Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities

Avishree Khare*
University of Pennsylvania
Philadelphia, USA
akhare@seas.upenn.edu

Saikat Dutta*
Cornell University
Ithaca, USA
saikatd@cornell.edu

Ziyang Li
University of Pennsylvania
Philadelphia, USA
liby99@seas.upenn.edu

Alaia Solko-Breslin
University of Pennsylvania
Philadelphia, USA
alaia@seas.upenn.edu

Rajeev Alur
University of Pennsylvania
Philadelphia, USA
alur@seas.upenn.edu

Mayur Naik
University of Pennsylvania
Philadelphia, USA
mhnaik@seas.upenn.edu

***Abstract*—Security vulnerabilities in modern software are prevalent and harmful. While automated vulnerability detection techniques have made promising progress, their scalability and applicability remain challenging. The remarkable performance of Large Language Models (LLMs), such as GPT-4 and CodeLlama, on code-related tasks has prompted recent works to explore if LLMs can be used to detect security vulnerabilities. In this paper, we perform a more comprehensive study by examining a larger and more diverse set of datasets, languages, and LLMs, and qualitatively evaluating detection performance across prompts and vulnerability classes. Concretely, we evaluate the effectiveness of 16 pre-trained LLMs on 5,000 code samples—1,000 randomly selected each from five diverse security datasets. These balanced datasets encompass synthetic and real-world projects in Java and C/C++ and cover 25 distinct vulnerability classes.**

Our results show that LLMs across all scales and families show modest effectiveness in end-to-end reasoning about vulnerabilities, obtaining an average accuracy of 62.8% and F1 score of 0.71 across all datasets. LLMs are significantly better at detecting vulnerabilities that typically only need intra-procedural reasoning, such as OS Command Injection and NULL Pointer Dereference. Moreover, LLMs report higher accuracies on these vulnerabilities than popular static analysis tools, such as CodeQL.

We find that advanced prompting strategies that involve step-by-step analysis significantly improve performance of LLMs on real-world datasets in terms of F1 score (by up to 0.18 on average). Interestingly, we observe that LLMs show promising abilities at performing parts of the analysis correctly, such as identifying vulnerability-related specifications (e.g., sources and sinks) and leveraging natural language information to understand code behavior (e.g., to check if code is sanitized). We believe our insights can motivate future work on LLM-augmented vulnerability detection systems.

I. Introduction

Security vulnerabilities afflict software despite decades of advances in programming languages, program analysis tools, and software engineering practices. Even well-tested and critical software such as OpenSSL, a widely used library for applications that provide secure communications, contains trivial buffer overflow vulnerabilities, e.g., [1] and [2]. A recent study by Microsoft showed that more than 70% of vulnerabilities are still caused by well-understood memory safety issues [3]. This is alarming given the rapidly growing size and complexity of modern software systems, encompassing numerous programs, libraries, and modules that interact with each other. Hence, we need major technical advances to effectively detect security vulnerabilities in such complex software.

Traditional techniques for automated vulnerability detection, such as fuzzers [4], and static analyzers such as CodeQL [5] and Semgrep [6] have made promising strides. For example, in the last two years, researchers found over 300 security vulnerabilities through custom CodeQL queries [7], [8]. However, these techniques face challenges in scalability and applicability. Fuzzing requires manually crafted fuzz drivers and does not scale to large critical programs with complex inputs, such as network servers, embedded firmware, and system services. On the other hand, static analysis relies heavily on manual API specifications, and skillfully crafted heuristics to balance precision and scalability. Until recently, GitHub paid a bounty of over 7K USD for each CodeQL query that found new critical security bugs [9].

Large Language Models (LLMs), including pre-trained models such as GPT-4 and CodeLlama, have made remarkable advances in code-related tasks in a relatively short period. Such tasks include code completion [12], automated program repair [13]–[15], test generation [16], [17], code evolution [18], and fault localization [19]. These results clearly show the promise of LLMs, opening up a new direction for exploring advanced techniques. Hence, an intriguing question is whether the state-of-the-art pre-trained LLMs can also be used for detecting security vulnerabilities in code.

