.89%	1.89%	5.66%	0.00%	0.00%	4.43%	4.43%	
ZSC	16.67%	0.00%	0.00%	4.16%	12.50%	4.17%	1.27%	0.00%	0.00%	1.27%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.89%	1.89%	0.00%	0.00%	4.85%	4.85%		
Llama-2-3b	RAG	8.33%	0.00%	0.00%	8.33%	0.00%	8.33%	0.00%	0.42%	0.00%	0.42%	0.00%	0.97%	0.00%	0.97%	0.00%	1.89%	1.89%	3.77%	0.00%	0.00%	1.84%	1.84%	
	ZVSP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.89%	1.89%	3.77%	0.00%	1.84%	1.84%	3.68%	
	VSP	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%	
	SP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.27%	0.00%	0.00%	1.27%	0.00%	0.00%	1.94%	1.94%	0.00%	1.89%	1.89%	3.77%	0.00%	1.48%	2.88%	4.36%	
	SF	8.33%	4.17%	0.00%	0.00%	4.17%	1.69%	0.85%	0.00%	2.54%	0.00%	0.00%	1.94%	1.94%	0.00%	0.00%	0.00%	3.77%	3.77%	0.00%	1.74%	2.13%	3.87%	
	NC	8.33%	0.00%	4.17%	4.17%	0.00%	8.33%	0.42%	0.85%	0.42%	1.69%	0.00%	0.00%	0.97%	0.97%	1.89%	1.89%	3.77%	5.66%	0.00%	1.69%	2.08%	3.97%	
ZSC	20.83%	4.17%	0.00%	4.17%	12.50%	4.17%	0.00%	0.85%	0.42%	1.27%	0.97%	0.00%	0.00%	0.97%	0.00%	0.00%	1.89%	1.89%	0.21%	2.11%	3.94%	6.26%		
CodeLlama	RAG	8.33%	0.00%	0.00%	8.33%	0.00%	8.33%	0.00%	0.85%	0.42%	1.27%	0.00%	0.00%	1.94%	1.94%	0.00%	1.89%	1.89%	3.77%	0.00%	0.84%	1.33%	2.17%	
	ZVSP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.27%	0.00%	0.00%	1.27%	0.00%	0.00%	0.00%	0.00%	0.00%	1.89%	1.89%	3.77%	0.00%	0.84%	3.02%	3.87%	
	VSP	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.63%	7.90%	
	SP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.42%	0.42%	0.42%	1.27%	0.00%	0.00%	1.94%	1.94%	0.00%	0.00%	1.89%	1.89%	0.00%	0.00%	1.31%	1.31%	
	SF	8.33%	0.00%	4.17%	4.17%	0.00%	8.33%	0.00%	0.42%	1.27%	1.69%	0.00%	0.97%	0.97%	1.94%	0.00%	1.89%	1.89%	3.77%	0.00%	1.84%	1.05%	2.89%	
	NC	8.33%	0.00%	4.17%	4.17%	0.00%	8.33%	0.42%	0.85%	1.27%	2.12%	0.00%	0.00%	0.97%	0.97%	1.89%	1.89%	3.77%	5.66%	0.00%	1.69%	2.08%	3.77%	
ZSC	16.67%	0.00%	0.00%	4.17%	12.50%	4.17%	0.42%	0.42%	0.85%	1.69%	0.00%	0.97%	0.00%	0.97%	0.00%	0.00%	1.89%	1.89%	0.00%	0.63%	1.36%	1.99%		
Qwen-72b	RAG	4.17%	0.00%	0.00%	4.17%	0.00%	4.17%	0.00%	0.42%	0.85%	1.27%	0.00%	0.00%	1.94%	1.94%	0.00%	1.89%	1.89%	3.77%	0.00%	0.84%	1.33%	2.17%	
	ZVSP	8.33%	4.17%	0.00%	4.17%	0.00%	8.33%	0.42%	0.42%	1.27%	2.12%	0.00%	0.97%	0.97%	1.94%	0.00%	0.00%	1.89%	1.89%	0.63%	1.05%	1.21%	2.89%	
	VSP	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.69%	2.32%</		

effectiveness in terms of *correct rate*=#correct patches/#testing samples. Falcon failed in all the settings on this task. Thus, we omit it in Table 6.

As Table 6 shows, with **existing prompting strategies**, the two Llama2 models barely work, only generating up to <10% (0.97–8.33%) correct patches against the four datasets. Using the existing strategies, patches from the four commercial models (Gemini, Claude3, and both GPTs) achieve up to 29.17–70.83% correct rates on the ExtractFix dataset, although much lower (10.68–33.96%) on the other datasets. Against the BigVul+CVEFixes, GPT-4 outperforms Claude3 (30.24% versus 20.76%). The largest LLM Gemini (with 50T parameters, as shown in Table 1) achieves a much lower best correct rate (10.68–29.17%) than the other three commercial LLMs (16.98–54.17% on Claude3, 18.45–50.00% on GPT-3.5, and 20.40–70.83% on GPT-4). Both GPT and Llama2 have their newer/larger versions performing better than the smaller ones. The code-specific LLMs (CodeLlama, Qwen2.5-Coder, and DeepSeek-Coder-V2) generally exhibit significantly lower correct rates compared to the leading general-purpose commercial LLMs, with their best performance being 12.50% on ExtractFix (Qwen2.5-Coder), and much lower on other datasets (e.g., 1.94–5.66% on BigVul+CVEFixes, Zero-Day, and PrimeVul-Subset). Via other metrics, it appears that all the LLMs can generate semantically equivalent and plausible patches besides the syntactic equivalent ones. The PrimeVul-Full dataset, with its 62 diverse CWEs, presents a significant challenge for vulnerability repair. Most models struggle to generate correct patches for this broad range of vulnerabilities. For example, GPT-4’s correct rate drops from 33.96% on PrimeVul -Subset (using RAG) to just 20.68% (using Standard Prompting) on PrimeVul-Full. This indicates that while LLMs can repair common vulnerabilities like buffer overflows, their ability to fix a wide variety of less common security flaws is limited. However, GPT-3.5 shows relative resilience, achieving a correct rate of 20.04% with standard prompting on this diverse dataset.

We also evaluate the performance of the fine-tuned Llama2-13b model with existing prompting strategies. With standard prompting, the fine-tuned model demonstrates a clear improvement over the base Llama2-13b model across all datasets. For example, on the BigVul+CVEFixes dataset, the correct rate increases from 1.27% (base) to 26.69% (fine-tuned). Similarly, on PrimeVul-Subset, the correct rate jumps from 3.77% to 15.09%. This substantial gain indicates that fine-tuning is highly effective for the complex task of vulnerability repair, enabling the model to learn the specific patterns and syntactic requirements of generating valid patches. Furthermore, the fine-tuned model shows a high degree of stability across different existing strategies (Standard Prompting, Standard Few-Shot, Naive CoT, and RAG), with correct rates often remaining within a close range (e.g., 25.85% to 27.12% on BigVul+CVEFixes).

Among the five strategies, standard prompting has the worst performance on the two Llama2s and Gemini (mostly near 0% correct rate), while zero-shot-completion performs the worst on other LLMs (0.00%–20.83%). While patches generated by zero-shot-completion (s.2 template [75]) pass test cases in many ExtractFix samples, most of them are unreasonable (as marked in yellow). The reason is that, by removing the vulnerable code and letting the LLMs generate the fixed code, LLMs may generate code that alters the program’s functionalities, although such code may still pass the test cases if the functionality tests are not comprehensive. Standard few-shot, Naive CoT, and RAG alternatively achieve the best correct rate. For example, Gemini achieves its best correct rates 10.68%–29.17% with Naive CoT. Claude3 achieves its best correct rates 54.17% and 20.76% with standard few-shot against the ExtractFix and BigVul+CVEFixes datasets, respectively, while it achieves the best correct rate 19.43% and 16.98% with RAG against the Zero-Day and PrimeVul-Subset dataset, respectively.

Between the datasets, the LLMs with any prompting strategy almost consistently perform better on the ExtractFix dataset (up to 70.83% correct rate) than on BigVul+CVEFixes (up to 30.24% correct rate). This is mainly because the ExtractFix dataset is a relatively simple testing set for traditional automatic vulnerability patching tools, while BigVul+CVEFixes is more complex which contains more challenging samples. Against the Zero-Day dataset, we notice

performance drops compared to the BigVul+CVESFixes dataset in some cases (e.g., Gemini’s 12.71% to 10.68% and Claude3’s 20.76%-19.43%), indicating the potential data leakage/contamination issue. PrimeVul-Subset dataset also presents varied performance, with correct rates ranging from 3.77% to 33.96%.

**Finding 5 (RQ1):** For vulnerability repair, with existing prompting strategies GPT-4 achieves up to 70.83% correct rate. Fine-tuning boosts the performance of Llama2-13b and makes them robust to different prompting strategies. Standard few-shot, Naive CoT, and RAG alternatively achieve the best correct rate among different LLMs and datasets. Zero-shot completion is the worst on most LLMs, as many of its generated patches break the program and thus are unusable. The low correct rates on PrimeVul-Full highlight the difficulty of repairing diverse types of vulnerabilities.

With **VSP strategies**, some of the LLMs show improved correct rates against the four datasets (up to 83.33% on GPT-4. Figure 8 shows an example where VSP helps an LLM repair a vulnerability successfully while the existing prompting strategies do not. Relative to the best performance with existing strategies, VSP’s improvements are up to 29.17 points (i.e., by 58.34%) in correct rate on GPT-3.5 against ExtractFix, and 12.50 points (by 17.65%) on GPT-4 against ExtractFix. On the two Llama2 models, the improvements are 100% (e.g., from 8.33% to 16.67%). The improvements on the Zero-Day dataset is large, from up to 20.40% (GPT-4 with RAG) to 31.07% (Claude3), indicating that VSP is effective for improvement zero-day vulnerability repair. Figure 8 shows an example for vulnerability repair. With VSP, GPT-3.5 successfully identifies the root cause of the buffer overflow and fixes it while Standard Prompting and Naive CoT simply add a boundary check without analyzing the vulnerability root cause correctly. Therefore, vulnerability-specific prompting adaptation is crucial for effective analysis. Furthermore, when applying VSP to the fine-tuned model, we observe additional performance gains. On the BigVul+CVESFixes dataset, the fine-tuned model with VSP achieves a correct rate of 30.08%, surpassing its best performance with existing strategies (27.12% with RAG). This again indicates that guiding the model’s reasoning process with vulnerability semantics remains beneficial and complementary, even after the model has been explicitly fine-tuned for the repair task. On the PrimeVul-Full dataset, VSP again demonstrates its effectiveness. Despite the increased difficulty, VSP improves repair performance in many cases. For instance, GPT-3.5 achieves a correct rate of 29.00% with VSP, compared to 20.04% with Standard Prompting. GPT-4 improves from 20.68% (Standard Prompting) to 31.26% (VSP). This suggests that the structured reasoning provided by VSP helps models generate more accurate patches even for a diverse set of vulnerabilities. VSP also leads to increases in other metrics, especially the percentage of patches that are valid and syntactically/semantically equivalent (to ground truth). Zero-shot VSP, however, is not even as helpful as naive CoT in most cases, and almost always underperforms the best-performing existing prompting strategies.

Statistically, Table 4 shows that VSP yields distinguishable gains ( $p < 0.001$ ) in the rate of correct patches compared to other strategies. The magnitude of this difference varies by baseline: VSP achieves a large effect size ( $d = -0.5712$ ) against Zero-Shot Completion (ZSC), indicating that vulnerability semantics focused guidance is effective compared to unguided code completion. Against Standard Prompting, the effect size is medium ( $d = -0.4628$ ). When compared to Naive CoT, which also uses reasoning steps, VSP maintains an advantage with a small effect size ( $d = -0.2496$ ), suggesting that the specific focus on vulnerability semantics contributes to the observed repair success.

**Finding 6 (RQ2):** VSP enables the LLMs to achieve up to 83.3% correct rate in vulnerability repair, where the best correct rate with existing prompting strategies is 70.83%. For smaller models, the improvements can be over 100%. VSP further enhances the fine-tuned model, pushing its performance beyond what is achievable with existing strategies. VSP also helps produce correct patches.

Table 7. Impact of temperature (Temp) on the LLM performance of vulnerability detection (with VSP)

LLM	Temp	VulChecker				BigVul-CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full								
		Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC					
Gemini	0	80.00%	26.12%	39.38%	0.3705	0.5395	74.19%	40.05%	52.02%	0.5214	0.5762	33.33%	54.55%	41.38%	0.6064	0.6061	62.51%	40.24%	48.96%	0.4849	0.5193	63.75%	31.00%	41.72%	0.4237	0.4786
	0.25	77.65%	26.09%	39.05%	0.3627	0.5286	74.09%	39.97%	51.92%	0.5204	0.5753	33.01%	53.13%	40.72%	0.6020	0.5981	63.89%	40.35%	49.46%	0.4912	0.5265	66.94%	30.69%	42.09%	0.4323	0.4926
	0.5	74.12%	25.40%	37.84%	0.3436	0.5091	74.63%	40.09%	52.16%	0.5234	0.5783	33.01%	57.63%	41.98%	0.9066	0.6207	61.11%	41.12%	49.16%	0.4849	0.5169	68.28%	31.00%	42.64%	0.4393	0.5006
	0.75	70.59%	24.90%	36.81%	0.3267	0.4905	74.84%	40.14%	52.25%	0.5245	0.5797	31.07%	53.33%	39.26%	0.8427	0.5943	59.72%	41.35%	48.86%	0.4806	0.5113	69.64%	30.38%	42.30%	0.4390	0.5041
Claude3	0	69.41%	24.08%	35.76%	0.3135	0.4810	74.52%	40.06%	52.11%	0.5227	0.5778	30.10%	54.39%	38.75%	0.6543	0.5972	56.94%	39.42%	46.59%	0.4567	0.4882	72.43%	29.77%	42.20%	0.4432	0.5147
	0.25	57.57%	45.24%	50.67%	0.4563	0.5329	51.28%	76.92%	61.54%	0.6164	0.6504	32.04%	70.21%	44.00%	0.2198	0.6812	46.96%	64.12%	54.22%	0.5384	0.5633	49.08%	19.46%	27.87%	0.2800	0.3495
	0.5	55.29%	45.37%	50.78%	0.4575	0.5339	51.28%	76.71%	61.47%	0.6155	0.6494	33.01%	66.67%	44.16%	0.1912	0.6659	44.44%	69.57%	54.24%	0.5467	0.5783	48.10%	20.24%	28.49%	0.2838	0.3486
	0.75	60.00%	47.22%	52.85%	0.4807	0.5539	51.39%	77.62%	61.83%	0.6201	0.6544	34.95%	61.02%	44.44%	0.1396	0.6425	45.83%	63.46%	53.23%	0.5288	0.5548	49.02%	18.65%	27.02%	0.2745	0.3452
GPT-3.5	0	55.29%	45.63%	50.00%	0.4487	0.5245	51.39%	78.12%	61.99%	0.6222	0.6569	33.01%	60.71%	42.77%	0.1309	0.6361	44.44%	64.00%	52.46%	0.5230	0.5504	49.02%	19.40%	27.80%	0.2793	0.3489
	0.25	58.79%	38.18%	54.23%	0.5025	0.5667	33.08%	72.03%	45.34%	0.4754	0.5384	33.01%	59.65%	42.50%	0.1194	0.6308	38.96%	63.29%	48.23%	0.4861	0.5203	39.88%	26.32%	31.71%	0.3013	0.3390
	0.5	54.12%	39.74%	56.79%	0.5289	0.5896	32.52%	72.27%	44.85%	0.4722	0.5369	32.04%	58.93%	41.51%	0.1091	0.6247	37.50%	57.45%	45.38%	0.4526	0.4840	41.87%	26.05%	32.12%	0.3072	0.3473
	0.75	52.94%	39.21%	55.90%	0.5197	0.5816	32.09%	71.33%	44.26%	0.4656	0.5302	33.01%	61.82%	43.04%	0.1427	0.6416	40.28%	58.00%	47.54%	0.4718	0.5002	39.78%	25.53%	31.10%	0.2955	0.3346
GPT-4	0	67.11%	56.12%	60.11%	0.5609	0.6198	56.72%	48.94%	52.54%	0.5965	0.5366	44.37%	55.33%	49.56%	0.1083	0.6136	59.72%	47.25%	52.76%	0.5169	0.5408	59.52%	34.10%	36.61%	0.3485	0.3761
	0.25	68.24%	56.86%	62.03%	0.5826	0.6396	56.18%	48.62%	52.13%	0.5021	0.5324	43.40%	55.35%	48.46%	0.0856	0.6053	56.94%	48.81%	52.56%	0.5133	0.5351	37.55%	33.76%	35.55%	0.3374	0.3648
	0.5	64.71%	59.38%	62.98%	0.5927	0.6468	55.65%	48.51%	51.84%	0.4990	0.5294	43.40%	55.83%	49.33%	0.1117	0.6127	58.33%	50.60%	54.19%	0.5299	0.5589	37.17%	35.44%	36.29%	0.3451	0.3714
	1	61.18%	58.43%	59.77%	0.5577	0.6152	55.44%	48.37%	51.66%	0.4972	0.5276	44.37%	56.28%	50.00%	0.1214	0.6173	56.94%	48.24%	52.23%	0.5100	0.5323	39.03%	36.15%	37.54%	0.3579	0.3840
Llama2-7b	0	42.85%	37.50%	39.99%	0.3382	0.4271	53.12%	44.74%	48.57%	0.4651	0.4983	38.23%	60.93%	46.98%	0.1478	0.6502	44.73%	47.72%	46.18%	0.4478	0.4704	39.96%	12.31%	18.82%	0.1844	0.2693
	0.25	43.53%	37.00%	40.00%	0.3377	0.4277	53.20%	45.00%	48.75%	0.4670	0.5000	37.86%	58.21%	45.88%	0.1140	0.6357	44.44%	48.48%	46.38%	0.4502	0.4729	40.36%	12.07%	18.58%	0.1827	0.2701
	0.5	44.71%	36.73%	39.34%	0.3308	0.4210	52.77%	44.96%	48.55%	0.4648	0.4977	36.89%	55.88%	44.44%	0.0826	0.6216	43.06%	50.00%	46.27%	0.4504	0.4737	41.16%	12.67%	19.38%	0.1913	0.2770
	0.75	43.53%	35.58%	39.15%	0.3277	0.4206	53.09%	45.15%	48.80%	0.4674	0.5002	34.95%	59.02%	43.90%	0.1170	0.6325	43.06%	53.45%	47.69%	0.4671	0.4909	42.40%	12.80%	19.66%	0.1958	0.2836
Llama2-13b	0	43.53%	37.00%	40.00%	0.3377	0.4277	53.41%	45.42%	49.09%	0.4705	0.5031	33.01%	57.63%	41.98%	0.9066	0.6207	41.67%	50.00%	45.45%	0.4430	0.4670	40.28%	12.28%	18.83%	0.1849	0.2708
	0.25	54.12%	32.17%	40.35%	0.3432	0.4518	51.63%	50.48%	51.04%	0.4907	0.5199	39.22%	63.49%	48.48%	0.1804	0.6655	48.32%	48.71%	48.52%	0.4712	0.4928	57.87%	12.92%	21.13%	0.2345	0.3595
	0.5	58.82%	33.78%	42.92%	0.3745	0.4813	51.28%	50.47%	50.87%	0.4889	0.5181	39.81%	62.12%	48.52%	0.1665	0.6601	48.61%	47.95%	48.28%	0.4685	0.4904	58.44%	13.57%	22.03%	0.2439	0.3656
	0.75	56.47%	32.65%	41.38%	0.3561	0.4649	51.07%	50.53%	50.80%	0.4881	0.5174	38.83%	63.49%	48.19%	0.1791	0.6645	45.83%	48.53%	47.14%	0.4576	0.4798	55.52%	13.98%	22.33%	0.2417	0.3534
Falcon	0	54.12%	32.17%	40.35%	0.3432	0.4518	50.96%	50.64%	50.80%	0.4882	0.5174	40.78%	60.80%	48.55%	0.1435	0.6519	47.22%	49.28%	48.23%	0.4685	0.4905	58.30%	14.54%	23.27%	0.2551	0.3697
	0.25	55.20%	33.33%	41.39%	0.3574	0.4630	50.85%	50.53%	50.69%	0.4871	0.5164	40.78%	57.53%	47.73%	0.1116	0.6296	44.44%	47.76%	46.04%	0.4465	0.4692	57.71%	14.10%	22.66%	0.2485	0.3647
	0.5	63.53%	27.41%	38.29%	0.3317	0.4709	44.77%	36.83%	40.41%	0.3798	0.4186	35.29%	43.37%	38.92%	-0.1098	0.5551	47.86%	35.87%	41.01%	0.3959	0.4264	14.58%	11.29%	12.73%	0.1012	0.1407
	0.75	62.35%	27.75%	38.41%	0.3313	0.4672	44.78%	37.00%	40.52%	0.3809	0.4195	35.92%	43.02%	39.15%	-0.1181	0.5549	48.61%	35.00%	40.70%	0.3936	0.4256	15.02%	11.74%	13.18%	0.1059	0.1451
CodeLlama	0	62.35%	27.46%	38.13%	0.3284	0.4658	44.99%	37.25%	40.75%	0.3833	0.4218	34.95%	42.86%	38.50%	-0.1185	0.5517	50.00%	34.95%	41.14%	0.3991	0.4321	15.32%	11.62%	13.22%	0.1062	0.1460
	0.25	61.18%	27.37%	37.82%	0.3235	0.4599	44.78%	37.14%	40.60%	0.3817	0.4202	33.01%	40.96%	36.56%	-0.1485	0.5373	47.22%	33.66%	39.31%	0.3793	0.4122	16.09%	12.09%	13.80%	0.1123	0.1520
	0.5	62.35%	28.34%	38.97%	0.3372	0.4702	44.35%	36.98%	40.33%	0.3788	0.4174	33.98%	42.17%	37.63%	-0.1287	0.5458	48.61%	33.02%	39.33%	0.3809	0.4157	15.93%	12.33%	13.90%	0.1134	0.1525
	0.75	47.11%	23.53%	7.84%	0.0693	0.1835	14.75%	9.55%	11.59%	0.0745	0.1939	14.12%	15.09%	14.86%	-0.6208	0.3637	11.19%	16.25%	13.26%	0.1154	0.1504	12.50%	9.38%	10.71%	0.0800	0.1210
Qwen2.5	0	5.88%	31.25%	9.90%	0.1026	0.2274	13.11%	8.64%	10.41%	0.0620	0.1255	13.59%	15.22%	13.47%	-0.6249	0.3601	11.11%	18.60%	13.91%	0.1259	0.1617	11.75%	9.00%	10.20%	0.0768	0.1155
	0.25	37.65%	27.59%	31.84%	0.2443	0.3538	27.72%	50.39%	35.76%	0.3558	0.4044	32.62%	29.71%	31.10%	-0.4475	0.4801	32.66%	35.90%	34.20%	0.3253	0.3527	21.54%	20.45%	20.98%	0.1877	0.2204
	0.5	36.47%	26.05%	30.39%	0.2275	0.3408	28.25%	50.48%	36.23%	0.3596	0.4073	34.95%	29.75%	32.14%	-0.4832	0.4861	34.72%	34.72%	34.72%	0.3294	0.3569	21.97%	21.07%	21.51%	0.1952	0.2256
	0.75	36.47%	25.62%	30.10%	0.2239	0.3386	27.83%	50.29%	35.83%	0.3561	0.4045	34.95%	29.27%	31.86%	-0.5048	0.4837	31.94%	30.26%	31.08%	0.2916	0.3211	22.63%	21.06%	21.82%	0.1962	0.2288
DeepSeek-Coder-V2	0	37.65%	25.60%	30.48%	0.2279	0.3439	27.83%	50.39%	35.85%	0.3565	0.4050	33.01%	27.20%	29.82%	-0.5665	0.4685	31.94%	30.67%	31.29%	0.2938	0.3231	22.18%	21.06%	21.61%	0.1941	0.2265
	0.25	28.24%	32.00%	30.00%	0.2370	0.3330	17.27%	37.07%	23.56%	0.2333	0.2876	30.12%	31.00%	30.55%	-0.3693	0.4803	25.21%	33.36%	28.72%	0.2732	0.3039	39.13%	11.64%	17.94%	0.1749	0.2619
	0.5	29.41%	31.65%	30.49%	0.2401	0.3366	17.38%	37.05%	23.66%	0.2340	0.2880	30.10%	30.39%	30.24%	-0.3884	0.4772	25.00%	32.14%	28.13%	0.2663	0.2968	37.96%	11.41%	17.54%	0.1694	0.2551
	0.75	27.06%	29.87%	28.40%	0.2183	0.3170	17.80%	37.36%																		

Table 8. Impact of temperature (Temp) on the LLM performance of vulnerability classification (with VSP)

LLM	Temp	VulChecker Dataset					BigVul-CVEFixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full				
		MacF1	MicroF1	MCC	MacAu	MicAu	MacF1	MicroF1	MCC	MacAu	MicAu	MacF1	MicroF1	MCC	MacAu	MicAu	MacF1	MicroF1	MCC	MacAu	MicAu	MacF1	MicroF1	MCC	MacAu	MicAu
Gemini	0	28.28%	31.40%	0.2081	0.3976	0.5245	39.47%	47.27%	0.3544	0.4523	0.5903	21.37%	27.06%	0.1982	0.2967	0.5510	28.44%	35.24%	0.2228	0.3515	0.4670	4.20%	23.32%	0.1674	0.1386	0.4544
	0.25	28.73%	31.34%	0.2089	0.3984	0.5255	40.10%	47.84%	0.3551	0.4531	0.5915	21.20%	27.28%	0.1986	0.2972	0.5521	27.87%	35.74%	0.2189	0.3579	0.5063	4.19%	23.04%	0.1677	0.1388	0.4552
	0.75	29.02%	31.71%	0.2206	0.3480	0.4625	40.58%	47.84%	0.3509	0.4478	0.5846	21.37%	27.17%	0.1962	0.2938	0.5457	28.26%	35.02%	0.2206	0.3480	0.4625	4.21%	22.85%	0.1658	0.1372	0.4499
Claude3	0	28.44%	31.27%	0.2089	0.3993	0.5267	40.50%	47.67%	0.3559	0.4541	0.5928	21.63%	27.49%	0.1990	0.2979	0.5533	28.55%	34.95%	0.2237	0.3529	0.4690	4.28%	23.88%	0.1681	0.1392	0.4562
	0.25	27.98%	31.27%	0.2089	0.3993	0.5267	39.69%	48.22%	0.3559	0.4541	0.5928	21.80%	27.60%	0.1990	0.2979	0.5533	29.00%	35.92%	0.2237	0.3529	0.4690	4.28%	22.89%	0.1681	0.1392	0.4562
	0.75	29.93%	32.04%	0.2230	0.4580	0.5549	31.45%	36.92%	0.2174	0.3438	0.4583	37.16%	40.16%	0.3345	0.4677	0.6160	25.95%	34.87%	0.2142	0.3501	0.4952	3.45%	14.95%	0.1039	0.0617	0.4136
GPT-3.5	0	35.09%	37.42%	0.2740	0.4105	0.5773	27.44%	28.03%	0.1497	0.4283	0.4916	14.27%	19.23%	0.0646	0.2425	0.3918	30.30%	39.16%	0.2571	0.3983	0.5754	3.93%	16.89%	0.1048	0.0622	0.4172
	0.25	30.47%	31.53%	0.2236	0.4654	0.5458	31.45%	37.22%	0.2210	0.3494	0.4658	37.53%	40.56%	0.3399	0.4753	0.6259	25.59%	34.17%	0.2072	0.3960	0.5224	3.50%	14.74%	0.1056	0.0617	0.4136
	0.75	30.96%	32.16%	0.2222	0.4563	0.5528	30.95%	36.69%	0.2166	0.3425	0.4566	38.13%	39.75%	0.2767	0.4934	0.5747	26.00%	34.04%	0.2031	0.3882	0.5121	3.50%	14.62%	0.1035	0.0615	0.4120
GPT-4	0	30.52%	32.16%	0.2222	0.4563	0.5528	31.32%	36.47%	0.2166	0.3425	0.4566	37.52%	39.27%	0.2767	0.4934	0.5747	25.48%	33.42%	0.2100	0.2927	0.3966	3.46%	14.39%	0.1035	0.0615	0.4120
	0.25	30.10%	32.35%	0.2209	0.4536	0.5495	30.75%	36.98%	0.2154	0.3405	0.4540	38.05%	38.72%	0.2751	0.4905	0.5713	25.83%	33.36%	0.2019	0.3860	0.5091	3.40%	14.22%	0.1029	0.0611	0.4096
	0.75	35.09%	37.42%	0.2740	0.4105	0.5773	27.44%	28.03%	0.1497	0.4283	0.4916	14.27%	19.23%	0.0646	0.2425	0.3918	30.30%	39.16%	0.2571	0.3983	0.5754	3.93%	16.89%	0.1048	0.0622	0.4172
Llama2-7b	0	34.98%	37.12%	0.3245	0.4862	0.6538	27.22%	28.22%	0.1711	0.3659	0.4837	14.07%	18.90%	0.0766	0.2873	0.4640	30.73%	38.69%	0.3045	0.4717	0.6815	3.87%	16.58%	0.1241	0.0737	0.4942
	0.25	34.84%	36.38%	0.2670	0.4000	0.5626	27.06%	27.94%	0.1459	0.4173	0.4790	13.90%	19.00%	0.0630	0.2363	0.3818	30.48%	37.92%	0.2505	0.3881	0.5607	3.87%	16.61%	0.1021	0.0606	0.4066
	0.75	35.39%	35.94%	0.2748	0.4117	0.5789	26.68%	27.82%	0.1449	0.3097	0.4095	13.68%	18.92%	0.0648	0.2432	0.3929	30.18%	38.45%	0.2578	0.3994	0.5770	3.86%	16.35%	0.1051	0.0624	0.4184
Llama2-13b	0	35.54%	35.44%	0.3049	0.4568	0.6424	26.79%	27.71%	0.1607	0.3437	0.4544	13.57%	19.07%	0.0719	0.2699	0.4359	30.18%	38.06%	0.2441	0.3859	0.5145	3.90%	16.28%	0.1166	0.0692	0.4642
	0.25	44.46%	42.10%	0.3133	0.5147	0.5752	33.00%	34.97%	0.2297	0.4718	0.5715	35.98%	39.04%	0.2798	0.4192	0.5895	30.73%	39.53%	0.2625	0.4066	0.5875	5.59%	20.95%	0.1690	0.1399	0.4587
	0.75	43.84%	42.02%	0.3489	0.5347	0.6370	32.34%	33.32%	0.2328	0.4782	0.5793	35.26%	39.27%	0.2836	0.4249	0.5975	30.48%	39.85%	0.2661	0.4122	0.5951	5.66%	21.16%	0.1713	0.1418	0.4649
DeepSeek-Coder-V2	0	44.36%	42.77%	0.3507	0.5763	0.6440	32.92%	35.67%	0.2572	0.5283	0.6399	34.70%	39.55%	0.3133	0.4693	0.6600	30.48%	39.05%	0.2939	0.4553	0.6578	5.59%	20.78%	0.1892	0.1566	0.5135
	0.25	44.10%	42.69%	0.3490	0.5628	0.6440	33.38%	34.96%	0.2629	0.5400	0.6542	35.32%	39.90%	0.3203	0.4798	0.6748	30.79%	39.68%	0.3005	0.4655	0.6725	5.48%	20.57%	0.1934	0.1601	0.5250
	0.75	43.30%	43.37%	0.3546	0.5453	0.6474	33.32%	35.17%	0.2366	0.4860	0.5888	34.61%	39.82%	0.2882	0.4318	0.6073	31.15%	39.44%	0.2704	0.4189	0.6052	5.48%	20.28%	0.1741	0.1441	0.4725
CodeLlama	0	20.58%	20.51%	0.0706	0.2987	0.3930	22.27%	22.60%	0.1110	0.3082	0.4430	18.13%	18.86%	0.0758	0.3039	0.4624	26.88%	30.77%	0.2316	0.4427	0.5840	1.00%	4.37%	0.0354	0.0181	0.1044
	0.25	20.33%	20.84%	0.0729	0.3082	0.4055	22.27%	22.15%	0.1145	0.3180	0.4571	18.35%	18.52%	0.0782	0.3136	0.4771	27.36%	30.65%	0.2390	0.4568	0.6025	1.02%	4.40%	0.0365	0.0186	0.1077
	0.75	20.17%	20.67%	0.0695	0.2941	0.3869	22.49%	21.88%	0.1093	0.3033	0.4361	18.68%	18.52%	0.0875	0.3031	0.4152	27.75%	30.95%	0.2280	0.4358	0.5748	1.01%	4.48%	0.0348	0.0178	0.1027
Falcon	0	20.29%	20.34%	0.0678	0.2867	0.3771	22.67%	21.49%	0.1065	0.2957	0.4251	18.45%	18.67%	0.0727	0.2916	0.4438	27.97%	30.89%	0.2223	0.4248	0.5604	1.01%	4.51%	0.0339	0.0173	0.1002
	0.25	19.89%	20.30%	0.0654	0.2768	0.3641	22.67%	21.27%	0.1028	0.2855	0.4104	18.79%	18.74%	0.0823	0.2853	0.3908	27.58%	30.52%	0.2146	0.4102	0.5410	1.01%	4.42%	0.0328	0.0167	0.0967
	0.75	32.48%	38.19%	0.2662	0.4588	0.6628	31.16%	34.94%	0.2537	0.4003	0.5319	27.12%	31.02%	0.2369	0.4528	0.5973	27.79%	31.14%	0.2369	0.4528	0.5973	1.08%	4.74%	0.0270	0.0220	0.0818
PT-Llama2	0	32.93%	37.43%	0.2836	0.4249	0.5976	30.60%	34.59%	0.2279	0.3597	0.4779	27.66%	31.64%	0.2129	0.4069	0.5367	27.73%	31.45%	0.2129	0.4069	0.5367	1.07%	4.83%	0.0242	0.0198	0.0735
	0.25	32.80%	36.83%	0.3116	0.4668	0.6565	30.66%	34.66%	0.2504	0.3952	0.5251	28.16%	32.15%	0.2339	0.4471	0.5897	28.29%	31.51%	0.2339	0.4471	0.5897	1.04%	4.90%	0.0266	0.0217	0.0808
	0.75	32.74%	36.53%	0.2359	0.4729	0.4971	30.23%	33.97%	0.2368	0.3736	0.4964	27.99%	31.95%	0.2212	0.4227	0.5576	27.72%	32.02%	0.2212	0.4227	0.5576	1.04%	4.96%	0.0252	0.0205	0.0764
CodeLlama	0	32.15%	35.80%	0.2284	0.4692	0.5884	29.93%	34.44%	0.2236	0.3528	0.4688	27.71%	32.54%	0.2089	0.3992	0.5266	27.33%	31.95%	0.2089	0.3992	0.5266	1.03%	5.06%	0.0271	0.0125	0.0619
	0.25	30.20%	28.57%	0.2449	0.4891	0.6540	27.22%	24.80%	0.1665	0.4761	0.5464	18.30%	19.04%	0.0748	0.3000	0.4565	10.71%	15.00%	0.0359	0.2486	0.4002	1.32%	1.06%	0.0062	0.0073	0.0490
	0.75	30.74%	28.97%	0.2415	0.4823	0.6449	27.11%	24.35%	0.1642	0.4695	0.5389	18.45%	19.08%	0.0738	0.2959	0.4502	10.52%	15.12%	0.0354	0.2451	0.3947	1.33%	1.07%	0.0061	0.0072	0.0541
CodeLlama	0	31.36%	29.79%	0.2348	0.4690	0.6272	26.78%	23.68%	0.1596	0.4566	0.5241	18.66%	18.89%	0.0841	0.2916	0.3993	10.27%	14.82%	0.0345	0.2384	0.3839	1.31%	1.07%	0.0060	0.0070	0.0526
	0.25	31.86%	29.73%	0.2359	0.4706	0.6285	26.67%	23.44%	0.1600	0.4575	0.5251	18.52%	19.15%	0.0719	0.2883	0.4387	10.41%	14.85%	0.0345	0.2384	0.3846	1.29%	1.07%	0.0060	0.0070	0.0527
	0.75	25.79%	25.69%	0.1409	0.3013	0.3984	21.08%	21.68%	0.0961	0.2669	0.3837	20.99%	22.88%	0.0961	0.2669	0.3837	16.12%	21.28%	0.0656	0.2632	0.4096	0.46%	2.30%	0.0171	0.0125	0.0619
CodeLlama	0	25.53%	25.69%	0.1640	0.3506	0.4635	21.16%	21.89%	0.1119	0.3106	0.4465	20.87%	22.65%	0.1119	0.3106	0.4465	16.25%	21.11%	0.0764	0.3063	0.4061	0.46%	2.32%	0.0199	0.0175	0.0694
	0.25	25.12%	25.33%	0.1675	0.3581	0.4734	20.82%	21.72%	0.1143	0.3172	0.4560	21.12%	22.74%	0.1143	0.3172	0.4560	16.38%	21.11%	0.0780	0.3128	0.4060	0.47%	2.36%	0.0204	0.0113	0.0682
	0.75	25.32%	25.84%	0.1695	0.3624	0.4792	20.78%	21.89%	0.1156	0.3211	0.4615	21.33														

Table 9. Impact of temperature (Temp) on the LLM performance of vulnerability repair (with VSP)

LLM	Temp	ExtractFix Dataset					BigVul-CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full				
		Valid	SynEq	SemEq	Plus	Unres	Correct	SynEq	SemEq	Plus	Correct	SynEq	SemEq	Plus	Correct	SynEq	SemEq	Plus	Correct	SynEq	SemEq	Plus	Correct
Gemini	0	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	1.27%	13.14%	2.91%	5.83%	6.80%	15.53%	3.77%	3.77%	11.32%	18.87%	0.42%	3.94%	7.90%	12.26%
	0.25	29.17%	12.50%	8.33%	8.33%	0.00%	29.17%	6.36%	5.93%	1.69%	13.98%	2.91%	4.85%	6.80%	14.56%	3.77%	5.66%	11.32%	20.75%	0.42%	3.94%	7.90%	12.26%
	0.5	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.93%	1.69%	13.98%	2.91%	3.88%	7.77%	14.56%	3.77%	5.66%	11.32%	20.75%	0.42%	3.94%	7.90%	12.26%
	0.75	33.33%	12.50%	8.33%	12.50%	0.00%	33.33%	6.36%	5.51%	1.69%	13.56%	2.91%	4.85%	7.77%	15.53%	5.66%	3.77%	9.43%	18.87%	0.42%	3.94%	7.90%	12.26%
	1	29.17%	12.50%	8.33%	8.33%	0.00%	29.17%	6.36%	5.93%	1.27%	13.56%	2.91%	4.85%	7.77%	15.53%	5.66%	3.77%	11.32%	20.75%	0.42%	3.94%	8.12%	12.48%
Claude3	0	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.32%	6.36%	22.88%	11.65%	10.80%	8.74%	31.07%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.04%	13.40%
	0.25	62.50%	25.00%	20.83%	16.67%	0.00%	62.50%	7.20%	8.90%	5.93%	22.03%	11.65%	9.71%	9.71%	31.07%	3.77%	3.77%	9.43%	16.98%	3.68%	3.68%	6.04%	13.40%
	0.5	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	8.47%	5.51%	21.19%	11.65%	9.71%	9.71%	31.07%	3.77%	3.77%	7.55%	15.09%	3.68%	3.45%	6.04%	13.17%
	0.75	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.63%	8.47%	5.93%	22.03%	11.65%	8.74%	8.74%	29.13%	3.77%	3.77%	9.43%	16.98%	3.68%	3.23%	6.04%	12.95%
	1	66.67%	20.83%	25.00%	20.83%	0.00%	66.67%	7.20%	8.47%	5.93%	21.61%	11.65%	8.74%	8.74%	29.13%	3.77%	3.77%	7.55%	15.09%	3.68%	3.23%	6.04%	12.95%
GPT-3.5	0	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.75%	8.47%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	16.98%	32.08%	0.22%	15.39%	13.38%	29.00%
	0.25	75.00%	25.00%	33.33%	16.67%	0.00%	75.00%	9.75%	8.47%	12.29%	30.51%	15.53%	3.88%	23.30%	42.72%	7.55%	9.43%	15.09%	32.08%	0.22%	15.39%	13.61%	29.22%
	0.5	79.17%	25.00%	37.50%	16.67%	0.00%	79.17%	9.75%	8.90%	12.29%	31.36%	15.53%	4.85%	23.30%	43.69%	7.55%	9.43%	16.98%	33.96%	0.22%	15.39%	13.61%	29.22%
	0.75	79.17%	25.00%	41.67%	12.50%	0.00%	79.17%	9.32%	8.47%	12.29%	30.51%	15.53%	4.85%	23.30%	43.69%	7.55%	7.55%	15.09%	30.19%	0.22%	15.17%	13.61%	29.00%
	1	79.17%	29.17%	37.50%	12.50%	0.00%	79.17%	9.32%	8.90%	13.14%	31.36%	15.53%	4.85%	23.30%	43.69%	5.66%	7.55%	15.09%	28.30%	0.00%	15.17%	13.83%	29.00%
GPT-4	0	83.33%	25.00%	41.67%	16.67%	0.00%	83.33%	17.02%	12.77%	19.15%	48.93%	16.50%	6.80%	20.39%	43.69%	7.55%	9.43%	20.75%	37.74%	1.84%	7.36%	22.07%	31.26%
	0.25	83.33%	25.00%	37.50%	20.83%	0.00%	83.33%	17.02%	12.77%	19.15%	48.93%	16.50%	6.80%	21.36%	44.66%	9.43%	9.43%	20.75%	39.62%	1.84%	7.36%	21.84%	31.04%
	0.5	83.33%	20.83%	41.67%	20.83%	0.00%	83.33%	17.02%	12.34%	19.15%	48.31%	16.50%	6.80%	20.39%	43.69%	9.43%	7.55%	18.87%	35.85%	1.84%	7.36%	22.07%	31.26%
	0.75	83.33%	20.83%	41.67%	20.83%	0.00%	83.33%	17.02%	12.34%	19.15%	48.31%	16.50%	6.80%	21.36%	44.66%	9.43%	7.55%	18.87%	35.85%	1.84%	7.36%	22.07%	31.26%
	1	79.17%	20.83%	37.50%	20.83%	0.00%	79.17%	17.45%	12.34%	19.15%	48.93%	16.50%	5.83%	20.39%	42.72%	9.43%	7.55%	20.75%	37.74%	1.84%	7.13%	22.07%	31.04%
Llama2-7b	0	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	2.06%	0.00%	5.06%	7.13%
	0.25	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	3.77%	5.66%	2.06%	0.00%	5.06%	7.13%
	0.5	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	3.77%	5.66%	2.29%	0.00%	5.06%	7.35%
	0.75	16.67%	8.33%	8.33%	0.00%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	3.77%	5.66%	2.29%	0.00%	5.06%	7.35%
	1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	1.27%	0.00%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	2.06%	0.00%	5.06%	7.13%
Llama2-13b	0	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.63%	7.90%
	0.25	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.63%	7.90%
	0.5	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.63%	7.90%
	0.75	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	1.69%	1.27%	0.42%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.85%	8.12%
	1	16.67%	12.50%	4.17%	0.00%	0.00%	16.67%	1.69%	1.27%	0.42%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.85%	8.12%
Codellama	0	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%
	0.25	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%
	0.5	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%
	0.75	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.42%	1.69%	2.12%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%
	1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	0.00%	1.94%	0.97%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%
Qwen2.5-72b	0	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	0.97%	1.94%	1.94%	4.85%	1.89%	3.77%	1.89%	7.55%	0.00%	2.21%	3.49%	5.70%
	0.25	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	0.97%	0.97%	2.91%	4.85%	1.89%	3.77%	1.89%	7.55%	0.00%	1.98%	3.27%	5.25%
	0.5	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	0.97%	0.97%	2.91%	4.85%	1.89%	3.77%	1.89%	5.66%	0.00%	1.98%	3.27%	5.25%
	0.75	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.97%	5.08%	0.97%	0.97%	2.91%	4.85%	1.89%	3.77%	3.77%	9.43%	0.00%	1.98%	3.27%	5.25%
	1	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.97%	5.08%	0.97%	0.97%	2.91%	4.85%	1.89%	3.77%	3.77%	9.43%	0.00%	1.98%	3.27%	5.25%
DSC-V2	0	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	0.97%	1.94%	3.88%	1.89%	1.89%	1.89%	5.66%	0.21%	1.85%	2.05%	3.88%
	0.25	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	0.97%	1.94%	3.88%	1.89%	1.89%	1.89%	5.66%	0.21%	1.85%	1.82%	3.88%
	0.5	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	0.97%	1.94%	3.88%	1.89%	1.89%	1.89%	5.66%	0.21%	1.62%	2.05%	3.88%
	0.75	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.69%	0.85%	2.12%	4.66%	1.94%	0.97%	1.94%	4.85%	1.89%	1.89%	1.89%	5.66%	0.44%	1.40%	2.05%	3.88%
	1	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.69%	0.85%	2.12%	4.66%	1.94%	0.97%	1.94%	4.85%	3.77%	1.89%	1.89%	7.55%	0.44%	1.40%	2.05%	3.88%
FT-Llama2	0	25.00%	4.17%	12.50%	8.33%	0.00%	25.00%	0.00%	14.41%	15.68%	30.08%	0.00%	4.85%	13.59%	18.45%	0.00%	9.43%	18.87%	28.30%	0.63%	13.08%	11.38%	25.09%
	0.25	25.00%	4.17%	12.50%	8.33%	0.00%	25.00%	0.00%	14.41%	15.68%	30.08%	0.00%	4.85%	13.59%	18.45%	1.89%	9.43%	18.87%	26.42%	0.63%	13.08%	11.15%	24.87%
	0.5	25.00%	4.17%	16.67%	4.17%	0.00%	25.00%	0.00%	13.98%	15.68%	29.66%	0.00%	4.85%	13.59%	18.45%	1.89%	9.43%	18.87%	26.42%	0.63%	13.08%	11.15%	24.87%
	0.75	29.17%	8.33%	16.67%	4.17%	0.00%	29.17%	0.00%	13.56%	15.68%	29.24%	0.00%	4.85%										