To develop LLM-based solutions, an important first step is to systematically evaluate the ability of LLMs in detecting *known* vulnerabilities. This is especially important in light of the rapidly evolving landscape of LLMs in three aspects: *scale*, *diversity*, and *applicability*. First, scaling these models to ever larger numbers of parameters has led to significant improvements in their capabilities [20]. For instance, GPT-4, which is presumably orders of magnitude larger than its 175-

*Equal contribution

TABLE I: Summary of our research questions and key findings

Research Questions	Findings
RQ1: How do different pre-trained LLMs perform in detecting security vulnerabilities across different languages and datasets? (Section §III-A)	✓ LLMs across all sizes report a mean accuracy of about 62.8% and a mean F1 score of 0.71 across all datasets. ✗ Average accuracy on real-world datasets is 10.5% lower than that on synthetic datasets. ✓ In stark contrast to other domains, smaller models such as Qwen-2.5-14B and Qwen-2.5-32B report higher accuracies on the real-world datasets than much larger models such as GPT-4.
RQ2: How do different prompting strategies affect the performance of LLMs? (Section III-B)	✓ Using prompts that focus on detecting specific CWEs improves the performance of LLMs. ✓ The Dataflow analysis-based prompt prompt further improves results for larger LLMs with an increase of up to 0.18 in F1 score on real-world datasets. ✓ We also observe that LLMs often infer the correct sources/sinks/sanitizers but fail in end-to-end reasoning.
RQ3: How does the performance of LLMs vary across different vulnerability classes? (Section III-C)	✓ LLMs are better at detecting local vulnerabilities that require no global context across datasets (OS Command Injection, NULL Pointer Dereference, Out-of-bounds Read/Write, etc.). ✗ LLMs struggle to detect vulnerabilities that require additional context or reasoning about complex data structures (Out-of-bounds Read/Write with C++ structs and pointers). ✓ Certain LLMs are very good at detecting specific CWEs across datasets (Llama-3.1-70B on OS Command Injection, DeepSeekCoder-7B on NULL Pointer Dereference, etc.).
RQ4: How do LLMs compare to state-of-the-art static analysis tools? (Section III-D)	✗ LLMs report lower overall accuracies than CodeQL on all synthetic datasets. ✓ LLMs report higher accuracies than CodeQL on certain vulnerability classes across datasets (Path Traversal, OS Command Injection, etc). CodeQL reports higher accuracies on Integer Overflow across datasets.
RQ5: How do LLMs compare to state-of-the-art deep-learning-based tools? (Section III-E)	✓ Deep Learning(DL)-based tools such as DeepDFA [10] and LineVul [11] report accuracies similar to Qwen-2.5-32B on CVEFixes C/C++ even after being trained on in-distribution samples whereas Qwen-2.5-32B reports higher F1 scores. ✓ DL-based tools report lower accuracies and F1 scores than LLMs when trained and evaluated on different datasets. ✓ LLMs provide natural language explanations for their predictions while DL-based tools provide binary scores and line numbers that are often difficult to interpret.

billion predecessor GPT-3.5, significantly outperforms GPT-3.5 on a wide range of code-understanding tasks [21]. Second, the diversity of LLMs has grown rapidly and now includes not only proprietary general-purpose ones such as GPT-4 but also open-source code-specific LLMs such as CodeLlama [22] and StarCoder [23]. Finally, the reasoning capabilities of LLMs (and hence their applicability) may vary significantly across different prompting strategies and programming languages. All these factors open up a large exploration space for applying LLMs to the challenging task of vulnerability detection.

TABLE II: Comparison with other studies that focus on vulnerability detection with LLMs. Superscript U indicates an unbalanced dataset. Static Analysis is abbreviated as SA and Deep Learning as DL.

Study Features	[24]	[25]	[26]	[27]	[28]	Our Work
Languages	C/C++	C/C++	C/C++	C,Py	C/C++	C/C++,Java
#Samples	368	347	100	96	25.9K^{U}	5000
#CWEs	25	N/A	N/A	8	140	25
#LLMs (>1B parameters)	3	16	11	5	4	16
Comparison of various LLMs	✓	✓	✓	✓	✓	✓
Qualitative prompt analysis	✗	✗	✗	✓	✗	✓
CWE-wise analysis	✗	✗	✗	✗	✗	✓
Comparison with SA tools	✗	✓	✗	✗	✗	✓
Comparison with DL tools	✓	✗	✗	✗	✓	✓

Our Work. We study the vulnerability detection capabilities of 16 state-of-the-art LLMs across different scales and families, including proprietary models such as Gemini and GPT-4, and open-source models like CodeLlama and Qwen. We evaluate these models on five popular security vulnerability datasets across two languages, C/C++ and Java, and 25 vulnerability classes.