in on BigVul+CVFixes at higher temperatures—the overarching trend supports the use of deterministic settings to maintain consistent and precise classification boundaries.

Table 9 illustrates the effect of temperature on vulnerability repair performance. Consistent with the findings for detection and classification, the deterministic setting (temperature 0) tends to be the most effective for the majority of models, particularly the high-performing commercial ones. For instance, Claude3 sees a reduction in correct patches on the Zero-Day dataset from 31.07% at the default setting to 29.13% at temperature 1. While certain code-specific models like DeepSeek-Coder-V2 exhibit minor fluctuations—occasionally benefiting slightly from higher diversity (e.g., improving from 3.88% to 4.85% on Zero-Day)—these gains are inconsistent across datasets. The fine-tuned Llama2-13b model also favors the default temperature setting on the large-scale BigVul+CVFixes dataset (30.08% at temperature 0 vs. 29.24% at temperature 1).

The statistical analysis results in Table 10 corroborate the stability of VSP across various temperature settings. For vulnerability detection, the performance differences between the default temperature (0) and higher settings are generally not statistically significant ( $p > 0.05$ ), with the exception of the most diverse setting (0 vs 1). In classification, results are similarly stable, with most comparisons failing to reach statistical significance. While the differences in repair performance are statistically distinguishable ( $p < 0.001$ ), the effect sizes across all three tasks are consistently negligible ( $|d| \leq 0.015$ ).

**Finding 7 (RQ3):** Lower temperatures yield more robust and optimal effectiveness across vulnerability detection, classification, and repair. Increasing the temperature to introduce diversity generally degrades performance or offers inconsistent gains (although the differences/changes are mostly not statistically significant nor large), indicating that deterministic generation is preferable for vulnerability analysis to ensure both stable performance and reproducibility.

## 4.2 Impact of Top-P

We then examine the impact of Top-P. We again evaluate values of 0, 0.25, 0.5, 0.75, and 1.

Table 11 summarizes the performance of LLMs across different Top-P values for vulnerability detection. Unlike temperature, where the lowest setting was almost universally superior, the impact of Top-P is more nuanced. For several models, the unconstrained setting (Top-P = 1) yields the highest effectiveness. For instance, Gemini sees a steady improvement on the VulChecker dataset as Top-P increases, rising from 37.71% F1 at Top-P 0 to 39.38% at Top-P 1. However, we observe a notable exception with GPT-3.5 on the Zero-Day dataset, where performance drops significantly from 48.68% at Top-P 0 to 42.50% at Top-P 1. This suggests that for certain models facing novel vulnerabilities, constraining the token search space can prevent the generation of low-probability, incorrect reasoning paths. The MCC and AuPRC metrics further nuance the impact of Top-P on detection reliability, particularly regarding the stability of predictions on unseen data. For example, on the Zero-Day dataset, GPT-3.5's MCC plummets from 0.2850 at Top-P 0 to 0.1194 at Top-P 1, and its AuPRC drops from 0.7174 to 0.6308. Conversely, on established datasets like VulChecker, Gemini sees consistent gains, with MCC rising from 0.3487 at Top-P 0 to 0.3705 at Top-P 1, suggesting that for known patterns, allowing the model to explore the token space can refine decision boundaries.

Table 12 presents the impact of Top-P on vulnerability classification. We observe that restricting the token search space (lower Top-P) does not consistently improve performance and, in some cases, significantly hinders it. For example, Claude3 achieves its best Zero-Day performance (40.24%) at Top-P 0.75. While there are minor regressions for models like GPT-4 on the VulChecker dataset (dropping from 44.16% to 42.10%), its performance on the critical Zero-Day dataset remains stable (38.33% to 39.04%). Unlike detection, where lower Top-P often stabilizes performance on novel

Table 11. Impact of Top-P on the LLM performance of vulnerability detection (with VSP)

LLM	Top-P	VulChecker				BigVul-CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full								
		Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC	Rec	Prec	F1	MCC	AnPRC					
Gemma	0	77.65%	24.91%	37.71%	0.3487	0.5227	73.67%	40.03%	51.88%	0.5193	0.5736	33.01%	53.13%	40.72%	0.0420	0.5981	58.33%	39.25%	46.93%	0.4613	0.4941	68.27%	29.42%	41.12%	0.4265	0.4927
	0.25	80.00%	26.56%	39.88%	0.3757	0.5417	74.09%	40.01%	51.96%	0.5207	0.5755	33.01%	53.97%	40.96%	0.0527	0.6024	62.50%	40.18%	48.91%	0.4844	0.5189	66.93%	28.84%	40.31%	0.4173	0.4833
	0.5	78.82%	25.87%	38.95%	0.3638	0.5329	74.09%	39.99%	51.94%	0.5206	0.5754	33.01%	53.97%	40.96%	0.0527	0.6024	61.11%	37.93%	46.81%	0.4638	0.5010	67.61%	28.27%	39.87%	0.4149	0.4837
	0.75	77.65%	25.38%	38.26%	0.3544	0.5251	74.20%	40.12%	52.08%	0.5219	0.5765	33.98%	54.69%	41.92%	0.0629	0.6084	59.72%	38.05%	46.49%	0.4591	0.4948	66.94%	29.76%	41.20%	0.4249	0.4879
1	80.00%	26.12%	39.38%	0.3705	0.5395	74.19%	40.05%	52.02%	0.5214	0.5763	33.33%	54.55%	41.38%	0.0604	0.6061	62.51%	40.24%	48.96%	0.4849	0.5185	63.75%	31.00%	41.72%	0.4237	0.4786	
Claude3	0	56.47%	45.71%	50.53%	0.4546	0.5302	51.60%	75.98%	61.46%	0.6143	0.6472	33.98%	61.40%	43.75%	0.1411	0.6420	47.22%	59.65%	52.71%	0.5194	0.5422	53.55%	18.31%	27.29%	0.2825	0.3655
	0.25	56.47%	44.86%	50.00%	0.4488	0.5260	51.17%	77.29%	61.58%	0.6174	0.6517	32.04%	66.00%	43.14%	0.1812	0.6601	47.22%	68.00%	55.74%	0.5571	0.5839	55.21%	18.49%	27.70%	0.2890	0.3745
	0.5	56.47%	43.64%	49.23%	0.4403	0.5198	51.49%	77.03%	61.73%	0.6182	0.6520	33.01%	64.15%	43.59%	0.1666	0.6533	47.22%	66.77%	55.28%	0.5512	0.5772	53.08%	18.49%	27.43%	0.2829	0.3641
	0.75	55.29%	43.52%	48.70%	0.4343	0.5139	51.92%	76.57%	61.88%	0.6188	0.6517	33.01%	65.38%	43.87%	0.1788	0.6594	48.61%	64.81%	55.56%	0.5511	0.5747	51.04%	19.46%	28.18%	0.2860	0.3590
1	57.57%	45.24%	50.67%	0.4563	0.5329	51.28%	76.92%	61.54%	0.6164	0.6504	32.04%	70.21%	44.00%	0.2198	0.6812	46.96%	64.12%	54.22%	0.5384	0.5633	49.08%	19.46%	27.72%	0.2800	0.3495	
GPT-3.5	0	48.24%	51.90%	50.00%	0.4536	0.5264	33.90%	72.11%	46.12%	0.4817	0.5428	35.92%	75.31%	48.68%	0.2850	0.7174	37.50%	65.85%	47.79%	0.4870	0.5260	39.39%	26.82%	31.92%	0.3028	0.3391
	0.25	49.41%	59.15%	53.85%	0.5003	0.5653	32.94%	72.88%	45.37%	0.4775	0.5420	33.98%	67.31%	45.16%	0.2011	0.6715	37.50%	62.79%	46.96%	0.4748	0.5107	41.03%	26.30%	32.05%	0.3057	0.3445
	0.5	49.41%	56.76%	52.83%	0.4872	0.5533	33.05%	73.29%	45.55%	0.4797	0.5446	33.98%	72.92%	46.36%	0.2526	0.6995	37.50%	69.23%	48.65%	0.5002	0.5429	41.03%	25.53%	31.48%	0.3004	0.3407
	0.75	49.41%	53.16%	51.22%	0.4670	0.5353	33.58%	72.75%	45.95%	0.4817	0.5444	34.95%	80.00%	48.65%	0.3172	0.7374	38.89%	75.68%	51.38%	0.5342	0.5819	39.08%	26.05%	31.26%	0.2964	0.3338
1	50.79%	58.18%	54.23%	0.5025	0.5667	33.08%	72.03%	45.34%	0.4754	0.5384	33.01%	59.65%	42.50%	0.1194	0.6308	38.96%	63.29%	48.23%	0.4861	0.5303	39.88%	26.32%	31.71%	0.3013	0.3390	
GPT-4	0	70.59%	54.55%	61.54%	0.5785	0.6387	56.61%	49.08%	52.57%	0.5067	0.5368	46.31%	54.62%	49.79%	0.0822	0.6139	39.72%	53.75%	56.58%	0.5541	0.5733	33.92%	31.46%	32.64%	0.3076	0.3357
	0.25	71.76%	58.55%	64.35%	0.6109	0.6646	56.82%	49.08%	52.67%	0.5078	0.5378	45.34%	53.18%	48.51%	0.0663	0.6043	38.33%	52.50%	55.26%	0.5405	0.5603	35.70%	32.44%	33.99%	0.3214	0.3492
	0.5	70.59%	56.60%	62.83%	0.5921	0.6490	57.14%	49.40%	52.99%	0.5111	0.5410	44.37%	51.79%	47.26%	0.0377	0.5949	38.33%	54.55%	56.38%	0.5518	0.5705	37.19%	33.10%	35.02%	0.3320	0.3598
	0.75	71.76%	55.45%	62.56%	0.5900	0.6486	56.72%	48.99%	52.57%	0.5067	0.5369	45.34%	53.51%	48.72%	0.0727	0.6059	38.33%	52.50%	55.26%	0.5405	0.5603	37.57%	33.77%	35.57%	0.3376	0.3650
1	70.00%	56.75%	62.68%	0.5903	0.6471	56.81%	49.01%	52.63%	0.5073	0.5374	44.36%	54.09%	48.70%	0.0952	0.6064	60.39%	49.95%	54.68%	0.5357	0.5576	39.13%	32.47%	35.49%	0.3372	0.3661	
Llama3-7b	0	40.00%	35.42%	37.57%	0.3115	0.4037	53.30%	45.13%	48.88%	0.4683	0.5011	39.81%	62.12%	48.52%	0.1665	0.6601	45.83%	44.00%	44.90%	0.4337	0.4627	39.52%	13.15%	19.74%	0.1922	0.2714
	0.25	42.35%	39.13%	40.68%	0.3468	0.4330	52.99%	44.86%	48.58%	0.4652	0.4983	38.83%	61.54%	47.26%	0.1567	0.6548	45.83%	46.48%	46.15%	0.4469	0.4696	39.52%	13.85%	20.51%	0.1993	0.2749
	0.5	41.18%	39.33%	40.23%	0.3427	0.4286	52.77%	44.84%	48.48%	0.4641	0.4971	39.81%	60.29%	47.95%	0.1445	0.6510	44.44%	44.44%	44.44%	0.4293	0.4527	38.37%	13.57%	20.05%	0.1934	0.2679
	0.75	41.18%	38.04%	39.55%	0.3344	0.4222	52.77%	44.88%	48.51%	0.4643	0.4973	38.83%	57.14%	46.24%	0.1025	0.6328	45.83%	42.31%	44.00%	0.4244	0.4467	39.56%	12.93%	19.49%	0.1900	0.2705
1	42.85%	37.50%	39.99%	0.3382	0.4271	53.12%	44.74%	48.57%	0.4651	0.4983	38.23%	60.93%	46.98%	0.1478	0.6502	44.73%	47.22%	46.18%	0.4478	0.4704	39.96%	12.51%	18.82%	0.1844	0.2693	
Llama3-13b	0	56.47%	31.58%	40.51%	0.3468	0.4596	51.71%	50.31%	51.00%	0.4901	0.5194	40.78%	55.26%	46.93%	0.0805	0.6283	47.22%	53.97%	50.37%	0.4922	0.5137	64.45%	12.52%	20.66%	0.2441	0.2893
	0.25	54.12%	31.51%	39.83%	0.3375	0.4485	51.49%	50.21%	50.84%	0.4885	0.5178	38.83%	63.49%	48.19%	0.1791	0.6645	47.22%	50.75%	48.92%	0.4761	0.4976	61.97%	12.27%	20.49%	0.2353	0.2763
	0.5	55.29%	31.33%	40.00%	0.3403	0.4530	51.71%	50.21%	50.95%	0.4896	0.5189	39.81%	60.29%	47.95%	0.1445	0.6510	45.83%	49.25%	47.48%	0.4613	0.4834	59.02%	12.52%	20.66%	0.2321	0.2632
	0.75	55.29%	30.92%	39.66%	0.3367	0.4509	52.13%	50.52%	51.31%	0.4934	0.5223	39.81%	57.75%	47.13%	0.1124	0.6382	47.22%	50.00%	48.57%	0.4723	0.4939	57.87%	13.18%	21.47%	0.2578	0.2608
1	54.12%	32.17%	40.35%	0.3432	0.4518	51.63%	50.48%	51.04%	0.4907	0.5199	39.22%	63.49%	48.43%	0.1804	0.6655	48.32%	48.71%	48.52%	0.4712	0.4928	57.87%	12.92%	21.13%	0.2545	0.2695	
Falcon	0	62.50%	26.50%	37.19%	0.3184	0.4610	43.71%	36.25%	39.63%	0.3716	0.4106	37.86%	40.21%	39.00%	-0.1848	0.5457	45.83%	38.82%	42.04%	0.4047	0.4313	14.27%	11.28%	12.60%	0.1000	0.1392
	0.25	63.53%	27.14%	38.03%	0.3289	0.4695	44.55%	36.59%	40.10%	0.3765	0.4154	35.92%	41.57%	38.54%	-0.1470	0.5477	45.83%	37.50%	41.25%	0.3969	0.4247	14.42%	11.39%	12.73%	0.1013	0.1404
	0.5	63.53%	26.73%	37.63%	0.3247	0.4675	43.82%	36.31%	39.71%	0.3724	0.4114	36.99%	40.43%	38.58%	-0.1754	0.5444	47.22%	40.00%	43.31%	0.4178	0.4449	13.86%	11.75%	12.72%	0.1017	0.1395
	0.75	63.53%	26.87%	37.76%	0.3261	0.4682	43.92%	36.30%	39.75%	0.3728	0.4119	36.89%	38.38%	37.62%	-0.2235	0.5342	45.83%	37.93%	41.51%	0.3994	0.4268	14.29%	11.40%	12.69%	0.1009	0.1399
1	63.53%	27.41%	38.29%	0.3317	0.4709	44.77%	36.83%	40.41%	0.3798	0.4186	35.29%	43.75%	38.92%	-0.1998	0.5551	47.86%	35.87%	41.01%	0.3959	0.4264	14.58%	11.29%	12.73%	0.1012	0.1407	
CodeLlama	0	3.53%	33.33%	6.38%	0.0838	0.2271	13.43%	8.79%	10.63%	0.0642	0.1278	15.53%	16.84%	16.16%	-0.6135	0.3730	13.89%	15.15%	14.49%	0.1227	0.1579	12.53%	8.46%	10.20%	0.0739	0.1131
	0.25	3.53%	17.65%	5.88%	0.0415	0.1487	14.61%	9.46%	11.48%	0.0733	0.1368	13.59%	14.43%	14.00%	-0.6710	0.3561	11.11%	16.00%	13.11%	0.1137	0.1487	12.24%	8.90%	10.31%	0.0756	0.1174
	0.5	3.53%	16.67%	5.83%	0.0379	0.1438	14.29%	9.31%	11.27%	0.0711	0.1345	13.59%	15.22%	13.46%	-0.6249	0.3601	12.50%	16.07%	14.06%	0.1211	0.1558	12.62%	8.90%	10.56%	0.0782	0.1201
	0.75	3.53%	21.43%	6.06%	0.0538	0.1676	13.75%	8.98%	10.87%	0.0668	0.1303	14.56%	16.48%	15.46%	-0.5963	0.3688	13.89%	16.13%	14.93%	0.1282	0.1628	12.25%	8.91%	10.31%	0.0756	0.1175
1	4.71%	23.33%	7.84%	0.0693	0.1835	14.75%	9.55%	11.59%	0.0745	0.1379	14.12%	15.6														

Table 12. Impact of Top-P on the LLM performance of vulnerability classification (with VSP)