We first study how LLMs perform on the task of vulnerability detection using three prompting strategies and how these strategies qualitatively compare against each other. We also attempt to identify vulnerability classes that most benefit from the use of LLMs and these prompting techniques. Our simplest prompting strategies include the *Basic prompt*, which simply asks an LLM to check for any vulnerabilities in the given code and the *CWE specific prompt*, which asks the LLM to check for a specific class of vulnerabilities or CWEs (such as Buffer Overflows). Inspired by the success of static analysis tools like CodeQL that use dataflow analysis to detect vulnerabilities, we design a prompting strategy called *Dataflow analysis-based prompt*. This prompt asks the LLM to simulate a source-sink-sanitizer based dataflow analysis on the target code snippet before predicting if it is vulnerable.

We next compare LLMs with existing vulnerability detection tools, namely static analysis-based CodeQL and two deep learning-based techniques, DeepDFA [10] and LineVul [11]. As discussed earlier, static vulnerability detection tools are limited by the need for concrete API specifications and require compiling / building entire target projects before detection. Pre-LLM deep learning-based approaches such as DeepDFA [10] and LineVul [11] attempt to mitigate some of these limitations through fine-tuned neural representations of code. On the other hand, LLMs do not require the compilation of complete projects as they can be prompted to analyze partial code snippets. Moreover, they already have an internal model of APIs through pre-training and do not need to be trained

on large datasets from scratch. We analyze the benefits and shortcomings of LLMs over CodeQL, including vulnerability classes where they outperform each other. We also study how they compare against the deep learning based-approaches in terms of generalization across datasets.

Comparison with other studies. There are other studies that also evaluate the effectiveness of LLMs on the task of vulnerability detection [24]–[28]. Table II presents a comparison of our work with these studies. We present our study as the most comprehensive on this topic for the following reasons:

- **Size and diversity of the datasets:** We curate a dataset of 5K samples, with equal number of vulnerable and non-vulnerable snippets. The only larger dataset is from [28] but only 695 of their 25.9K samples are vulnerable. Furthermore, our study is the first to include Java code.
- **Comparison with non-LLM-based tools:** Our study is the first to compare LLMs with CodeQL and specialized Deep Learning-based tools. Moreover, we also find vulnerability classes where LLMs outperform CodeQL and vice versa which is useful for deployment.
- **Qualitative analysis of prompts and vulnerability classes:** While other studies quantitatively compare prompting strategies, we also attempt to qualitatively identify the benefits of various prompt elements. Furthermore, we identify partial capabilities offered by some of these prompts that can be leveraged in LLM-based detection tools. We also identify vulnerability classes where LLMs perform well.

Contributions. Our research questions and key findings are summarized in Table I. To summarize, we make the following contributions in this paper:

- **Empirical Study:** We conduct the largest comprehensive study on how state-of-the-art LLMs perform in detecting security vulnerabilities across 5000 samples from five datasets, two programming languages (C/C++ and Java) and covering 25 unique vulnerability classes.
- **Comparison with other vulnerability detection tools**: We contrast the performance of LLMs against popular static analysis and deep-learning-based vulnerability detection tools. We also identify vulnerability classes where LLMs perform better/worse than some of these tools.
- **Qualitative comparison of prompting strategies:** We quantitatively and qualitatively compare three prompting strategies, including a novel prompt inspired by dataflow analysis-based vulnerability detection.
- **Insights:** We perform a rigorous manual analysis of LLMs' predictions and highlight vulnerability patterns that impact the performance of these models.

II. Approach

A. Datasets

For our study, we select five diverse synthetic/real-world vulnerability datasets from two languages: C++ and Java. Table III presents the details of each dataset, such as the dataset size, programming language, number of vulnerable and non-vulnerable samples, and the number of unique CWEs. The synthetic benchmarks, *OWASP* and *Juliet*, allow for easy comparison with CodeQL and the real-world benchmarks, *CVEFixes*, are useful for evaluating practical utility. While many real-world datasets have been proposed in the literature, we selected *CVEFixes* because it is the only dataset that 1) contains vulnerability *metadata* such as CVE and CWE IDs, 2) is *two-sided*, i.e., contains both vulnerable and non-vulnerable code samples, and 3) covers multiple languages such as Java and C/C++. Table IV shows a comparison of existing real-world vulnerability datasets. We briefly describe each of the selected datasets next:

TABLE III: Details of Selected Datasets

Dataset	Language	Size	Vul/Non-Vul	CWEs
OWASP [29]	Java	2740	1415/1325	11
SARD Juliet (C/C++) [30]	C/C++	81,280	40,640/40,640	118
SARD Juliet (Java) [31]	Java	35,940	17,970/17,970	112
CVEFixes [32]	C/C++	19,576	8223/11,347	131
CVEFixes [32]	Java	3926	1461/2465	68

1) OWASP (Synthetic): The Open Web Application Security Project (OWASP) benchmark [29] is a Java test suite designed to evaluate the effectiveness of vulnerability detection tools. Each test represents a synthetically designed code snippet containing a security vulnerability.

2) Juliet (Synthetic): Juliet [33] is a widely-used vulnerability dataset developed by NIST with thousands of synthetically generated test cases from various known vulnerability patterns.

3) CVEFixes (Real-World): Bhandari et al. [32] curated a dataset, known as CVEFixes, from 5365 Common Vulnerabilities and Exposures (CVE) records from the National Vulnerability Database (NVD). From each CVE, they automatically extracted the vulnerable and patched versions of each method in open-source projects, along with extensive meta-data such as the corresponding CWEs, project information, and commit data. These methods span multiple programming languages but we only consider the C/C++ and Java methods in this work.

TABLE IV: Comparison of Real-World Datasets

Dataset	Languages	CVE Metadata	Two-Sided	Multi-Lang
BigVul [34]	C/C++	✓	✗	✗
Reveal [35]	C/C++	✗	✗	✗
DiverseVul [36]	C/C++	✗	✓	✗
DeepVD [37]	C/C++	✗	✗	✗
CVEFixes [32]	C/C++, Java, ...	✓	✓	✓

B. Metrics

To evaluate the effectiveness of each tool, we use the standard metrics used for classification problems. In this work, a true positive represents a case when a tool detects a true vulnerability. In contrast, a false positive is when the tool detects a vulnerability that is not exploitable. True and false negatives are defined analogously. We describe each metric in the context of vulnerability detection.

- **Accuracy:** Accuracy measures how often the tool makes a correct prediction, i.e., whether a code snippet is vulnerable or not. It is computed as: $\frac{True\ Positives\ +\ True\ Negatives}{\#Samples}$.

- **Precision:** Precision represents what proportion of cases that a tool detects as a vulnerability is a correct detection. It is computed as: $\frac{True\ Positives}{True\ Positives\ +\ False\ Positives}$.
- **Recall:** Recall represents what proportion of vulnerabilities the tool can detect. It is computed as: $\frac{True\ Positives}{True\ Positives\ +\ False\ Negatives}$.
- **F1 score:** The F1 score is a harmonic mean of precision and recall. It is computed as: $2 * \frac{Precision\ *\ Recall}{Precision\ +\ Recall}$.

C. Large Language Models

We choose the most popular state-of-the-art pre-trained Large Language Models (LLMs) for our evaluation. We choose three closed-source models (GPT-4, GPT-3.5 and Gemini-1.5-Flash) and thirteen open-source models from the Codellama-x, Llama-3.1-x, Mistral-Codestral-x, DeepSeekCoder-x, Qwen2.5-x and Qwen2.5-Coder-x series. We use the "Instruct" variants of the models wherever applicable since they are fine-tuned to follow user instructions and hence can better adapt to specific reasoning tasks. We access the GPT-x models and Gemini-1.5-Flash using the OpenAI and Google Gemini APIs respectively and use the Hugging Face APIs [38] to access the open-source models. Table V presents more details about the models.

TABLE V: Details of LLMs (increasing order of size)

Model	Model Version	Size	Context Size
Qwen-2.5C-1.5B	`qwen2.5-coder-1.5b`	1.5B	128K
Qwen-2.5C-7B	`qwen2.5-coder-7b`	7B	128K
CodeLlama-7B	`Codellama-7b-instruct`	7B	16K
DSCoder-7B	`deepseekcoder-7b`	7B	4K
Llama-3.1-8B	`llama-3.1-8b`	8B	128K
CodeLlama-13B	`CodeLlama-13B-Instruct`	13B	16K
Qwen-2.5-14B	`qwen2.5-14b`	14B	128K
DSCoder-15B	`deepseekcoder-v2-15b`	33B	128K
Codestral-22B	`mistral-codestral-22b`	22B	32K
Qwen-2.5-32B	`qwen2.5-32b`	32B	128K
DSCoder-33B	`deepseekcoder-33b`	33B	16K
CodeLlama-34B	`CodeLlama-34B-Instruct`	34B	16K
Llama-3.1-70B	`llama-3.1-70b`	70B	128K
Gemini-1.5-Flash	`gemini-1.5-flash`	N/A	1M
GPT-3.5	`gpt-3.5-turbo-0613`	N/A	4K
GPT-4	`gpt-4-0613`	N/A	8K

D. Prompting Strategies for LLMs

We explore various prompting strategies that can assist LLMs in predicting if a given code snippet is vulnerable. The LLMs discussed in this study support chat interactions with two major types of prompts: the *system prompt* can be used to set the context for the entire conversation while *user prompts* can be used to provide specific details throughout the chat session. We include a *system prompt* at the start of each input to describe the task and expected structure of the response. Since persona assignment has been shown to improve the performance of GPT-4 on specialized tasks [39], we add the line "*You are a security researcher, expert in detecting security vulnerabilities*" at the start of every system prompt to assign a persona of a Security Researcher to the model. The system prompt for all experiments ends with the statement "*Provide response only in the following format:*" followed by an expected structure of the response from the model. The system prompt is followed by a *user prompt* that varies across the various prompting strategies. In all our experiments, we incorporate the target code snippet into the user prompt without any changes.

We construct three different prompting strategies:

1) Basic prompt: We design a very simple prompt (shown in Listing 4 in the Appendix) to test if the model can take a target code snippet as input and detect if it is vulnerable and determine the correct CWE as well. The prompt begins with the message "*Is the following code snippet prone to any security vulnerability?*" followed by the code snippet.

TABLE VI: Dataset Processing and Selection

Steps	OWASP	Juliet C/C++	Juliet Java	CVEFixes C/C++	CVEFixes Java	Total
Original	2740	128,198	56,162	19,576	3926	210,602
Filtering	2740	81,280	35,940	19,576	3926	144,002
Top 25 CWE	1478	11,766	8,506	12,062	1810	23,560
Random Selection	1000	1000	1000	1000	1000	5000

2) CWE specific prompt: The CWE specific prompt is presented in Listing 5 in the Appendix. This prompt is similar to the Basic prompt except that it asks the model to predict if the given code snippet is vulnerable to a specific target CWE. Hence, the user prompt starts with "*Is the following code snippet prone to <CWE>?*" followed by the code snippet. For instance, for CWE-476, the user prompt would start with "*Is the following code snippet prone to CWE-476 (NULL Pointer Dereference)?*" followed by the target code snippet.

3) Dataflow analysis-based prompt: Dataflow analysis is used by several static analysis tools to infer if there exists an unsanitized path from a source to a target node. Further, prior literature has shown step-by-step instructions can often elicit better reasoning from LLMs [40]. Motivated by these observations, we designed the CWE-DF prompt (shown in Listing 6 in Appendix) that prompts the model to simulate a source-sink-sanitizer-based dataflow analysis on the target code snippet before predicting if it is vulnerable. Naturally, this prompt is costlier since it generates more tokens than the other prompts. We provide the full prompts in Appendix B.

4) Other prompting strategies: We also tried other prompting strategies such as `Few-shot prompting` and `Chain-of-thought prompting`. In the few-shot prompting setup, we include two examples of the task (one with a vulnerability and one without) in the CWE specific prompt before providing the target code snippet. With Chain-of-thought prompting, we prompt the model to generate a reasoning chain before the final answer by adding a "*Let's think step-by-step*" statement at the end of the CWE specific prompt. Our initial experiments with GPT-4 prompted using these techniques did not yield results better than the Dataflow analysis-based prompt on 100 random samples from two datasets. Hence, we do not include these prompts in this study. We refer readers to Appendix C for more details.