LLM	Top-p	VulChecker Dataset					BigVul-CVEFixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full				
		MacF1	MicroF1	MCC	MacAu	MicroAu	MacF1	MicroF1	MCC	MacAu	MicroAu	MacF1	MicroF1	MCC	MacAu	MicroAu	MacF1	MicroF1	MCC	MacAu	MicroAu	MacF1	MicroF1	MCC	MacAu	MicroAu
Gemini	0	27.55%	31.58%	0.269	0.3954	0.5215	39.14%	47.16%	0.3524	0.4497	0.5870	21.89%	26.95%	0.1970	0.2950	0.5479	27.70%	35.22%	0.2173	0.3552	0.5024	4.34%	23.83%	0.1665	0.1378	0.4518
	0.25	27.66%	31.14%	0.2998	0.4010	0.5289	38.52%	46.61%	0.3574	0.4560	0.5953	21.71%	27.11%	0.1998	0.2991	0.5556	28.04%	34.67%	0.2203	0.3602	0.5095	4.29%	23.55%	0.1688	0.1397	0.4582
	0.5	27.66%	31.78%	0.2056	0.3929	0.5182	39.31%	47.46%	0.3502	0.4469	0.5833	21.54%	27.01%	0.1958	0.2931	0.5445	27.76%	35.38%	0.2159	0.3529	0.4993	4.22%	23.50%	0.1654	0.1369	0.4489
	0.75	27.71%	31.71%	0.2060	0.3938	0.5194	39.15%	47.36%	0.3509	0.4478	0.5846	21.28%	27.17%	0.1962	0.2938	0.5457	27.93%	34.82%	0.2164	0.3537	0.5004	4.26%	23.46%	0.1658	0.1372	0.4499
Claude3	0	28.28%	31.40%	0.2081	0.3976	0.5245	39.47%	47.27%	0.3544	0.4523	0.5903	21.37%	27.06%	0.1982	0.2967	0.5510	28.44%	35.24%	0.2228	0.3515	0.4670	4.20%	23.32%	0.1674	0.1386	0.4544
	0	30.04%	31.34%	0.2232	0.4458	0.5962	31.95%	37.58%	0.2223	0.3515	0.4686	36.05%	38.96%	0.2777	0.4160	0.5851	26.42%	35.36%	0.2189	0.3579	0.5063	3.49%	14.62%	0.1062	0.0631	0.4228
	0.25	29.69%	31.09%	0.2294	0.4494	0.6010	32.14%	37.06%	0.2241	0.3543	0.4723	36.72%	39.76%	0.2862	0.5103	0.5945	26.63%	35.50%	0.2207	0.3608	0.5103	3.51%	14.57%	0.1071	0.0636	0.4262
	0.5	29.93%	31.59%	0.2262	0.4646	0.6288	32.14%	37.36%	0.2205	0.3487	0.4649	36.94%	39.76%	0.2817	0.5023	0.5851	26.11%	34.94%	0.2172	0.3551	0.5023	3.44%	14.71%	0.1054	0.0626	0.4195
GPT-3.5	0	29.93%	31.66%	0.2257	0.4645	0.6145	31.64%	37.36%	0.2201	0.3479	0.4638	36.94%	40.24%	0.3385	0.4733	0.6234	26.00%	34.87%	0.2167	0.3543	0.5011	3.50%	14.92%	0.1051	0.0624	0.4185
	0.25	29.93%	32.04%	0.2230	0.4580	0.5549	31.45%	36.92%	0.2174	0.3438	0.4583	37.16%	40.16%	0.3345	0.4677	0.6160	25.95%	34.87%	0.2142	0.3501	0.4952	3.45%	14.95%	0.1039	0.0617	0.4136
	0.5	35.89%	38.71%	0.3246	0.4864	0.6840	27.32%	28.27%	0.1712	0.3660	0.4839	14.70%	19.19%	0.0766	0.2973	0.4642	30.97%	39.63%	0.3046	0.4719	0.6817	3.82%	17.53%	0.1242	0.0737	0.4943
	0.75	35.61%	38.63%	0.3219	0.4823	0.6782	27.65%	28.21%	0.1759	0.5031	0.5774	14.70%	19.46%	0.0759	0.2849	0.4602	31.34%	39.32%	0.3020	0.4678	0.6759	3.79%	17.36%	0.1231	0.0731	0.4901
GPT-4	0	36.11%	38.02%	0.2691	0.4032	0.5670	27.32%	27.66%	0.1471	0.4206	0.4828	14.76%	19.46%	0.0635	0.2382	0.3848	30.73%	39.08%	0.2525	0.3912	0.5651	3.81%	17.02%	0.1029	0.0611	0.4098
	0.25	36.33%	37.79%	0.1732	0.4753	0.6684	26.89%	27.66%	0.1733	0.4958	0.5691	14.56%	19.15%	0.0748	0.2808	0.4536	30.30%	39.40%	0.2976	0.4611	0.6661	3.86%	16.99%	0.1213	0.0720	0.4030
	0.5	35.69%	37.42%	0.2782	0.4168	0.5862	27.44%	28.05%	0.1520	0.4348	0.4991	14.27%	19.23%	0.0656	0.2462	0.3978	30.30%	39.16%	0.2610	0.4044	0.5842	3.93%	16.89%	0.1064	0.0632	0.4236
	0.75	45.42%	44.16%	0.3209	0.5273	0.5893	33.06%	35.88%	0.2353	0.4834	0.5853	37.14%	39.02%	0.2931	0.5226	0.6087	30.91%	39.68%	0.2689	0.4166	0.6019	3.44%	20.99%	0.1731	0.1433	0.4699
Llama-2-7b	0	44.81%	43.55%	0.3566	0.5860	0.6549	32.87%	35.53%	0.2615	0.5371	0.6507	36.77%	38.33%	0.3257	0.5807	0.6765	30.85%	39.44%	0.2989	0.4630	0.6889	5.48%	20.99%	0.1924	0.1593	0.5222
	0.25	44.28%	43.29%	0.3407	0.5006	0.5595	33.00%	35.32%	0.2234	0.4589	0.5559	36.12%	39.64%	0.2721	0.4077	0.5734	30.48%	39.44%	0.2553	0.3955	0.5714	5.56%	20.78%	0.1444	0.1361	0.4461
	0.5	44.46%	42.86%	0.3070	0.4752	0.5634	33.20%	34.83%	0.2251	0.4624	0.5601	35.84%	38.85%	0.2742	0.4108	0.5777	30.48%	40.25%	0.2573	0.3985	0.5758	5.55%	20.95%	0.1656	0.1371	0.4495
	0.75	44.46%	42.10%	0.3159	0.5190	0.5801	33.00%	34.97%	0.2316	0.4758	0.5764	35.98%	39.04%	0.2821	0.4227	0.5945	30.73%	39.53%	0.2647	0.4101	0.5925	5.59%	20.95%	0.1704	0.1411	0.4625
Llama-2-13b	0	20.78%	20.79%	0.0638	0.2698	0.3550	21.65%	22.86%	0.1002	0.2783	0.4001	18.34%	18.63%	0.0684	0.2745	0.4177	26.76%	30.56%	0.2092	0.3998	0.5274	1.01%	4.38%	0.0319	0.0163	0.0943
	0.25	21.20%	20.55%	0.0656	0.2773	0.3649	21.91%	23.28%	0.1030	0.2861	0.4113	18.16%	18.41%	0.0703	0.2821	0.4293	26.24%	31.19%	0.2150	0.4110	0.5421	0.99%	4.32%	0.0328	0.0168	0.0969
	0.5	20.87%	20.14%	0.0708	0.2994	0.3940	22.22%	23.24%	0.1113	0.3089	0.4441	18.20%	18.75%	0.0759	0.3046	0.4635	26.88%	30.70%	0.2322	0.4438	0.5854	0.97%	4.36%	0.0354	0.0181	0.1046
	0.75	20.64%	20.30%	0.0695	0.2940	0.3868	21.87%	22.83%	0.1092	0.3033	0.4360	17.84%	18.86%	0.0746	0.2991	0.4551	26.56%	31.32%	0.2280	0.4357	0.5747	0.99%	4.43%	0.0348	0.0178	0.1027
Falcon	0	20.58%	20.51%	0.0631	0.2669	0.3512	22.27%	22.60%	0.0992	0.2753	0.3958	18.13%	18.86%	0.0677	0.2715	0.4132	26.88%	30.77%	0.2070	0.3956	0.5218	1.00%	4.37%	0.0316	0.0161	0.0933
	0	31.82%	36.98%	0.2493	0.3945	0.5259	29.50%	35.36%	0.2505	0.3952	0.5251	26.05%	30.77%	0.2340	0.4471	0.5898	27.89%	31.25%	0.2340	0.4471	0.5898	1.09%	4.97%	0.0266	0.0217	0.0808
	0.25	32.08%	37.65%	0.2570	0.4063	0.5416	30.11%	34.73%	0.2580	0.4070	0.5409	26.05%	30.40%	0.1692	0.3618	0.4784	28.34%	31.76%	0.2410	0.4605	0.6075	1.08%	4.87%	0.0274	0.0224	0.0832
	0.5	32.73%	38.27%	0.2827	0.4379	0.6327	30.72%	34.87%	0.2421	0.3821	0.5077	26.53%	30.71%	0.2262	0.4323	0.5702	28.17%	31.20%	0.2262	0.4323	0.5702	1.07%	4.80%	0.0258	0.0210	0.0781
Codellama	0	32.16%	38.27%	0.2592	0.4015	0.5801	31.28%	34.80%	0.2268	0.4659	0.5644	26.96%	31.08%	0.2074	0.3964	0.5229	27.62%	38.83%	0.2074	0.3964	0.5229	1.08%	4.73%	0.0236	0.0192	0.0716
	0.25	32.48%	38.19%	0.2770	0.4291	0.6199	31.16%	34.94%	0.2273	0.3744	0.4975	27.12%	31.02%	0.2216	0.4236	0.5587	27.79%	31.14%	0.2216	0.4236	0.5587	1.08%	4.74%	0.0252	0.0206	0.0765
	0	30.81%	28.68%	0.2439	0.4872	0.6515	26.30%	24.89%	0.1658	0.4743	0.5443	18.55%	19.15%	0.0745	0.2988	0.4548	10.75%	14.64%	0.0358	0.2476	0.3987	1.31%	1.04%	0.0062	0.0073	0.0541
	0.25	30.69%	29.09%	0.2405	0.4803	0.6423	26.67%	25.20%	0.1635	0.4676	0.5367	18.37%	19.12%	0.0734	0.2946	0.4483	10.84%	14.81%	0.0353	0.2441	0.3931	1.29%	1.05%	0.0061	0.0072	0.0539
Qwen-72b	0	30.44%	29.09%	0.2405	0.4803	0.6423	27.16%	24.85%	0.1635	0.4676	0.5367	18.04%	19.23%	0.0734	0.2946	0.4483	10.80%	14.99%	0.0353	0.2441	0.3931	1.29%	1.03%	0.0061	0.0072	0.0539
	0.25	30.26%	28.97%	0.2415	0.4823	0.6449	26.84%	25.00%	0.1642	0.4695	0.5389	18.12%	19.19%	0.0738	0.2959	0.4502	10.93%	14.70%	0.0354	0.2451	0.3947	1.31%	1.04%	0.0061	0.0072	0.0541
	0.5	30.20%	28.57%	0.2349	0.4891	0.6540	27.22%	24.80%	0.1665	0.4761	0.5464	18.30%	19.04%	0.0748	0.3000	0.4565	10.71%	15.00%	0.0359	0.2486	0.4002	1.32%	1.06%	0.0062	0.0073	0.0549
	0.75	32.50%	35.27%	0.1687	0.4827	0.6540	19.84%	21.67%	0.0767	0.2990	0.3933	20.98%	23.19%	0.1111	0.3084	0.4414	16.51%	21.23%	0.0758	0.3041	0.4628	0.45%	2.29%	0.0198	0.0062	0.0666
Qwen-72b	0	26.72%	25.23%	0.1525	0.4362	0.5007	20.24%	21.76%	0.1004	0.2787	0.4007	20.90%	22.78%	0.1004	0.2787	0.4007	16.51%	20.97%	0.0685	0.2749	0.4183	0.45%	2.27%	0.0507	0.0067	0.0603
	0.25	26.72%	25.23%	0.1525	0.4362	0.5007	20.24%	21.76%	0.1004	0.2787	0.4007	20.90%	22.78%	0.1004	0.2787	0.4007	16.51%	20.97%	0.0685	0.2749	0.4183	0.45%	2.27%	0.0507	0.0067	0.0603
	0.5	26.72%	25.23%	0.1525	0.4362	0.5007	20.24%	21.76%	0.1004	0.2787	0.4007	20.90%	22.78%	0.1004	0.2787	0										

Table 13. Impact of Top-P on the LLM performance of vulnerability repair (with VSP)

LLM	Top-P	ExtractFix Dataset					BigVul-CVEFixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full																									
		Valid	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct																				
Gemini	0	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	1.27%	13.14%	2.91%	5.83%	6.80%	15.53%	3.77%	1.89%	9.43%	15.09%	0.20%	4.16%	7.90%	12.26%	0.25	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	1.27%	13.14%	2.91%	5.83%	6.80%	15.53%	3.77%	1.89%	9.43%	15.09%	0.20%	4.16%	7.90%	12.26%	
	0.5	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	0.85%	12.71%	2.91%	5.83%	6.80%	15.53%	5.66%	1.89%	9.43%	16.98%	0.42%	3.94%	7.90%	12.26%	0.75	29.17%	12.50%	12.50%	4.17%	0.00%	29.17%	6.36%	5.51%	0.42%	12.29%	2.91%	4.85%	6.80%	14.56%	5.66%	1.89%	9.43%	16.98%	0.42%	3.94%	7.90%	12.26%	
	0.75	70.83%	29.17%	20.83%	20.83%	0.00%	70.83%	7.20%	9.32%	6.36%	22.88%	11.65%	10.80%	8.74%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.27%	13.40%	1	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.51%	1.27%	13.14%	2.91%	5.83%	6.80%	15.53%	3.77%	3.77%	11.32%	18.87%	0.42%	3.94%	7.90%	12.26%	
	1	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.32%	6.36%	22.88%	11.65%	10.80%	8.74%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.27%	13.40%	0	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	15.09%	30.19%	0.67%	15.39%	13.16%	29.22%	
	0.25	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.75%	6.36%	23.31%	12.62%	9.71%	7.77%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.27%	13.62%	0.5	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.75%	6.36%	23.31%	12.62%	9.71%	7.77%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.27%	13.62%	
0.75	70.83%	29.17%	20.83%	20.83%	0.00%	70.83%	7.20%	9.32%	6.36%	22.88%	13.59%	9.71%	7.77%	31.07%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.04%	13.40%	1	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.32%	6.36%	22.88%	11.65%	10.80%	8.74%	31.07%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.04%	13.40%		
Claude3	0	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	15.09%	30.19%	0.67%	15.39%	13.16%	29.22%	0.25	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	15.09%	30.19%	0.67%	15.39%	13.16%	29.22%	
	0.5	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	15.09%	30.19%	0.67%	15.39%	13.16%	29.22%	0.75	83.33%	25.00%	37.50%	20.83%	0.00%	83.33%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	11.32%	28.30%	0.45%	15.39%	13.38%	29.22%	
	0.75	83.33%	25.00%	33.33%	20.83%	0.00%	83.33%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	16.98%	32.08%	0.25%	15.39%	13.38%	29.22%	1	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.32%	8.90%	12.29%	30.51%	14.56%	4.85%	23.30%	42.72%	7.55%	7.55%	16.98%	32.08%	0.45%	15.39%	13.38%	29.22%	
	1	79.17%	20.83%	41.67%	16.67%	0.00%	79.17%	17.02%	12.77%	19.15%	48.93%	16.50%	5.83%	20.39%	42.72%	7.55%	9.43%	20.75%	37.74%	1.84%	6.91%	22.07%	30.82%	0.25	79.17%	20.83%	41.67%	16.67%	0.00%	79.17%	17.02%	12.77%	19.15%	48.93%	16.50%	5.83%	20.39%	42.72%	7.55%	9.43%	20.75%	37.74%	1.84%	7.13%	22.07%	31.04%	
	0.5	79.17%	20.83%	41.67%	16.67%	0.00%	79.17%	17.02%	12.77%	19.15%	48.93%	17.48%	5.83%	20.39%	43.69%	7.55%	9.43%	20.75%	37.74%	1.84%	7.36%	22.07%	31.26%	0.75	79.17%	25.00%	41.67%	12.50%	0.00%	79.17%	17.02%	12.77%	19.15%	48.93%	17.48%	5.83%	21.36%	44.66%	7.55%	9.43%	20.75%	37.74%	1.84%	7.36%	22.07%	31.26%	
1	83.33%	25.00%	41.67%	16.67%	0.00%	83.33%	17.02%	12.77%	19.15%	48.93%	16.50%	6.80%	20.39%	43.69%	7.55%	9.43%	20.75%	37.74%	1.84%	7.36%	22.07%	31.26%	Llama2-7b	0	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	0.85%	0.00%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%	
0.25	20.83%	12.50%	4.17%	4.17%	0.00%	20.83%	2.12%	0.85%	0.00%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%		0.5	20.83%	12.50%	4.17%	4.17%	0.00%	20.83%	2.12%	0.85%	0.00%	2.97%	0.00%	0.94%	1.94%	3.88%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%	
0.75	20.83%	12.50%	4.17%	4.17%	0.00%	20.83%	1.69%	0.85%	0.42%	2.97%	0.00%	0.94%	1.94%	3.88%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%		1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	0.85%	0.00%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%	
1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	2.12%	0.85%	0.00%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	0.00%	5.66%	7.55%	1.84%	0.00%	5.06%	6.90%		0.25	20.83%	12.50%	4.17%	4.17%	0.00%	20.83%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	3.77%	7.55%	0.00%	1.27%	6.85%	8.12%	
0.5	20.83%	12.50%	4.17%	4.17%	0.00%	20.83%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	3.77%	7.55%	0.00%	1.27%	6.85%	8.12%		0.75	25.00%	12.50%	8.33%	4.17%	0.00%	25.00%	2.12%	1.27%	0.42%	3.81%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.85%	8.12%	
1	25.00%	12.50%	8.33%	4.17%	0.00%	25.00%	2.12%	1.27%	0.85%	4.24%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	5.66%	9.43%	0.00%	1.27%	6.85%	7.90%	CodeLlama	0	20.83%	8.33%	8.33%	4.17%	0.00%	20.83%	0.00%	1.27%	1.69%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.10%	3.78%	
0.25	20.83%	8.33%	8.33%	4.17%	0.00%	20.83%	0.00%	1.27%	1.69%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.10%	3.78%		0.5	20.83%	8.33%	8.33%	4.17%	0.00%	20.83%	0.00%	1.27%	1.69%	2.97%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%	
0.75	20.83%	8.33%	8.33%	4.17%	0.00%	20.83%	0.42%	1.27%	1.69%	3.39%	0.00%	0.97%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%		1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	4.66%	1.94%	1.94%	2.91%	1.89%	1.89%	1.89%	5.66%	0.00%	1.69%	2.32%	4.01%	
1	16.67%	8.33%	4.17%	4.17%	0.00%	16.67%	0.00%	0.85%	1.69%	2.54%	4.66%	1.94%	1.94%	1.94%	5.83%	3.77%	3.77%	1.89%	7.55%	0.22%	2.21%	3.49%		5.93%	0.25	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.12%	4.24%	1.94%	1.94%	5.83%	3.77%	3.77%	1.89%	9.43%	0.22%	2.43%	3.49%	6.15%	
0.5	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	1.94%	1.94%	1.94%	5.83%	3.77%	3.77%	1.89%	7.55%	0.22%	2.43%	3.49%	6.15%		0.75	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	1.94%	1.94%	5.83%	3.77%	3.77%	1.89%	7.55%	0.00%	2.43%	3.49%	5.93%		
1	16.67%	4.17%	4.17%	8.33%	0.00%	16.67%	0.85%	1.27%	2.54%	4.66%	0.97%	1.94%	1.94%	4.85%	1.89%	3.77%	1.88%	7.55%	0.00%	2.21%	3.49%	5.70%	DSC-V2	0	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	1.94%	1.94%	4.85%	1.89%	1.89%	1.89%	5.66%	0.44%	1.40%	2.05%	3.88%	
0.25	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	0.85%	0.85%	1.69%	3.39%	0.97%	1.94%	1.94%	4.85%	0.00%	1.89%	1.89%	3.77%	7.55%	0.44%	1.40%	2.05%		3.88%	0.5	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	0.85%	0.85%	1.69%	3.39%	0.97%	2.91%	1.94%	5.83%	0.00%	3.77%	1.89%	5.66%	0.44%	1.62%	2.05%	4.10%
0.75	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	0.85%	0.85%	2.12%	3.81%	0.97%	2.91%	1.94%	5.83%	0.00%	3.77%	1.89%	5.66%	0.44%	1.85%	2.05%	4.33%		1	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	1.94%	1.94%	3.88%	1.89%	1.89%	1.89%	5.66%	0.21%	1.85%	2.05%	4.10%	
1	12.50%	4.17%	4.17%	4.17%	0.00%	12.50%	1.27%	0.85%	2.12%	4.24%	0.97%	1.94%	1.94%	3.88%	1.89%	1.89%	1.89%	5.66%	0.21%	1.85%	2.05%	4.10%		FT-Llama2	0	29.17%	0.00%	20.83%	8.33%	0.00%	29.17%	0.00%	14.41%	15.6													

Table 15. Impact of system prompt (SP) on the LLM performance of vulnerability classification (with VSP). NA: No System Prompt; HA: Helpful Assistant; SE: Security Expert

LLM	SP	VulChecker Dataset				BigVul+CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full								
		MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu	MacF1	MicF1	MCC	MacAu	MicAu					
Gemini	NA	28.28%	31.40%	0.2081	0.3976	0.5245	39.47%	47.27%	0.3544	0.4523	0.5903	21.37%	27.06%	0.1982	0.2967	0.5510	28.44%	35.24%	0.2228	0.3515	0.4670	4.20%	23.32%	0.1674	0.1386	0.4544
	HA	28.79%	31.97%	0.2043	0.3906	0.5152	39.71%	47.74%	0.3481	0.4442	0.5798	21.37%	27.28%	0.1946	0.2914	0.5412	28.33%	35.10%	0.2188	0.3452	0.4587	4.14%	23.08%	0.1644	0.1361	0.4463
	SE	28.21%	32.09%	0.2036	0.3891	0.5132	39.23%	48.51%	0.3465	0.4425	0.5777	21.46%	27.71%	0.1939	0.2903	0.5392	28.84%	34.54%	0.2180	0.3439	0.4570	4.15%	22.67%	0.1638	0.1356	0.4446
Claude3	NA	29.93%	32.04%	0.2230	0.4580	0.5549	31.45%	36.92%	0.2174	0.3438	0.4583	37.16%	40.16%	0.3345	0.4677	0.6160	25.95%	34.87%	0.2142	0.3501	0.4952	3.85%	14.95%	0.1039	0.0617	0.4136
	HA	29.99%	32.23%	0.2217	0.4553	0.5516	31.45%	37.07%	0.2162	0.3418	0.4556	36.42%	40.72%	0.3325	0.4650	0.6123	26.47%	34.73%	0.2129	0.3480	0.4923	3.47%	15.07%	0.1033	0.0613	0.4111
	SE	30.53%	32.49%	0.2199	0.4517	0.5472	31.26%	37.22%	0.2144	0.3390	0.4520	36.20%	41.46%	0.3298	0.4612	0.6074	26.52%	35.36%	0.2112	0.3452	0.4883	3.43%	15.34%	0.1024	0.0608	0.4078
GPT-3.5	NA	33.69%	37.42%	0.2782	0.4168	0.5862	27.44%	28.05%	0.1520	0.4348	0.4991	14.27%	19.23%	0.0656	0.2462	0.3978	30.30%	39.16%	0.2610	0.4044	0.5842	3.99%	16.89%	0.1064	0.0732	0.4236
	HA	36.05%	37.42%	0.3189	0.4778	0.6720	27.60%	28.33%	0.1743	0.4985	0.5721	14.53%	18.92%	0.0752	0.2823	0.4560	30.73%	38.61%	0.2992	0.4636	0.6697	3.99%	16.65%	0.1220	0.0724	0.4856
	SE	36.19%	36.90%	0.3063	0.4589	0.6454	28.05%	28.10%	0.1674	0.4788	0.5495	14.76%	18.96%	0.0723	0.2711	0.4380	30.11%	38.15%	0.2452	0.3877	0.5169	3.91%	16.42%	0.1172	0.0626	0.4664
GPT-4	NA	44.46%	42.10%	0.3159	0.5190	0.5801	33.00%	34.97%	0.2316	0.5764	0.5764	35.98%	39.04%	0.2821	0.4227	0.5945	30.73%	39.53%	0.2647	0.4101	0.5925	5.99%	20.95%	0.1704	0.1411	0.4625
	HA	45.62%	42.95%	0.3446	0.5661	0.6327	32.60%	35.04%	0.2327	0.5190	0.6287	36.63%	40.22%	0.3790	0.5299	0.6979	31.47%	40.33%	0.2888	0.4473	0.6462	5.60%	20.53%	0.1859	0.1552	0.5045
	SE	46.08%	43.72%	0.3644	0.5987	0.6691	33.00%	34.42%	0.2672	0.5488	0.6648	36.19%	39.66%	0.3254	0.4876	0.6857	31.34%	40.17%	0.3054	0.4730	0.6854	5.67%	20.61%	0.1966	0.1627	0.5335
Llama2-7b	NA	20.58%	20.51%	0.0631	0.2669	0.3512	22.27%	22.60%	0.0992	0.2753	0.3958	18.13%	18.86%	0.0677	0.2715	0.4132	26.88%	30.77%	0.2070	0.3956	0.5218	1.00%	4.37%	0.0316	0.0161	0.0933
	HA	20.83%	20.92%	0.0639	0.2703	0.3557	22.67%	23.10%	0.1004	0.2789	0.4009	18.57%	19.09%	0.0685	0.2750	0.4185	26.83%	31.07%	0.2096	0.4006	0.5285	1.02%	4.73%	0.0320	0.0164	0.0945
	SE	21.12%	20.96%	0.0667	0.2820	0.3710	22.76%	23.24%	0.1048	0.2909	0.4182	18.92%	19.43%	0.0839	0.2907	0.3982	27.20%	31.13%	0.2187	0.4179	0.5513	1.01%	4.42%	0.0334	0.0171	0.0985
Llama2-13b	NA	32.48%	38.19%	0.2760	0.4291	0.6199	31.16%	34.94%	0.2373	0.3744	0.4975	27.12%	31.02%	0.2216	0.4236	0.5587	27.79%	31.14%	0.2212	0.4236	0.5587	1.08%	4.74%	0.0252	0.0206	0.0765
	HA	33.13%	38.88%	0.2940	0.4554	0.6579	31.66%	35.36%	0.2518	0.3973	0.5280	27.45%	31.14%	0.2352	0.4495	0.5930	28.12%	31.76%	0.2352	0.4495	0.5930	1.09%	4.82%	0.0268	0.0218	0.0812
	SE	32.73%	39.58%	0.2618	0.4055	0.5858	32.04%	35.78%	0.2290	0.4704	0.5699	27.88%	32.01%	0.2094	0.4002	0.5280	28.23%	32.19%	0.2094	0.4002	0.5280	1.08%	4.79%	0.0239	0.0194	0.0723
Falcon	NA	30.20%	28.57%	0.2449	0.4891	0.6540	27.22%	24.80%	0.1665	0.4761	0.5464	18.30%	19.94%	0.0748	0.3000	0.4565	10.71%	15.00%	0.0359	0.2486	0.4002	1.32%	1.06%	0.0062	0.0073	0.0549
	HA	30.80%	28.91%	0.2420	0.4833	0.6463	27.44%	25.00%	0.1645	0.4705	0.5400	18.41%	19.23%	0.0739	0.2965	0.4511	10.88%	15.06%	0.0353	0.2457	0.3955	1.34%	1.04%	0.0064	0.0072	0.0542
	SE	30.88%	29.32%	0.2386	0.4765	0.6372	27.93%	25.10%	0.1622	0.4639	0.5328	18.65%	19.38%	0.0855	0.2962	0.4057	10.98%	15.21%	0.0350	0.2422	0.3900	1.32%	1.04%	0.0060	0.0071	0.0535
CodeLlama	NA	25.79%	25.69%	0.1409	0.3013	0.3984	21.08%	21.68%	0.0961	0.2669	0.3837	20.99%	22.88%	0.0961	0.2669	0.3837	16.12%	21.28%	0.0656	0.2632	0.4006	0.46%	2.30%	0.0171	0.0125	0.0619
	HA	26.09%	25.94%	0.1510	0.3228	0.4268	21.78%	22.02%	0.1030	0.2860	0.4111	21.09%	23.47%	0.1030	0.2860	0.4111	16.50%	21.96%	0.0703	0.2820	0.4291	0.64%	2.28%	0.0184	0.0134	0.0570
	SE	26.25%	25.89%	0.1458	0.4170	0.4787	21.82%	22.07%	0.0960	0.2665	0.3831	21.12%	23.56%	0.0960	0.2665	0.3831	16.77%	22.36%	0.0655	0.2628	0.3999	0.64%	2.26%	0.0054	0.0064	0.0481
QC-2.5	NA	23.45%	22.82%	0.1120	0.2832	0.4071	21.87%	22.32%	0.1020	0.2832	0.4071	20.57%	21.82%	0.1020	0.2832	0.4071	16.41%	20.31%	0.0696	0.2793	0.4250	1.28%	6.74%	0.0326	0.0309	0.0818
	HA	23.92%	23.43%	0.1082	0.3004	0.4318	22.00%	22.46%	0.1082	0.3004	0.4318	20.65%	21.99%	0.1082	0.3004	0.4318	16.67%	20.63%	0.0738	0.2962	0.4508	1.28%	6.77%	0.0346	0.0328	0.0868
	SE	24.06%	23.86%	0.1116	0.3098	0.4454	22.35%	22.57%	0.1116	0.3098	0.4454	20.80%	22.34%	0.1116	0.3098	0.4454	16.77%	20.84%	0.0762	0.3055	0.4649	1.29%	6.86%	0.0357	0.0338	0.0895
DSC-V2	NA	27.57%	26.67%	0.1736	0.4966	0.5700	29.07%	29.84%	0.2354	0.5102	0.6822	27.04%	27.93%	0.1736	0.4966	0.5700	24.49%	27.27%	0.1675	0.3382	0.4736	3.05%	7.63%	0.0126	0.0085	0.0959
	HA	27.77%	27.09%	0.1755	0.5021	0.5763	29.36%	30.02%	0.2382	0.5158	0.6897	27.15%	27.99%	0.1755	0.5021	0.5763	24.78%	27.93%	0.1694	0.3621	0.4788	3.02%	7.54%	0.0127	0.0111	0.0902
	SE	27.95%	27.47%	0.1731	0.4951	0.5683	29.59%	30.26%	0.2547	0.5086	0.6802	27.42%	28.09%	0.1731	0.4951	0.5683	25.18%	28.27%	0.1670	0.3571	0.4722	2.97%	7.56%	0.0126	0.0378	0.0947
FT-Llama2	NA	27.16%	33.54%	0.2124	0.4059	0.5354	25.54%	35.53%	0.2230	0.3646	0.5158	18.62%	32.02%	0.1961	0.2741	0.5242	19.91%	38.93%	0.2207	0.3183	0.4637	8.96%	17.13%	0.0667	0.2504	0.4045
	HA	27.59%	33.34%	0.2409	0.4605	0.6074	25.08%	35.25%	0.1672	0.3471	0.4704	18.25%	32.08%	0.2224	0.3110	0.5947	19.95%	39.63%	0.2504	0.3611	0.5261	8.80%	17.00%	0.0757	0.2841	0.4589
	SE	27.53%	33.40%	0.2066	0.3948	0.5208	25.18%	35.18%	0.2170	0.3547	0.5017	18.25%	31.96%	0.1907	0.2666	0.5099	19.55%	39.86%	0.2147	0.3096	0.4511	8.90%	17.54%	0.0649	0.2436	0.3935

Table 16. Impact of system prompt (SP) on the LLM performance of vulnerability repair (with VSP). NA: No System Prompt; HA: Helpful Assistant; SE: Security Expert

LLM	SP	ExtractFix Dataset					BigVul+CVEFixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full					
		Valid	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Unres	Correct
Gemini	NA	33.33%	12.50%	12.50%	8.33%	0.00%	33.33%	6.36%	5.93%	1.69%	13.98%	2.91%	4.85%	7.77%	15.53%	3.77%	5.66%	11.32%	20.75%	0.42%	3.94%	7.90%	12.26%	0.42%	3.94%	7.90%	12.26%
	HA	37.50%	12.50%	16.67%	8.33%	0.00%	37.50%	6.36%	5.93%	1.69%	13.98%	2.91%	4.85%	7.77%	15.53%	3.77%	5.66%	11.32%	20.75%	0.42%	3.94%	7.90%	12.26%	0.42%	3.94%	7.90%	12.26%
	SE	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.32%	6.36%	22.88%	11.65%	9.71%	8.74%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.68%	6.04%	13.40%	3.68%	3.45%	6.27%	13.40%
Claude3	NA	66.67%	25.00%	20.83%	20.83%	0.00%	66.67%	7.20%	9.75%	6.36%	23.31%	12.62%	9.71%	7.77%	30.10%	3.77%	5.66%	9.43%	18.87%	3.68%	3.45%	6.27%	13.40%	3.68%	3.23%	6.27%	13.17%
	HA	79.17%	25.00%	33.33%	20.83%	0.00%	79.17%	9.75%	8.47%	12.29%	30.51%	14.56%	5.83%	23.30%	43.69%	7.55%	7.55%	16.98%	32.08%	0.22%	1						

PrimeVul-Subset dataset increases from 0.5329 (No System Prompt) to 0.5468 with the 'Security Expert' persona, while AuPRC improves from 0.5546 to 0.5668.

Table 15 details the effect of system prompts on vulnerability classification. Similar to the detection task, the "Security Expert" persona provides benefits for specific models but lacks universal effectiveness. Claude3, for instance, exhibits a clear improvement on the Zero-Day dataset, rising from 40.16% Micro F1 without a system prompt to 41.46% with the "Security Expert" persona. Conversely, GPT-3.5 shows a slight decline on the same dataset (19.23% vs. 18.96%) when the persona is applied. The fine-tuned Llama2-13b model demonstrates remarkable stability, with its Zero-Day performance remaining virtually unchanged (32.02% vs. 31.96%). This indicates that because the fine-tuned model has already learned exactly how to perform the task during its training, adding extra instructions to ask the LLM to 'act like an expert' does not improve its performance.

```

1 int isoent_gen_joliet_identifier(struct archive_write *a, struct isoent *isoent, struct idr *idr){
2   struct iso9660 *iso9660;
3   struct isoent *np;
4   unsigned char *p;
5   size_t l;
6   int r;
7   int fmax, parent_len;
8   .....
9   r = idr_start(a, idr, isoent->children.cnt, fmax, 6, 2, &rb_ops);
10  for (np = isoent->children.first; np != NULL; np = np->chnext){
11    unsigned int ext_off, noff, weight;
12    .....
13    if ((int)l == fmax)
14      noff = ext_off + 6; // Calculation of noff has potential CWE-190 vulnerability
15    else if ((int)l == fmax - 2)
16      noff = ext_off + 4;
17    else if ((int)l == fmax - 4)
18      noff = ext_off + 2;
19    else
20      noff = ext_off;
21    idr_register(idr, np, weight, noff);
22  }
23  .....
24 }

```

A: Step 1: Find vulnerability semantics:  
The function has inputs "a", "isoent", and "idr". At line 17, the function starts an "idr" with a count of "isoent->children.cnt", a maximum size of "fmax", and other parameters. Then, at lines 23-26, a loop iterates over the children of "isoent" and performs various operations on them. Therefore, the vulnerability semantics is:

```

1 int isoent_gen_joliet_identifier(struct archive_write *a, struct isoent *isoent, struct idr *idr)
2   r = idr_start(a, idr, isoent->children.cnt, fmax, 6, 2, &rb_ops);
3   for (np = isoent->children.first; np != NULL; np = np->chnext)
4     idr_register(idr, np, weight, noff);

```

Step 2: Vulnerability semantics analysis:  
At line 17, the "idr" is started with a count of "isoent->children.cnt" and a maximum size of "fmax". However, the code does not check whether "isoent->children.cnt" is greater than "fmax", which may cause a heap-based buffer overflow (CWE-122) vulnerability.  
Conclusion: the code has a CWE-122 vulnerability.

Fig. 11. GPT-3.5 identifies an incorrect vulnerability cause on a vulnerability classification sample.

Expert" persona, increasing its correct rate from 42.72% to 45.63%. This indicates that for some models, persona-based guidance can be beneficial when tackling novel repair scenarios. Conversely, for the fine-tuned Llama2-13b model, system prompts can be slightly detrimental, causing a minor performance regression on datasets like BigVul+CVExFixes (30.08% vs. 29.24%).

Table 10 indicates that while adding a persona can lead to statistically distinguishable results—particularly for vulnerability detection ( $p < 0.001$ )—the magnitude of this change is limited. The effect sizes for all pairwise comparisons (No Prompt vs. Helpful Assistant, No Prompt vs. Security Expert, and Helpful Assistant vs. Security Expert) are negligible across detection, classification, and repair tasks ( $|d| < 0.05$ ). For classification specifically, the statistical significance varies by metric, further suggesting that system prompts do not fundamentally shift the performance baseline established by the VSP strategy.

**Finding 9** (RQ3): System prompts offer modest gains for general-purpose models on novel data, but often show negligible impact or slight regression for code-specific and fine-tuned models.

Table 16 presents the impact of system prompts on vulnerability repair. For this complex generative task, we observe that system prompts have a very limited and inconsistent effect across most models. High-performing commercial models like GPT-4 and Claude3 demonstrate remarkable stability, with their correct rates remaining almost entirely unchanged regardless of the persona assigned. This suggests that their repair capabilities are deeply ingrained and not easily influenced by high-level instructions. A notable exception is GPT-3.5, which shows a tangible improvement on the Zero-Day dataset when primed with the "Security

## 5 RQ4: LLMs VS. TRADITIONAL APPROACHES

Tables 17, 18, and 19 compare LLMs to traditional approaches on the three tasks. The best key results are marked in boldface. VulChecker, ExtractFix, and VulnFix require input programs to be compilable whereas the BigVul+CVEFixes, Zero-Day, and PrimeVul samples are not. Thus, corresponding results are marked as none (-).

```

1 char *string_crypt(const char *key, const char *salt) {
2     assertx(key);
3     assertx(salt);
4     char random_salt[12];
5     if (!*salt) {
6         memcpy(random_salt, "$1$", 3);
7         ito64(random_salt+3, rand(), 8);
8         random_salt[11] = '\0';
9         return string_crypt(key, random_salt);
10    }
11    ...
56}

```

A: Step 1: Find vulnerability semantics:  
The function has an input "ptr". "ptr" is first checked not to be NULL at line 6. If so, the members of "ptr", "m1" and "m2", are freed at lines 7-8. Finally, the pointer "ptr" itself is freed at line 9. Therefore, the vulnerability semantics is:

```

1 void func(struct st *ptr)
6   if (ptr){
7     free(ptr->m1);
8     free(ptr->m2);
9     free(ptr);
10  }
11  b=1;
12  int c;
13  c = a+b;
14 }

```

A: Step 2: Vulnerability semantics analysis:  
At line 6, "ptr" is first checked not to be NULL.  
At lines 7-8, the members are freed before the pointer "ptr" is freed. Therefore, the code has a double free (CWE-415) vulnerability.  
Conclusion: the code has a vulnerability.

Fig. 12. Falcon generates unrelated outputs during vulnerability detection.

With **existing prompting strategies**, the LLMs are close (e.g., 48.64% versus 48.92% on BigVul+ CVEFixes) or superior (e.g., 54.55% versus 52.31%, 47.06% versus 43.79%, 50.75% versus 13.04%, and 33.98% versus 9.56% on the VulChecker, Zero-Day, PrimeVul-Subset, and PrimeVul-Full datasets, respectively) to the best traditional approaches (CodeQL, FlawFinder, and Devign) in terms of F1 for vulnerability detection. This indicates the limitations of traditional static analyzers and the advantages of LLMs for vulnerability analysis. Figure 9 shows an example: FlawFinder does not conduct comprehensive checks on the whole program but directly reports the potential vulnerability based on its predefined patterns, causing the false positive. We notice that two of the DL-based vulnerability detectors, Devign and LineVul, achieve lower F1 compared to the one reported in their original papers. The reason is that their original testing sets have the overfitting issue, which inflates their effectiveness [68, 71].

Retraining the DL-based models on DiverseVul generally improves their performance compared to the pre-trained versions on our datasets. For example, Devign (Retrained) achieves an F1 score of 48.98% on BigVul+CVEFixes, up from 42.01% in its pre-trained state. For vulnerability classification, the merits of LLMs are 4.88–18.04 points or by 13.5%–110% greater than the best baseline (VulChecker and FlawFinder) against the VulChecker, BigVul+CVEFixes, and Zero-Day datasets in micro F1. For PrimeVul-Subset and PrimeVul-Full, all three traditional tools fail completely with 0% accuracy. However, for the repair task, the best baseline repair performance (83.33%, from the tool ExtractFix) is higher than LLMs (70.83%) on the ExtractFix datasets. VulRepair achieves relatively high correct rate against BigVul+CVEFixes, due to its DL model overfitting to the training data—it works better on the BigVul+CVEFixes dataset which overlaps with its training set [29]. When we retrain VulRepair on DiverseVul (removing the overlap), its correct rate drops to 17.37% on BigVul+CVEFixes, but it achieves higher correct rates on other datasets.

**Finding 10 (RQ4):** LLMs with the best existing prompting strategies perform similarly to or better than the best traditional approaches on vulnerability detection, classification, and repair.

With **VSP strategies**, however, some of the top-performing LLMs achieve higher F1 scores than the evaluated traditional approaches across the three tasks. For example, 62.68/61.54/48.70/54.68/41.72% vs. 52.31/48.92/43.79/13.04/9.06% for detection, 42.10/47.27/40.16/39.53/23.32% vs. 36.23/ 16.40/12.02/0.00/0.00% for classification, and 83.33/32.37/30.25/37.74/31.26% vs. 83.33/30.50/ 0.00/0.00/0.00% for repair. For detection and classification, on which best existing strategies already suffice to make LLMs edge over them, VSP pushes up the margins. VSP also enables certain LLMs (e.g., GPT-4) to match

Table 17. Comparison to traditional approaches on vulnerability detection

Approach	VulChecker Dataset				BigVul-CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full									
	Recall	Precision	F1	MCC AuPRC	Recall	Precision	F1	MCC AuPRC	Recall	Precision	F1	MCC AuPRC	Recall	Precision	F1	MCC AuPRC	Recall	Precision	F1	MCC AuPRC						
best LLM w/ existing	60.00%	50.00%	54.55%	0.4993	0.5677	40.91%	60.00%	48.64%	0.4792	0.5159	47.06%	47.06%	47.06%	-0.0588	0.6030	45.29%	57.72%	50.75%	0.4996	0.5231	60.69%	23.59%	33.98%	0.3526	0.4266	
best LLM with VSP	70.00%	56.75%	62.68%	0.5903	0.6471	51.28%	76.92%	61.54%	0.6164	0.6504	44.36%	54.09%	48.70%	0.0822	0.6564	60.39%	49.95%	54.68%	0.5357	0.5576	63.75%	31.00%	41.72%	0.4237	0.4786	
DeVign (Pretrained)	26.80%	30.95%	28.72%	0.2240	0.3212	28.09%	83.33%	42.01%	0.4737	0.5709	49.33%	39.36%	43.79%	-0.2757	0.5701	3.22%	38.59%	5.95%	0.1053	0.2233	5.06%	21.48%	8.19%	0.0929	0.1453	
DeVign (Retrained)	28.94%	31.57%	30.20%	0.2375	0.3341	39.61%	64.16%	48.98%	0.4681	0.5456	51.80%	39.36%	44.73%	0.3898	0.4772	11.40%	25.47%	15.75%	0.1187	0.2236	10.98%	15.04%	12.69%	0.0571	0.1696	
LineVul (Pretrained)	1.18%	50.00%	2.30%	0.0663	0.2997	5.65%	92.98%	10.65%	0.2243	0.5113	3.88%	50.00%	3.88%	0.0000	0.5097	26.39%	5.67%	9.34%	0.0703	0.1712	21.68%	5.73%	9.06%	0.0638	0.1475	
LineVul (Retrained)	1.36%	58.00%	2.65%	0.0788	0.3405	7.85%	61.37%	13.92%	0.1978	0.3870	4.07%	57.50%	7.61%	0.1360	0.3504	26.92%	4.54%	7.76%	-0.1608	0.1897	25.80%	3.84%	6.68%	-0.2151	0.1811	
VulChecker	95.29%	15.76%	27.04%	0.2666	0.5373	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CppCheck	1.18%	50.00%	2.30%	0.0663	0.2997	4.05%	77.55%	7.70%	0.1719	0.4265	7.77%	50.00%	13.45%	0.0000	0.5194	2.78%	1.43%	1.89%	-0.0178	0.0354	1.74%	5.28%	2.62%	0.0153	0.0482	
FlawFinder	40.00%	75.56%	52.31%	0.5207	0.6044	26.65%	98.04%	41.91%	0.5035	0.6376	21.36%	55.00%	30.77%	0.0491	0.5784	23.61%	6.85%	10.63%	0.0827	0.1636	3.91%	6.85%	4.98%	0.0323	0.0666	
CodeQL	14.12%	85.71%	24.24%	0.3291	0.5372	33.69%	89.27%	48.92%	0.5391	0.6276	17.48%	47.37%	25.53%	-0.0250	0.5306	69.44%	7.19%	13.04%	0.1655	0.3877	49.54%	5.29%	9.56%	0.0941	0.2809	

Table 18. Comparison to traditional approaches on vulnerability classification

Approach	VulChecker Dataset					BigVul-CVEFixes					Zero-Day					PrimeVul-Subset					PrimeVul-Full				
	Macro F1	Micro F1	MCC	MacAu	MicAu	Macro F1	Micro F1	MCC	MacAu	MicAu	Macro F1	Micro F1	MCC	MacAu	MicAu	Macro F1	Micro F1	MCC	MacAu	MicAu	Macro F1	Micro F1	MCC	MacAu	MicAu
best LLM w/ existing	38.28%	41.11%	0.3125	0.4957	0.5528	26.91%	34.44%	0.2213	0.4644	0.5579	19.57%	31.90%	0.2045	0.2858	0.5466	30.33%	38.92%	0.2570	0.3391	0.5699	5.03%	18.76%	0.1129	0.0670	0.4485
best LLM with VSP	44.46%	42.10%	0.3133	0.5147	0.5752	39.47%	47.27%	0.3544	0.4523	0.5903	37.16%	40.16%	0.3345	0.4677	0.6160	30.73%	39.53%	0.2625	0.4066	0.5875	5.59%	20.95%	0.1690	0.1399	0.4587
VulChecker	30.98%	36.23%	0.2474	0.4057	0.5131	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CppCheck	0.00%	0.00%	0.0329	0.0350	0.0257	1.22%	1.89%	0.0282	0.0180	0.0698	1.48%	1.90%	0.0269	0.0185	0.0718	0.00%	0.00%	0.0118	0.0308	0.0052	0.00%	0.00%	0.0231	0.0008	0.0314
FlawFinder	8.15%	12.16%	0.0349	0.2416	0.3890	12.01%	16.40%	0.0452	0.2541	0.3281	19.16%	12.02%	0.0332	0.2299	0.3701	0.00%	0.00%	0.0206	0.0096	0.0180	0.00%	0.00%	0.0378	0.0088	0.0040
CodeQL	2.50%	2.29%	0.0321	0.0304	0.0805	0.49%	0.42%	0.0059	0.0070	0.0523	0.00%	0.00%	0.0201	0.0353	0.0312	0.00%	0.00%	0.0131	0.0208	0.0466	0.00%	0.00%	-0.0152	0.0343	0.0347

Table 19. Comparison to traditional approaches on vulnerability repair

Approach	ExtractFix Dataset						BigVul-CVEFixes				Zero-Day				PrimeVul-Subset				PrimeVul-Full			
	Valid	SynEq	SemEq	Plaus	Unres	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct	SynEq	SemEq	Plaus	Correct
best LLM w/ existing	54.17%	8.33%	37.50%	8.33%	0.00%	54.17%	9.30%	10.47%	13.95%	33.72%	19.42%	6.80%	14.56%	40.78%	0.00%	18.87%	15.09%	33.96%	0.00%	6.33%	14.35%	20.04%
best LLM with VSP	83.33%	25.00%	41.67%	16.67%	0.00%	83.33%	17.02%	12.77%	19.15%	48.93%	16.50%	6.80%	20.39%	43.69%	0.00%	9.43%	28.30%	37.74%	1.84%	7.36%	22.07%	31.26%
VulRepair (Pretrained)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	21.19%	4.24%	2.54%	27.96%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
VulRepair (Retrained)	25.00%	4.17%	4.17%	16.67%	0.00%	25.00%	4.66%	4.66%	8.05%	17.37%	0.97%	2.91%	7.77%	11.65%	2.78%	4.17%	13.89%	20.83%	5.29%	5.75%	6.21%	17.24%
ExtractFix	83.33%	20.83%	33.33%	29.16%	0.00%	83.33%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
VulnFix	45.83%	20.83%	12.50%	12.50%	0.00%	45.83%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

the performance of ExtractFix, the most effective repair baseline, even though this tool takes advantage of tests to facilitate repair, which LLMs do not use/need. VSP also helps LLMs get closer to the best traditional approach on any of the three tasks.

**Finding 11 (RQ4):** VSP enables certain LLMs to outperform or match the best traditional approaches across the evaluated tasks. For vulnerability detection, some VSP-enhanced LLMs (e.g., GPT-4) surpass the traditional approaches on the unbiased Zero-Day dataset, with larger margins observed on other datasets. For classification and repair, some VSP-enhanced LLMs also surpass the traditional approaches.

## 6 RQ5/RQ6: CORRECTNESS OF REASONING

In addition to correct (final) *answers*, correct *reasoning* also matters as it indicates better certainty hence gives users more trust of the answers [87]. We assess LLMs’ reasoning correctness by examining whether their correct answers are reasonable (i.e., associated with correct reasoning). We then look at how VSP can improve LLMs in this regard too.

### 6.1 Quantitative Analysis of the Reasoning Correctness

We first conduct a quantitative analysis for the correctness of the reasoning. We manually examine the LLM-generated reasoning in each response. Given the manual nature and study scale, for each task and each LLM on each prompting strategy, we randomly sample the LLM responses for a sample size greater than the one statistically significant at 95% confidence level and 5% margin of error with respect to the population (i.e., total #responses by the LLM with the strategy)—466, 377, 247 for detection, classification, and repair, respectively.

We evaluate whether the generated reasoning captures the vulnerability semantics and justifies the outcome correctly. Since LLMs only generate final answers with standard few-shot and RAG (as the exemplars do), we only evaluate the other strategies. We notice that, on all the true-negative predictions, the reasoning is correct, and the reasoning for

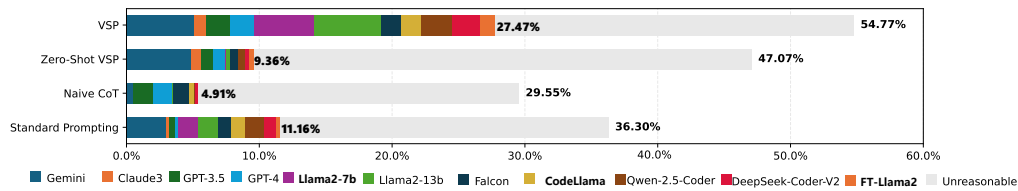


Fig. 13. Answer vs. reasoning correctness in vulnerability detection.

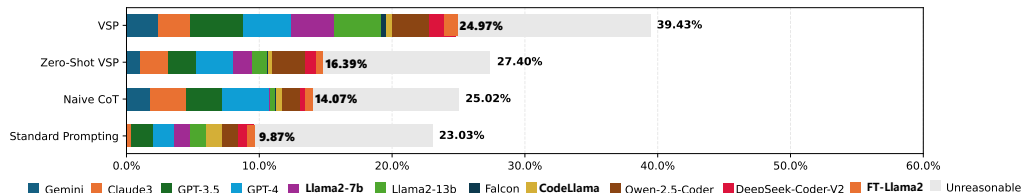


Fig. 14. Answer vs. reasoning correctness in vulnerability classification.

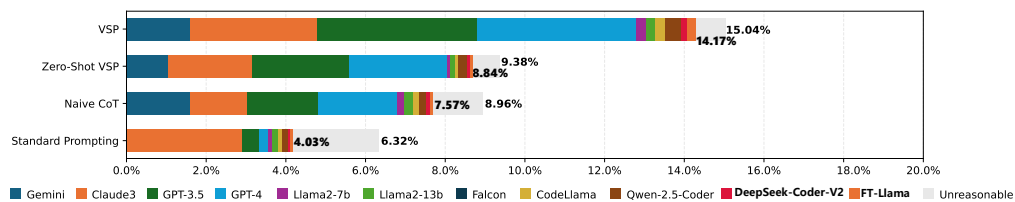


Fig. 15. Answer vs. reasoning correctness in vulnerability repair.

incorrect responses is consistently incorrect. Thus, we only show the results on the true-positive cases, as depicted in Figures 13, 14, and 15: the total bar length indicates the ratio of responses with correct answers (e.g., 29.55% on naive CoT in Fig. 13), while the solid-stacked-bar length indicates the ratio of responses with correct reasoning (e.g., 4.91%), for all the LLMs' respective responses combined on each strategy. Notably, for any LLM and strategy, *reasoning for incorrect answers is consistently incorrect*. We also note that the purpose of Figures 13, 14, and 15 is to illustrate the overall reasoning correctness for each prompting strategy, rather than to compare the reasoning correctness across different LLMs. We randomly sampled the correct prediction cases from all model outputs. The figures do not show the ratio of correct answers for each individual LLM.

With **existing prompting strategies**, the LLMs have up to 11.16%, 14.07%, and 7.57% of their responses backed by correct reasoning, although the ratios of correct answers are up to 36.30%, 25.02%, and 8.96%, for detecting, classifying, and repairing vulnerabilities, respectively. The ratio of correct reasoning for correct answers is  $\frac{11.16\%}{36.30\%} = 30.74\%$  on the detection task; similarly, the ratio on classification and repair is 56.23% and 84.49%, respectively. That is, around 70% of the correct detection outcomes are not justified by correct reasoning, but less (<50%) for correct CWE prediction, and only a few (around 15%) for correct repair, which can be explained by the fact that the more challenging tasks require correct reasoning to provide correct answers. For instance, while it is possible to *guess* correctly in detection and classification tasks, a correct repair requires LLMs to correctly analyze the code before generating the correct patches. However, there are still a few correct patches with incorrect reasoning; we found such cases happen when the patches are very simple and can be directly borrowed from seen training samples (e.g., change `<=` to `<`).

Our results also reveal that the LLMs have widely varying reasoning correctness for different tasks and with different prompting strategies. For instance, with standard prompting, Gemini has overall notably greater correctness than all the

other LLMs for detection, where Claude3 performs the best in this regard for repair. We notice that Llama2 has much less correct reasoning. This indicates that it is necessary to have more powerful LLMs for vulnerability patching. Overall, code-specific LLMs (CodeLlama, Qwen-2.5-Coder, and DeepSeek-Coder-V2) have less correct reasoning compared to general LLMs (e.g., Gemini), especially on the task of vulnerability repair, indicating that code-specific LLMs are not good at code semantic understanding and data/control flow analysis. Gemini generates patches without providing reasoning with standard prompting for classification and repair, thus the respective numbers are not shown. Naive CoT improves over standard prompting for both classification and repair, but not for detection.

**Finding 12 (RQ5):** The evaluated LLMs frequently suffer incorrect reasoning, even for correct answers (71.1% has incorrect reasoning on vulnerability detection, 46.4% on classification, and 14.5% on repair).

With **VSP strategies**, the LLMs have their reasoning correctness improved, both individually and collectively, raising the correctness ratios to 27.47%, 24.97%, and 14.17% for detection, classification, and repair (from 11.16%, 14.07% and 7.57% with existing strategies). Behind the *correct* answers, the reasoning correctness ratios are improved by VSP to  $\frac{27.47\%}{54.77\%}=50.15\%$  for vulnerability detection, and 63.33% and 94.22% for classification and repair, respectively. Without using exemplars, zero-shot VSP also boosts the LLMs' reasoning for classification and repair (from 14.07% to 16.39% and 7.57% to 8.84%), but not for detection (9.36% versus 11.16%).

**Finding 13 (RQ6):** VSP improves the LLMs in correct reasoning, especially for correct answers, by up to 63.14%, 12.62%, and 11.51% for vulnerability detection, classification, and repair, respectively.

## 6.2 Qualitative Analysis of the Incorrect Reasoning

While our quantitative results show that VSP improves overall prediction accuracy and reasoning correctness for some of the LLMs, we further conduct deeper qualitative analysis revealing specific types of reasoning errors that persist, even when VSP is applied. We manually examined a subset of LLM responses, focusing on cases where the reasoning was incorrect. As a result, we classify them into the following categories.

**Vague Interpretation of Code Semantics.** A prevalent type of incorrect reasoning stems from the LLMs' vague interpretation of code semantics, even when guided by VSP. This often makes the LLMs ignore the vulnerable code and directly output unreliable conclusions. For example, in Figure 10, the actual vulnerability is a CWE-121 at line 68. This occurs because the `d` pointer and `ds` length can be manipulated by the preceding while loops such that `d[0]` or `d[1]` accesses memory beyond the bounds of the `d_orig` buffer. However, GPT-3.5, despite using VSP, incorrectly concludes that "the code does not have any exploitable vulnerabilities." Its reasoning is vague, just going through the code like without actual vulnerability analysis, completely missing the critical data flow related to `d` and `ds` and their impact on the array access.

We observe that vague interpretation is not monolithic; rather, it manifests through several recurring reasoning patterns across different models and samples. In many cases, the LLM's reasoning remains at the level of functional description, accurately restating what the code does line by line, yet failing to transition from what happens to why it is security-relevant. Variables influenced by external inputs, memory accesses, or boundary-sensitive operations are described neutrally rather than being identified as potential sources of vulnerability.

A common manifestation of vagueness is the model's failure to prioritize security-critical elements within the code. Although multiple variables, loops, or memory operations may be present, the LLM often treats them as equally important, rather than identifying a small subset of security-relevant statements (e.g., buffer writes, pointer dereferences, or loop bounds derived from external input) that dominate the vulnerability semantics.

In some instances, the reasoning appears to rely on superficial pattern recognition (e.g., detecting the presence of arrays or loops) without reconstructing the underlying data or control dependencies that make a specific operation unsafe. As a result, the model may acknowledge that a construct could be risky, while failing to explain how attacker-controlled inputs propagate to the vulnerable operation.

```

1 gchar *rsvg_io_get_file_path(const gchar *filename, const gchar
2 *base_uri){
3     gchar *absolute_filename;
4     if (g_file_test(filename, G_FILE_TEST_EXISTS) ||
5         g_path_is_absolute(filename))
6         absolute_filename = g_strdup(filename);
7     else{
8         gchar *tmpcdirdir;
9         gchar *base_filename;
10        if (base_uri){
11            base_filename = g_filename_from_uri(base_uri, NULL,
12 NULL);
13        }
14        if (base_filename != NULL){
15            tmpcdirdir = g_path_get_dirname(base_filename);
16            g_free(base_filename); // First free
17        }
18        else
19            return NULL;
20        }
21        else
22            tmpcdirdir = g_get_current_dir();
23        absolute_filename = g_build_filename(tmpcdirdir, filename,
24 NULL);
25        g_free(base_filename); //Second free
26    }
27    return absolute_filename;
}

```

**Answer by GPT-3.5:**  
A: Step 1: Find vulnerability semantics:  
The function has inputs "filename" and "base\_uri". If "filename" exists or is an absolute path, it is duplicated and assigned to "absolute\_filename" at line 6. Otherwise, "tmpcdirdir" is assigned with the directory name of "base\_filename" at line 14. Then, "absolute\_filename" is built with "tmpcdirdir" and "filename" at line 22. Finally, "absolute\_filename" is returned. Therefore, the vulnerability semantics is:  
1 gchar \*rsvg\_io\_get\_file\_path(const gchar \*filename, const gchar \*base\_uri)  
6 absolute\_filename = g\_strdup(filename);  
14 tmpcdirdir = g\_path\_get\_dirname(base\_filename);  
22 absolute\_filename = g\_build\_filename(tmpcdirdir, filename, NULL);  
Step 2: Vulnerability semantics analysis:  
At line 6, "filename" is duplicated and assigned to "absolute\_filename".  
At line 14, "tmpcdirdir" is assigned with the directory name of "base\_filename".  
At line 22, "absolute\_filename" is built with "tmpcdirdir" and "filename".  
There is no vulnerability in this code because there is only one free operation at line 24.  
Conclusion: the code does not have an exploitable vulnerability.

Fig. 16. Incomplete CFA in vulnerability detection by GPT-3.5.

However, GPT-3.5, when prompted with VSP for classification, incorrectly identifies the vulnerability as CWE-122 (Heap-based Buffer Overflow). Its reasoning points to line `r = idr_start(...)` and claims that "the code does not check whether `isoent->children.cnt` is greater than `ffmax`, which may cause a heap-based buffer overflow."

Our analysis shows that misidentified vulnerability causes are not random mistakes, but arise from several recurring patterns in how LLMs attribute causality within code. A frequent cause of misidentification is the model's reliance on surface-level lexical or structural cues (e.g., the presence of arithmetic operations, pointer dereferences, or loop constructs) to infer the vulnerability type. In such cases, the LLM associates these cues with commonly co-occurring CWEs, even when they are not the root cause of the vulnerability. For instance, arithmetic expressions involved in loop bounds may lead the model to predict integer overflow (e.g., CWE-190), while the actual vulnerability arises from missing bounds checks on a buffer indexed by an externally controlled value (e.g., CWE-121/122).

Another recurring pattern is incorrect localization of the vulnerability's root cause. While the model may correctly identify a vulnerable statement (e.g., a buffer write or memory access), it often attributes the vulnerability to that statement in isolation, rather than to earlier statements that introduce unsafe values or violate required invariants. This leads to vulnerability cause misidentified when different vulnerability types share similar sinks but differ fundamentally in how unsafe states are introduced.

We observed these forms of vague interpretation consistently across multiple datasets and models, including cases where the final prediction was correct, indicating that coincidental correctness may mask fundamentally incomplete reasoning. These observations suggest that vague interpretation stems not from an inability to describe code, but from a lack of structured guidance on which semantics matter for security. By explicitly prompting models to approximate vulnerability semantics—i.e., to identify security-relevant statements and reason about their data/control dependencies—VSP reduces the likelihood that reasoning remains purely descriptive and helps shift the model's focus toward the core mechanisms underlying exploitable behavior.

**Misidentified Vulnerability Cause.** Another common error is that LLMs may misidentify an incorrect vulnerability cause within the code. Figure 11 presents an example, which contains a CWE-190 (Integer Overflow) vulnerability. The issue lies at lines 78-82, specifically in calculations like `noff = ext_off + 6;`. If `ext_off` is a very large integer, an integer overflow can occur.

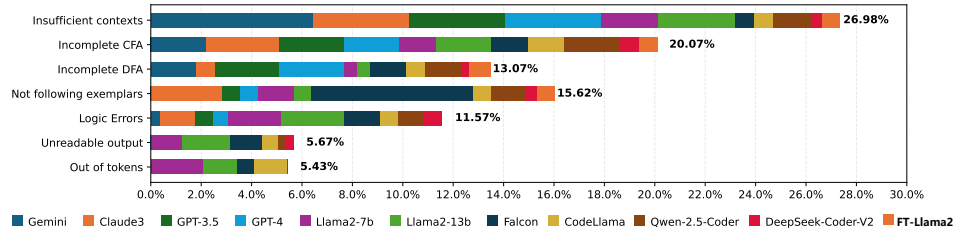


Fig. 17. Failure causes for vulnerability detection.

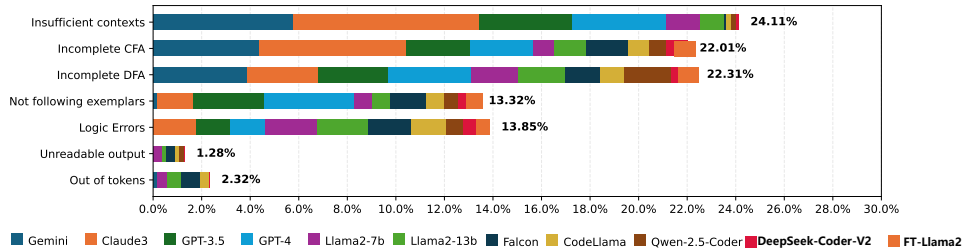


Fig. 18. Failure causes for vulnerability classification.

We also observe systematic confusion among semantically related CWEs, particularly those that share overlapping manifestations but differ in root cause. For example, the model may conflate out-of-bounds access, buffer overflow, and use-after-free vulnerabilities, as they all involve memory safety violations, despite their distinct causal mechanisms. This confusion suggests that the model recognizes the category of unsafe behavior but fails to precisely distinguish the underlying conditions that define each CWE.

Notably, such misattribution of vulnerability cause can occur even when the final CWE prediction happens to be correct, indicating that correct answers may still be supported by flawed or incomplete causal reasoning. These misidentifications are largely attributable to incomplete reconstruction of data and control dependencies that connect external inputs, intermediate computations, and vulnerable operations.

**Unrelated Outputs.** The most severe form of incorrect reasoning observed is unrelated outputs, where the LLM generates reasoning for a completely different, non-existent code snippet or vulnerability, entirely ignoring the provided input code. This renders the LLM’s output completely useless. Figure 12 shows the `string_crypt` function, which contains a subtle CWE-121 (Out-of-bounds write) vulnerability. However, Falcon, when given this code and VSP prompting, generates completely unrelated contents. It generates reasoning for a non-existent function `func(struct st *ptr)` and identifies a CWE-415 (Double Free) vulnerability, which has no relation to the `string_crypt` function provided in the prompt. This extreme failure to follow instructions and analyze the given code, instead generating entirely fabricated content, represents a critical limitation, particularly for smaller or less capable LLMs. Such instances underscore the need for more robust instruction-following and context-adherence mechanisms in LLMs for security-critical tasks.

## 7 RQ7: CHALLENGES

In this section, we investigate the challenges in LLM-based vulnerability analysis, by examining the causes of LLMs’ failures (incorrect answers). In RQ1, we show that VSP is constantly the most effective prompting strategy, while the LLMs’ performance varied across datasets and tasks. Thus, we conduct case analysis on the incorrect predictions of

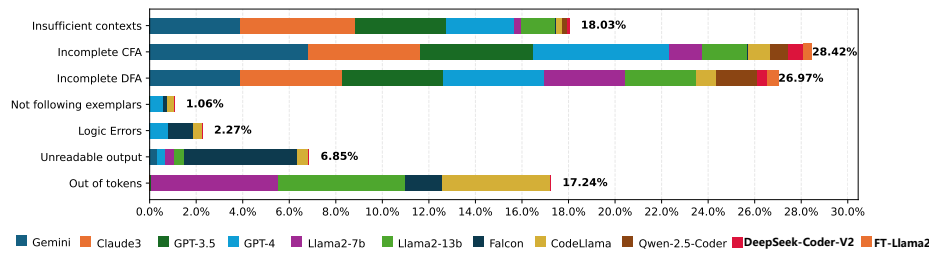


Fig. 19. Failure causes for vulnerability repair.

VSP on all the LLMs and datasets, following a sampling process similar to that for RQ5/RQ6. We ended up inspecting 481, 457, and 387 cases for detection, classification, and repair, respectively. Based on our observations, we classify the failure causes into seven categories, as shown in Figures 17–19.

**Insufficient Contexts.** Because most LLMs have limits on #tokens processed, it is infeasible to feed them with whole projects [75]. Thus, we only input individual functions to the LLMs. However, vulnerabilities can be inter-procedural, which cannot be comprehensively analyzed without scanning contextual code in the project. Figure 23 shows an example where a function can only be used in a safe situation, but GPT-3.5 predicts it to be vulnerable because it does not know the contexts. This is common for almost all the LLMs, with a total of 18.03-26.98% proportions for the three tasks, indicating the dominance of this failure cause. For vulnerability detection, this is common for Gemini, Claude3, 2 GPTs, and 2 Llama2s, with a total of 26.98% on all the inspected failure cases, while the numbers are 24.11% and 18.03% for classification and repair, respectively.

**Incomplete Control Flow Analysis (CFA).** In some cases, LLMs failed due to their inability to fully capture control-flow facts necessary for correct predictions. Figure 16 shows an example with a CWE-415 double free vulnerability, which the LLM fails to detect as it misses a path along which the pointer is freed twice. While VSP has guided CFA reasoning steps and LLMs are capable of such analysis to some extent, they still fall short of very complex and comprehensive analysis. For vulnerability detection and classification, 20.07% and 22.01% of the failure cases are due to this cause, indicating it is a common challenge for LLMs. For vulnerability repair, it is even more common with a 28.42% proportion. This is because repair requires even more accurate CFA.

Across multiple incorrect and partially correct cases, we observe that incomplete CFA does not simply arise from missing individual branches, but from systematic difficulty in composing non-linear execution paths. In particular, LLMs frequently under-approximate reachability when control flow involves nested loops, early exits (e.g., break, continue, or return), or conditionals whose predicates depend on runtime values derived from external inputs. In such cases, the model may correctly reason about a single execution path while failing to recognize alternative paths under which the vulnerable statement becomes reachable.

In addition, indirect control-flow constructs—such as function pointers, callbacks, or virtual dispatch—are often treated as opaque by the model. When the concrete call target is not syntactically explicit, the LLM tends to omit the potential effects of the call from its reasoning, leading to incomplete control-flow reconstruction. These patterns explain why CFA-related errors persist even when the vulnerable statement itself is correctly identified, and why reasoning remains incomplete without explicit guidance on vulnerability-relevant control-flow paths.

**Incomplete Data Flow Analysis (DFA).** Similar to *incomplete CFA*, LLMs failed in some cases because they are not able to perform necessary DFA. As shown in Figure 20 as an example: when identifying a buffer overflow vulnerability, the models may ignore the vulnerable buffer write even if the index may be beyond the boundary of the buffer. This is

a common failure cause with a total of 13.07-26.97% proportions for the three tasks. This indicates, again, that while LLMs are capable to some extent, they may stumble on complex and comprehensive DFA.

Our examination of DFA-related errors reveals that incomplete data-flow reasoning typically stems from the model's inability to track value propagation across multiple program statements and boundaries. LLMs often reason locally about individual assignments or operations, but fail to propagate constraints on externally influenced values through intermediate variables, arithmetic transformations, or conditional checks. As a result, the causal chain connecting an input source to a vulnerable sink is only partially reconstructed.

```

1 CURLcode Curl_urldecode(struct SessionHandle *data,
2                         const char *string, size_t length,
3                         char **ostring, size_t *olen,
4                         bool reject_ctrl){
5     size_t alloc = (length?length:strlen(string))+1;
6     char *ns = malloc(alloc);
7
8     .....
9
10    if(!ns)
11        return CURLE_OUT_OF_MEMORY;
12    while(--alloc > 0) {
13        in = *string;
14        if(('%' == in) && ISXDIGIT(string[1]) && ISXDIGIT(string[2])) {
15            .....
16            ns[stindex++] = in; //The write may be out of bound.
17            string++;
18        }.....
19        ns[stindex]=0; /* terminate it */
20        .....
21    }
22    return CURLE_OK;
23}

```

**Answer by Llama2:**  
A: Step 1: Find vulnerability semantics:  
The function has several inputs: "data", "string", "length", "ostring", "olen", and "reject\_ctrl". The function decodes the URL-encoded string in "string" and stores the decoded string in "ns". The decoded string is then assigned to "ostring" and its length is assigned to "olen". Therefore, the vulnerability semantics is:  
1 CURLcode Curl\_urldecode(struct SessionHandle \*data,  
2 const char \*string, size\_t length,  
3 char \*\*ostring, size\_t \*olen,  
4 bool reject\_ctrl)  
5 char \*ns = malloc(alloc);  
6 while(--alloc > 0){  
7 in = \*string;  
8 if(('%' == in) && ISXDIGIT(string[1]) && ISXDIGIT(string[2])) {  
9 .....  
10 ns[stindex]=0; /\* terminate it \*/  
11 }  
12 .....  
13}
Step 2: Vulnerability semantics analysis:  
1. The function allocates memory for the decoded string using malloc at line 6.  
2. It iterates through the input string and decodes each hexadecimal escape sequence at lines 15-39.  
3. It terminate decoded character to the corresponding Unicode character at line 48.  
Conclusion: the function has no exploitable vulnerabilities.

This limitation is particularly evident in the presence of pointer aliasing and shared mutable state, which are common in C/C++ code. When multiple pointers may reference the same memory location, LLMs frequently assume independence among variables, overlooking aliasing relationships that are essential for understanding how a memory access becomes unsafe. Similarly, inter-procedural data flow poses challenges: when unsafe values are introduced in one function and consumed in another, the model often fails to carry over assumptions about bounds or invariants across function calls. These factors collectively lead to misidentified vulnerability causes or incomplete vulnerability semantics, even when individual unsafe operations are correctly recognized.

**Not Following Exemplars.** The studied LLMs do not follow our VSP exemplars in some cases. Figure 21 shows an example: while we provide the LLMs with exemplars, the LLMs may not always follow the reasoning steps in the exemplars to analyze vulnerabilities. Instead, they act like in Standard Prompting and simply provide the answers with brief analysis. In all the inspected failure

Fig. 20. Incomplete DFA in vulnerability detection by Llama2-13b.

cases of detection, classification, and repair, 15.62%, 13.32%, and 1.06% are in this category. Among different LLMs, this is common with Claude3, GPTs, Llama2s, and Falcon against vulnerability detection and classification. A possible reason is that these LLMs may not have sufficient instruction tuning during the training [5, 6]. In comparison, Gemini follows the exemplars in most of the cases, indicating it is good for CoT prompting.

**Logic Errors.** We notice that the text generated by LLMs can be logically contradictory or make incorrect logical implications during vulnerability analysis. As Figure 22 shows, an LLM may claim that an integer is checked not to be zero, but still claim that the vulnerability exists because the integer can be zero. We notice that this is common on Claude3, Llama2s, and Falcon against vulnerability detection and classification. This indicates that some of the LLMs may not be logical for effective vulnerability analysis.

**Unreadable Output.** Some LLMs output nothing or repeat certain text sequences in some cases. For example, the output can be like "////////////////////.....". This is mostly seen on the three least capable LLMs (Llama2s and Falcon) against the three tasks, indicating that more capable LLMs are necessary for effective vulnerability analysis.

**Out of Token Limit.** When an LLM is producing a result, the output may exceed its token limit, leading to output truncation, even if the reasoning steps and conclusion the model wants to generate are correct. This mostly happens in Llama2s and Falcon across the three tasks due to their smaller token limits (4,096/2,048; Table 1).

**Finding 14 (RQ7):** The LLMs commonly face major challenges with insufficient code contexts and incomplete control/data flow analysis in vulnerability analysis. Small models (Llama2 and Falcon) suffer from an inability to follow exemplars, making VSP much less helpful to them.

## 8 DISCUSSION

### 8.1 Why the Performance of Existing Prompting Strategies Varies

```
// Exemplars
Q: Does the following code have exploitable vulnerabilities? Please
check whether the inputs can be maliciously used for crashing the
program.
Code to analyze ...
A: Step 1: Find vulnerability semantics: .....
Step 2: Vulnerability semantics analysis: .....
..... (Provide a total of 20 exemplars)
```

---

```
// System prompt: Follow the reasoning steps and answers in the
exemplars before the last question.
// Question
Q: Does the following code have exploitable vulnerabilities? Please
check whether the inputs can be maliciously used for crashing the
program.
Code to analyze ...
```

---

```
// Answer by Claude3
After analyzing the code, I did not find any obvious exploitable vulnerabilities that could be
used to crash the program. The code appears to be performing various operations related to
finding the best SAS (Short Authentication String) type based on the offered and configured
algorithms. It handles different cases and checks for mandatory SAS types. The code does
not seem to have any buffer overflows, use-after-free, or other memory corruption
vulnerabilities that could lead to a crash.
```

Fig. 21. Claude3 not following exemplars in vulnerability detection.

reasoning-based strategies frequently fall short for complex vulnerability analysis.

In this case, adapting prompting strategies with explicit reasoning seems promising. However, Naive Chain-of-Thought (CoT) often fails because it focuses on general code functionality rather than the specific code behaviors that constitute vulnerability semantics. For instance, as illustrated in Figure 6, GPT-3.5 with Naive CoT merely summarizes code operations, missing the critical unchecked buffer manipulation that leads to a CWE-122 vulnerability, thus failing to provide correct reasoning or prediction.

Zero-shot completion, primarily applied to vulnerability repair, often produces patches that, while sometimes passing test cases, are functionally unreasonable or introduce new issues, indicating a lack of deep semantic understanding of LLMs with this prompting strategy. Ultimately, the general limitations of existing strategies stem from either insufficient specific guidance for vulnerability semantics or an inability to deeply reason about complex code behaviors, leading to inconsistent and unreliable outcomes.

### 8.2 Why Semantics-Guided Prompting Can Help LLMs Improve

In our study, we show that our semantics-aware VSP strategy improves LLMs for vulnerability analysis of some of the evaluated LLMs. The main reason for the improvement is that it follows the human reasoning for vulnerability analysis and focuses on the parts that are most important for understanding vulnerabilities (i.e., vulnerability semantics). As Figure 6 shows, GPT-3.5 correctly detects a CWE-122 vulnerability with VSP but fails with Standard Prompting and Naive CoT—with which the LLM just simply scans the code and analyzes the major functionality of the code. Without focusing on the vulnerability semantics, it is easy to ignore the unchecked buffer manipulation at line 9. In

In this study, we notice that existing prompting strategies exhibit diverse effectiveness across vulnerability analysis tasks. Standard prompting relies on the LLM’s inherent knowledge, often insufficient for complex vulnerability analysis. Standard few-shot learning, while guided by examples, struggles to generalize effectively due to its dependency on fixed exemplars. Retrieval-Augmented Generation (RAG) attempts to provide context through dynamic retrieval, which can be beneficial for some LLMs, but its success is contingent on the accuracy of the retrieval mechanism. Overall, these non-

contrast, with VSP, GPT-3.5 focuses on vulnerability semantics and notices that the write of `&s->cmdbuf[s->cmdlen]` is unchecked and may have a buffer overflow issue. Figure 8 shows another example for vulnerability repair. With VSP, GPT-3.5 successfully identifies the root cause of the buffer overflow and fixes it while standard prompt and Naive CoT simply add a boundary check without analyzing the vulnerability root cause correctly. Therefore, vulnerability-specific prompting adaptation is crucial for effective analysis.

This comparison with Naive CoT further clarifies that VSP’s effectiveness arises not merely from encouraging structured outputs, but from steering the substance of the model’s reasoning toward vulnerability-relevant code semantics. Since Naive CoT also enforces a step-by-step format but fails to achieve comparable results, we can infer that the benefit lies in the specific focus on dependency-oriented reasoning rather than the template itself. However, this reliance on vulnerability semantics focused guidance also explains why VSP does not uniformly improve performance across all models and datasets. Because the strategy depends on the model’s ability to internalize reasoning patterns from exemplars and apply them to new contexts, its success is bound by the model’s capacity to generalize beyond superficial similarities. When this generalization fails—often due to complex interprocedural flows or missing context—the model may produce structured yet substantively incomplete reasoning, consistent with our observation that VSP mitigates but does not fully eliminate reasoning errors.

### 8.3 Capabilities of LLMs

Our study reveals that the inherent capabilities of LLMs, beyond just prompting strategies, influence their performance in software vulnerability analysis. This is evident in how different models, despite similar prompting, yield varied results. Larger models, such as GPT-4 and Claude3, generally demonstrate a superior capacity to analyze code security. For instance, when tasked with detecting a buffer overflow (like the CWE-122 in Figure 6) that requires tracing data flow and understanding memory operations, a larger LLM is more likely to correctly identify the intricate dependencies and potential exploit paths. In contrast, smaller models, such as Llama2-7b or Falcon, might only catch more obvious, localized issues.

```

A: Step 1: Find vulnerability semantics:
The function has two input "x" and "y". Then, at lines 3-9, the function performs a division
operation. Therefore, the vulnerability semantics is:
1 static int32_t divide(int32_t x, int32_t y){
2     if (y == 0){
3         if (x < 0)
4             return 1;
5         return -1;
6     }
7     return x / y;
8 }
Step 2: Vulnerability semantics analysis:
At line 3-4, the function checks whether "y" is 0. If "y" is 0, the function returns 1 or -1. However,
it is possible that "y" is set to 0 maliciously, which may cause a division by zero vulnerability.
Conclusion: the code has an exploitable vulnerability

```

Fig. 22. Logic error encountered by Llama2-13b.

The architectural choices or training methodologies of the LLMs also play a crucial role for the model effectiveness. General-purpose LLMs (e.g., GPT-4, Claude3), trained on vast and diverse text and code, often exhibit a more robust understanding of abstract reasoning and logical inference. This allows them to effectively explain the underlying vulnerability semantics, even for complex C/C++ memory errors, by drawing on a wider range of

learned patterns. In contrast, code-specific LLMs (e.g., CodeLlama, DeepSeek-Coder-V2), while trained for specific tasks like code generation, may not have sufficiently emphasized the deep semantic understanding critical for identifying subtle security vulnerabilities. Therefore, a general LLM providing a more insightful vulnerability explanation is better than a code-focused one for complex vulnerability analysis.

Finally, our results shed light on the trade-off between fine-tuning and prompt engineering for LLMs’ performance on various vulnerability analysis tasks. While fine-tuning boosts the performance of open-source models (e.g., Llama2-13b) on established datasets by aligning them with specific task formats and patterns, it does not inherently solve the challenge of generalization. This suggests that the reasoning gap in fine-tuned models also need to be bridged by

providing explicit vulnerability semantics focused guidance via prompting rather than relying solely on weight updates from historical data, which may not cover novel vulnerability manifestations.

#### 8.4 Impact of Model Configurations

Beyond the external prompting strategies like VSP, our study reveals that the internal configuration of LLMs—specifically decoding strategies and system-level instructions—plays a pivotal role in the reliability of vulnerability analysis across all three tasks. Our investigation highlights the delicate balance between model creativity and the rigorous precision required for security engineering.

Regarding temperature, our overall results favor deterministic generation across detection, classification, and repair. We observed that lower temperatures (specifically 0) yield the most robust performance for all three tasks. While high temperature is often associated with creativity, in the context of vulnerability analysis, it manifests as noise. For detection and classification, higher temperatures degrade the consistency of verdicts, where a model might correctly identify a vulnerability in one run but fail in a subsequent run under identical conditions if the decoding parameters allow for variance. This stochasticity undermines the trustworthiness of LLMs as automated scanners. For repair, the negative impact of high temperature is even more pronounced because vulnerability repair requires precise syntactic adherence to the codebase. We found that increasing temperature often leads models to hallucinate non-existent library functions or generate syntactically incorrect patches, rather than finding creative solutions to the bug.

In terms of nucleus sampling (Top-P), our findings challenge the conventional wisdom that constraining the token search space always improves quality. While lowering Top-P is effective for reducing incoherence in open-ended text generation, we found that the unconstrained setting (Top-P = 1) often yields superior results, particularly for classification and repair. Vulnerability analysis relies on specific technical terminology, such as specific CWE identifiers, rare function calls, or precise patch patterns. These correct tokens may not always reside within a narrow probability mass. Artificially truncating the vocabulary distribution via a low Top-P can prevent the model from accessing the precise technical terms required for accurate classification or the specific code tokens needed for a valid patch, thereby hindering performance.

Finally, the use of system prompts to assign a persona, such as "You are a Security Expert," demonstrates a dichotomy between general-purpose and specialized models. For general-purpose models like GPT-3.5 and GPT-4, persona priming appears to activate latent domain-specific knowledge, effectively grounding the model's focus. This was particularly helpful in detection and classification on the Zero-Day dataset, where the persona helped the model navigate novel vulnerabilities without prior training data. However, this benefit does not extend to fine-tuned or code-specific models. For these models, the task definition is already internalized via their fine-tuning distribution. Introducing strong system prompts can inadvertently cause a distribution shift, interfering with the model's learned patterns and leading to performance regression, particularly in repair tasks where the model needs to strictly follow the patch format rather than engaging in conversational role-play.

#### 8.5 Comparison Between Different Datasets

A critical observation across our experiments is the performance drop of LLMs when evaluated on the Zero-Day dataset compared to older datasets like BigVul+CVFixes or ExtractFix. This disparity strongly suggests the presence of data leakage or contamination in the non-zero-day datasets, where LLMs may have inadvertently encountered and learned from these samples during their extensive training. The Zero-Day dataset, comprising vulnerabilities reported after the LLMs' training cutoff dates, thus serves as a more unbiased and realistic benchmark, highlighting the true generalization

capabilities of these models on unseen security flaws. This performance gap underscores the imperative for future LLM-based vulnerability analysis research to prioritize evaluation against truly novel and unexposed data to ensure valid and reliable conclusions.

Conversely, when comparing performance on PrimeVul-Subset, a dataset specifically curated for its high label accuracy, against BigVul+CVESFixes, our findings present a different picture. While PrimeVul-Subset’s superior ground truth quality might intuitively lead to different LLM performance, the results are mixed across models and tasks. Some LLMs perform comparably or even slightly better on BigVul+CVESFixes. This indicates that while high label accuracy is undoubtedly beneficial for evaluation, it is not the determinant of LLM effectiveness. Other factors, such as the inherent complexity, diversity, or specific characteristics of the samples within each dataset, also significantly influence the analytical capabilities. The varied outcomes suggest that LLMs’ performance is impacted by multiple factors, although it is undoubted that we should use high-quality datasets for evaluation.

Furthermore, the introduction of the PrimeVul-Full dataset allows us to assess LLM generalization across a broader spectrum of vulnerability types. We observe a marked performance decline when expanding the evaluation from the five most common CWEs in PrimeVul-Subset to the 62 diverse CWEs in PrimeVul-Full. This disparity suggests that current LLMs might have been optimized for prevalent, high-frequency vulnerabilities (e.g., buffer overflows, null pointer dereferences) which are likely over-represented in their pre-training corpora. Consequently, evaluations restricted to top-tier CWEs may overestimate the practical utility of LLMs for general-purpose security auditing, emphasizing the need for benchmarks that cover the "long tail" of software security flaws.

## 8.6 Comparison Between Different Tasks

We notice consistent effectiveness drops from the first task to the third task. For example, the best effectiveness (i.e., F1, micro F1, and correct rate) of detection, classification, and repair against the BigVul+CVESFixes and Zero-Day datasets is 48.64/47.06% versus 34.44/25.40% versus 30.24 /20.40% with the existing prompting strategy, and 61.54/48.70% versus 47.27/40.16% versus 32.37/30.25% with VSP. The reason is that the difficulties of the three tasks increase because more detailed and accurate analysis is required. However, we also notice that on the ExtractFix dataset, the correct rates are higher. A possible reason is that the samples are relatively simple as the ExtractFix tool is a traditional technique which deals with relatively limited vulnerability repair scenarios.

## 8.7 Failure Causes and Recommendations

*8.7.1 Insufficient Contexts.* We observed that LLMs commonly fail on real-world samples due to insufficient contexts. As real-world projects are often large and involve many functions/files, vulnerabilities there are often inter-procedural. This makes vulnerability analysis difficult if we only provide a single function. However, given that LLMs can only process a limited number of tokens at a time, it is impractical to input the whole project into an LLM. **Therefore, a possible solution for this is to provide the context information (e.g., by adding code comments).** This can be achieved by again leveraging (other) LLMs to summarize the external relevant functions. Nevertheless, how to effectively prompt LLMs to generate useful external function summaries still needs to be explored.

*8.7.2 Incomplete Control and Data Flow Analysis.* We notice that, while LLMs have the ability to understand code semantics and perform code analysis to some extent, they fall short of conducting comprehensive control/data flow analysis, compared to conventional static/dynamic techniques. **Thus, for advanced vulnerability analysis, we recommend integrating traditional code analysis techniques with VSP-based approaches.** Specifically, static

analyzers may be employed to extract the code slices dependent on external inputs. This can be helpful to improve both semantics extraction and analysis.

**8.7.3 Out of Token Limit.** LLMs may run out of tokens and thus be unable to output the complete analysis result. This can be due to either the input function or the result being too long. One solution here is to reduce the length of inputs and outputs. Our VSP strategy is an important step which makes the outputs more concise. However, this is not enough because we still need to input a whole function into the LLMs which may consume many tokens. **We recommend integrating program reduction techniques to reduce code length.**

```

1 static void remsock(SocketInfo *f){
2 //all the SocketInfo pointers are set to NULL after freeing.
3 if (!f)
4 return;
5 if (f->ev)
6 g_source_remove(f->ev);
7 g_free(f);
8 }

```

**Answer without comments:**  
A: Step 1: Find vulnerability semantics:  
The function has an input "f" of type SocketInfo. At line 2, "f" is checked not to be NULL. If it is NULL, the function returns. At line 4, the member "ev" of "f" is checked not to be NULL. If it is not NULL, the function removes the event source at line 5. Finally, the pointer "f" is freed at line 6. Therefore, the vulnerability semantics is:  
1 static void remsock(SocketInfo \*f)  
2 if (!f)  
3 return;  
4 if (f->ev)  
5 g\_source\_remove(f->ev);  
6 g\_free(f);  
Step 2: Vulnerability semantics analysis:  
At line 2, "f" is checked not to be NULL.  
At line 4, the member "ev" of "f" is checked not to be NULL before removing the event source.  
At line 6, the pointer "f" is freed. However, it is not checked whether "f" has been freed before. Therefore, the code has a use-after-free (CWE-416) vulnerability.  
Conclusion: the code has an exploitable vulnerability.

**Answer with comments:**  
A: Step 1: Find vulnerability semantics:  
The function has an input "f" of type "SocketInfo". If "f" is not NULL, the function removes the event source at line 5 and frees "f" at line 6. Therefore, the vulnerability semantics is:  
1 static void remsock(SocketInfo \*f)  
2 if (!f)  
3 return;  
4 if (f->ev)  
5 g\_source\_remove(f->ev);  
6 g\_free(f);  
Step 2: Vulnerability semantics analysis:  
At line 2, "f" is checked not to be NULL before accessing its members.  
At line 4, the member "ev" of "f" is checked not to be NULL before removing the event source.  
At line 6, "f" is freed and set to NULL, which prevents double free vulnerabilities.  
Conclusion: the code does not have an exploitable vulnerability.

Fig. 23. Telling LLMs the missing contexts by adding comments helps the model do correct analysis.

to mitigate this threat, we follow them and collect a dataset where samples are all published after the LLM cutoff dates.

However, even if we collected a Zero-Day dataset to conduct the experiments, the data leakage/contamination is still possible by the time we move to the next task. (e.g., the same LLM has already seen the zero-day samples during Task 1 by the time of Task 2, since we ran Task 1 before Task 2, if the testing datasets used for Task 1 and Task 2 overlap). To mitigate this possible issue, we started a completely new session for each of the sample to avoid the impact of previous analyzed samples. Also, the tasks are different. Even if the code has been seen by the LLMs, the labels/ground-truth patches are not leaked.

**Reproducibility.** Another threat to the validity is the result reproducibility because LLMs are statistical models and they might return different results when prompted multiple times with the same input. To mitigate this issue, we set temperature to zero to minimize the randomness of the LLMs and provide the detailed versions of the LLMs. We also mitigate this issue by using larger datasets. In this case, while the results on single samples may not be reproducible, the overall performance on the whole datasets is more reproducible.

**Generalizability.** We would like to note that our study currently utilizes existing datasets, which do not cover all the different types of vulnerabilities. For consistency and comparison purposes, we also had to align the CWE coverage

#### 8.7.4 Not Following Exemplars/Logic Errors/Unreadable Output.

We notice that LLMs suffer from not following exemplars, logic errors, and unreadable output issues because of their own capability limitations. Besides, we also notice the CoT-based prompting may not always work for LLMs, indicating that simple few-shot learning may not be enough. This indicates that some current LLMs may not be capable enough for effective vulnerability analysis. **Thus, using powerful LLMs is necessary for effective vulnerability analysis.**

## 8.8 Threats to Validity

**Data Leakage/Contamination.** The major threat to the validity is that the latest LLMs are trained with existing data on the Internet. Therefore, the evaluation data may have been seen by the LLMs during the training, causing data leakage/contamination. This is a challenging problem because most powerful LLMs do not disclose training datasets, and ordinary users also do not have the capacity to train such LLMs. Existing works have noticed this issue and tried to solve this [94] by building new datasets. To

across selections from the different original datasets. For instance, in order to compare the LLMs' performance between our curated zero-day dataset and existing datasets, we aim to ensure they cover the same set of CWEs. Covering more or all CWEs in our datasets collected from the existing data sources implies that we need to collect the same CWEs in our zero-day dataset. However, given that our zero-day dataset can only include the most recent CVEs that are disclosed after the latest model cutoff date among the studied LLMs, it is not feasible to collect such zero-day samples of as desirably many different CWEs. Of course, we could use more CWEs in one experiment and then compare between datasets via other experiments (e.g., focusing the comparisons on the common CWEs); yet that would further complicate the presentation and discussions. Our goal was to assess how LLMs handle such complex reasoning tasks, which are often more difficult than vulnerabilities commonly found in higher-level applications (e.g., injections, XSS). While we acknowledge that these datasets do not cover the full spectrum of vulnerabilities, they offer a critical testing ground for evaluating the fundamental reasoning capabilities of LLMs. The vulnerabilities that we target are mostly in the top-25 most dangerous CWE list [1].

Besides, our work is also subject to certain limitations. Firstly, our evaluation focuses exclusively on C/C++ code and a predefined set of five Common Weakness Enumerations (CWEs) for classification and repair tasks. While C/C++ is highly vulnerable and these CWEs are among the most dangerous, our findings may not directly generalize to other programming languages or a broader spectrum of vulnerability types. Secondly, we assessed a selection of ten LLMs, prioritizing widely used and powerful versions. While we included a fine-tuned baseline for comparison, we did not explore extensive fine-tuning across all open-source models or more complex prompting strategies like Tree of Thoughts, due to practical constraints and our focus on direct prompt engineering. Lastly, while our datasets are large-scale within the chosen context (over 8,000 C/C++ samples), they do not encompass the entirety of real-world software projects, which often involve inter-procedural vulnerabilities requiring analysis beyond single functions. These deliberate choices allowed for an in-depth empirical study within manageable bounds, but they inherently define the boundaries of our conclusions.

**Prompting Strategies.** Although we constrain Naive CoT using systematic rules, any natural-language description of code inevitably contains some degree of abstraction. However, this limitation applies uniformly across all Naive CoT exemplars and does not affect our core conclusion: generic chain-of-thought structure alone is insufficient, whereas vulnerability-semantics-guided prompting yields consistent improvements.

Furthermore, we cannot verify whether the dependencies referenced in LLM-generated reasoning correspond to those identified by formal program analysis; accordingly, VSP should be viewed as a practical heuristic for improving reasoning relevance rather than a substitute for static or dynamic analysis.

**Statistical Significance.** Our empirical evaluation involved running each LLM and prompting strategy configuration once per dataset. This approach was necessitated by the extensive computational resources and time required for a study of this scale, encompassing multiple LLMs, prompting strategies, tasks, and large real-world datasets (over 8,000 samples). To minimize the inherent randomness of LLM outputs, we consistently set the temperature parameter to zero for all experiments. While this helps reduce variability, we acknowledge that without repeated experiments and formal statistical significance testing, we cannot definitively claim that every observed performance difference, particularly smaller ones, is statistically significant. Therefore, our conclusions regarding performance improvements are based on consistent trends observed across multiple LLMs, tasks, and datasets, as well as the substantial improvements in reasoning correctness, rather than statistically proven superiority in every single instance. Future work could involve repeated experiments and statistical analysis to further validate these findings.

## 9 RELATED WORK

**Vulnerability detection/classification.** Traditionally, this problem is addressed through static [22, 45, 48] and/or dynamic code analysis [13, 91]. Respective evaluations for these techniques also exist [47, 67]. Another direction is to leverage machine learning, especially deep learning (DL) [28, 38, 60, 88, 95, 105]. Besides, there exist empirical studies evaluating and trying to improve them [15, 71].

The vulnerability detection task in this paper differs from *vulnerability identification*—asking if a given program has a specific type (CWE) of vulnerability [66], which is also a binary decision problem but with a focused search target (i.e., CWE) as an additional hint. Another related task is *vulnerability localization* [98], which aims to pinpoint the vulnerability code lines hence is intuitively more challenging for both smaller, deep-learning models [70] and larger LLMs [100]—although both can be improved via (e.g., training/fine-tuning [68] and RAG [100]) data augmentation.

**Vulnerability repair.** There exist traditional vulnerability repair techniques based on test cases [32] or learned patterns [10, 58]. DL-based models are also trained for vulnerability repair [17, 18, 29, 88]. Lately, Pearce et al. [75] applied several LLMs in the zero-shot setting to vulnerability repair, showing the challenges with repairing real-world vulnerabilities. Other LLM-based techniques are also proposed for vulnerability repair [36, 46, 80, 83, 96], but they require external information (e.g., error messages) and fine-tuning. In comparison, we comprehensively evaluate LLM-prompting-based approaches alongside fine-tuning. We examine various settings beyond zero-shot to determine if prompting can serve as an effective approach to improve LLM-based vulnerability repair.

In this work, we examine different prompting strategies and suggest solutions to improve them, which complement relevant other efforts such as generating training data for DL-based vulnerability repair [14, 72] and CoT exemplars for LLM-based vulnerability patching [73]—which is further enhanced through neural-symbolic reasoning synergies [74].

**LLMs for vulnerability analysis.** Existing studies on LLM-based vulnerability analysis have usually been conducted on small datasets, with less than 300 samples in total [84, 87] or only 30 real-world samples [87]. Ding et al. [21] built a larger dataset with 6,968 samples to evaluate LLMs on vulnerability analysis, but it only evaluates the LLMs without in-depth analysis. They also only focus on single vulnerability-analysis tasks [21, 75, 84, 87], which limits their scope. In contrast, we conduct comprehensive evaluation on LLMs for vulnerability analysis across three different tasks and prompting strategies and provide potential solutions to improve them.

## 10 CONCLUSION

We present an extensive study assessing how LLMs fare on a range of software vulnerability analysis tasks (detection, classification, and repair) versus a suite of state-of-the-art traditional approaches to these same tasks against a large set of real-world code samples. We revealed varying yet overall undesirable performance across the LLMs and identified underlying challenges, and proposed an improvement strategy via vulnerability-semantics-guided prompting (VSP). Our results indicate that VSP enhances reasoning correctness and aids prediction capabilities for some of the LLMs, signaling a potential direction for LLM-based vulnerability analysis.

## ACKNOWLEDGMENT

We thank our associate editor and reviewers for insightful advice and constructive comments. For this work, Yu Nong, Guangbei Yi, and Haipeng Cai were supported by Army Research Office (ARO) under Grant No. W911NF21-1-002, National Science Foundation (NSF) under Grant No. CCF-2146233 and CCF-2505223, and Office of Naval Research (ONR) under Grant No. N000142212111 and 000142512252. Mohammed Aldeen and Long Cheng were supported by NSF

under Grant No. 2523814, 2448205, 2239605 and 2228616. Hongxin Hu was supported by NSF under Grant No. 2228617, 2120369, and 2129164.

## REFERENCES

- [1] 2022. 2022 CWE Top 25 Most Dangerous Software Weaknesses. [https://cwe.mitre.org/top25/archive/2022/2022\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html).
- [2] 2022. FlawFinder. <https://dwheeler.com/flawfinder/>.
- [3] 2024. CodeQL Documentation. <https://codeql.github.com/docs/>.
- [4] 2024. Meet Claude. <https://www.anthropic.com/claude>.
- [5] 2024. Why instruction tune LLMs? <https://www.ibm.com/topics/instruction-tuning>.
- [6] 2024. Why is Claude COMPLETELY ignoring basic instructions despite triple-mentioning them? [https://www.reddit.com/r/ClaudeAI/comments/1bvoacr/why\\_is\\_claude\\_completely\\_ignoring\\_basic/](https://www.reddit.com/r/ClaudeAI/comments/1bvoacr/why_is_claude_completely_ignoring_basic/).
- [7] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2006. Compilers: Principles, Techniques, and Tools.
- [8] Masaki Aota, Hideaki Kanehara, Masaki Kubo, Noboru Murata, Bo Sun, and Takeshi Takahashi. 2020. Automation of vulnerability classification from its description using machine learning. In *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–7.
- [9] Andrew Austin, Casper Holmgreen, and Laurie Williams. 2013. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology* 55, 7 (2013), 1279–1288.
- [10] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
- [11] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 30–39.
- [12] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 213–223.
- [13] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 133–143.
- [14] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating vulnerable code via learning-based program transformations. In *AI Embedded Assurance for Cyber Systems*. Springer, 123–138.
- [15] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering (TSE)* 48, 9 (2021), 3280–3296.
- [16] Yizheng Chen, Zhoujie Ding, Lamyia Alowain, Xinyun Chen, and David Wagner. 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*. 654–668.
- [17] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.
- [18] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
- [19] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.
- [20] Seyed Shayan Daneshvar, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. 2026. VulScribeR: Exploring RAG-based vulnerability augmentation with LLMs. *ACM Transactions on Software Engineering and Methodology* 35, 5 (2026), 1–26.
- [21] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 469–481.
- [22] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The 2021 Annual Network and Distributed System Security Symposium (NDSS’21)*. 1–15.
- [23] Ericsson. 2023. Software vulnerability: Impact & ways to avoid it. <https://www.ericsson.com/en/security/vulnerability-management>.
- [24] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. 508–512.
- [25] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, et al. 2024. Large language models for code analysis: Do LLMs really do their job?. In *USENIX Security Symposium*. 829–846.
- [26] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.
- [27] Forbes Technology Council. 2023. Zero-Day Vulnerabilities: 17 Consequences And Complications. <https://www.forbes.com/sites/forbestechcouncil/2023/05/26/zero-day-vulnerabilities-17-consequences-and-complications/?sh=711e37204b41>.
- [28] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 608–620.

- [29] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 935–947.
- [30] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 2093–2110.
- [31] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.
- [32] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.
- [33] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).
- [34] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 49–64. <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller>
- [35] Daniel Han and Michael Han. 2024. Unslloth. <https://github.com/unsllothai/unslloth>.
- [36] Sichong Hao, Xianjun Shi, Hongwei Liu, and Yanjun Shu. 2023. Enhancing Code Language Models for Program Repair by Curricular Fine-tuning Framework. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 136–146.
- [37] Roece Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 118–128.
- [38] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 596–607.
- [39] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).
- [40] Ilan Peleg. 2023. The High Cost Of Security Vulnerabilities. <https://www.forbes.com/sites/forbesbusinesscouncil/2023/04/10/the-high-cost-of-security-vulnerabilities-why-observability-is-the-solution/?sh=90da08612ae6>.
- [41] Information Technology Laboratory at NIST. 2023. National Vulnerability Database (NVD) Dashboard. <https://nvd.nist.gov/general/nvd-dashboard>.
- [42] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.
- [43] Harshit Joshi, José Cambrero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.
- [44] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.
- [45] Daniel Kroening and Michael Tautschnig. 2014. CBMC—C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 389–391.
- [46] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2024. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. *arXiv preprint arXiv:2406.00992* (2024).
- [47] Peng Li and Baojiang Cui. 2010. A comparative study on software vulnerability static analysis techniques and tools. In *International Conference on Information Theory and Information Security*. 521–524.
- [48] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE-Demo)*. 1621–1625.
- [49] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 847–859.
- [50] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. {PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.
- [51] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*.
- [52] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. Pyrrfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1659.
- [53] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. 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*. 292–303.
- [54] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.
- [55] 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 *Network and Distributed System Security (NDSS) Symposium*. 1–15.

- [56] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 544–555.
- [57] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.
- [58] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security—ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11–15, 2017, Proceedings, Part II 22*. Springer, 229–246.
- [59] Daniel Marjamäki. 2024. Cppcheck: a tool for static C/C++ code analysis. <https://cppcheck.sourceforge.io/>.
- [60] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6557–6574. <https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky>
- [61] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. *arXiv preprint arXiv:2410.05229* (2024).
- [62] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2450–2462 pages.
- [63] National Institute of Standards and Technology (NIST). 2022. National Vulnerability Database (NVD). <https://nvd.nist.gov>.
- [64] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.
- [65] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for {Use-After-Free} Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 47–62.
- [66] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230* (2024).
- [67] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology* 137 (2021), 106614.
- [68] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 1837–1849.
- [69] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109.
- [70] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VulGen: Realistic Vulnerable Sample Generation via Pattern Mining and Deep Learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2527–2539.
- [71] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering (TSE)* 49, 4 (2022), 1983–2005.
- [72] Yu Nong, Haoran Yang, Feng Chen, and Haipeng Cai. 2024. VinJ: An automated tool for large-scale software vulnerability data generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 567–571.
- [73] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2024. Automated software vulnerability patching using large language models. *arXiv preprint arXiv:2408.13597* (2024).
- [74] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {Real-World} software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*. 4481–4500.
- [75] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.
- [76] Andre Protas and Steve Manzuik. 2023. Skeletons in Microsoft’s Closet - Silently Fixed Vulnerabilities. <https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Manzuik.pdf>.
- [77] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
- [78] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen’sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*, Vol. 14. Citeseer.
- [79] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [80] Fernando Vallecillos Ruiz, Anastasiia Grishina, Max Hort, and Leon Moonen. 2024. A Novel Approach for Automatic Program Repair using Round-Trip Translation with Large Language Models. *arXiv preprint arXiv:2401.07994* (2024).
- [81] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.
- [82] Deepika Shanmugasundaram, Pallavi Arivukkarsu, Huaming Chen, and Haipeng Cai. 2025. Deep Learning Representations of Programs: A Systematic Literature Review. *Comput. Surveys* 58, 5 (2025), 1–37.

- [83] André Silva, Sen Fang, and Martin Monperrus. 2023. Repairllama: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698* (2023).
- [84] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. *arXiv preprint arXiv:2403.17218* (2024).
- [85] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).
- [86] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
- [87] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy*. 862–880.
- [88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)* 30 (2017), 1–11.
- [89] Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. 1993. *Probability and statistics for engineers and scientists*. Vol. 5. Macmillan New York.
- [90] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 999–1010.
- [91] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. 2018. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *J. Parallel and Distrib. Comput.* 118 (2018), 100–106.
- [92] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
- [93] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface’s transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).
- [94] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2023. Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair. *arXiv preprint arXiv:2310.16253* (2023).
- [95] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*. 2365–2376.
- [96] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).
- [97] Guanqun Yang, Shay Dineen, Zhipeng Lin, and Xueqing Liu. 2021. Few-sample named entity recognition for security vulnerability reports by fine-tuning pre-trained language models. In *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*. Springer, 55–78.
- [98] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to detect and localize multilingual bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.
- [99] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.
- [100] Guangbei Yi, Yu Nong, Minzhang Li, and Haipeng Cai. 2026. Exploring and Improving Real-World Vulnerability Data Generation via Prompting Large Language Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2044–2056.
- [101] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.
- [102] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. 2022. Example-based vulnerability detection and repair in Java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 190–201.
- [103] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and Roadmap. *arXiv preprint arXiv:2404.02525* (2024).
- [104] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *ICSE New Ideas and Emerging Results Track*. 47–51.
- [105] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019), 10197–10207.
- [106] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931* (2024).
- [107] Yoshua X ZHxang, Yann M Haxo, and Ying X Mat. 2023. Falcon LLM: A New Frontier in Natural Language Processing. *AC Investment Research Journal* 220, 44 (2023).