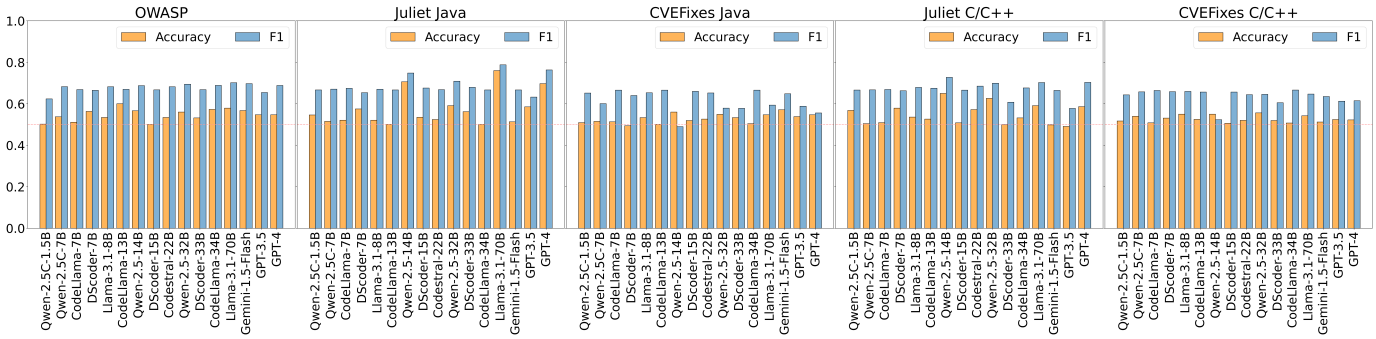


Fig. 1: Effectiveness of LLMs in Predicting Security Vulnerabilities in Java and C/C++ (highest accuracy and F1 scores per model per dataset, across all prompting strategies).

E. Dataset Processing and Selection

We preprocess each dataset before evaluation by removing or anonymizing information such as commits, benchmark IDs, or vulnerability names that may provide obvious hints about the vulnerability. We also skip benchmarks that are spread across multiple files, due to limitations of prompt size. We only consider samples corresponding to vulnerability types (CWEs) listed in MITRE's Top 25 Most Dangerous Software Weaknesses [41]. We then filter code snippets with more than 2048 tokens due to prompt size limitations and randomly sample 500 vulnerable and 500 non-vulnerable samples per dataset. Table VI presents the details of our selection stages. We provide more details for each dataset in Appendix A.

F. Experimental Setup

Experiments with closed-source models. We use the OpenAI public API's `ChatCompletions` endpoint to perform the experiments with GPT-3.5 and GPT-4. We use Google's Gemini API for the experiments with Gemini-1.5-Flash.

Experiments with open-source models. We run all experiments with the open-source LLMs using the HuggingFace API on a cluster with A100, A6000, and RTX 2080 GPUs.

In all our experiments, we set the sampling temperature to 0 for obtaining deterministic predictions, the maximum number of tokens to 1024, and use the default values for all other parameters. We use the top-1 predictions for evaluation.

III. Results

*A. **RQ1:** Effectiveness of LLMs*

We evaluate the performance of pre-trained LLMs on five open-source datasets discussed in Section II-A. Figure 1 presents the best accuracy and F1 scores (across prompts) of the 16 models from Table V on all datasets. The more detailed metrics for all prompts are presented in Appendix D.

Finding 1.1: Modest Vulnerability Detection Performance Across LLMs. The best performing models and prompts per dataset report an accuracy of 62.8% on average. The highest of these is reported by Llama-3.1-70B (with CWE) on the Juliet Java dataset (76%). The other best performing models per dataset are: CodeLlama-13B on OWASP (60%), Gemini-1.5-Flash on CVEFixes Java (57%), Qwen-2.5-14B on Juliet C/C++ (65%) and Qwen-2.5-32B on CVEFixes C/C++ (56%). This confirms that no model individually performs the best across multiple datasets. Moreover, the best accuracies on synthetic datasets are 10.5% higher on average than those on the real-world datasets, suggesting that these models might not be suitable for real-world vulnerability detection yet.

Finding 1.2: Performance does not improve with scale. On the real-world datasets, Qwen-2.5-14B and Qwen-2.5-32B report higher accuracies than the GPT-x models despite being much smaller. We see many similar patterns across all models studied implying that model scale does not impact performance. Within the same model class, the GPT-x models and the Llama-3.1-x models exhibit improvements in accuracy as the size of the model increases on at least three datasets. However, this is not observed in other model classes, i.e., the Qwen-2.5C-x, Qwen-2.5-x, Codellama-x and the DSCoder-x series. This is in stark contrast to other domains where increasing model size leads to better performance. Listing 1 presents a representative example where CodeLlama-7B correctly predicts that an integer overflow vulnerability (CWE-190) cannot occur in the given context while CodeLlama-13B does not.

*B. **RQ2:** Comparison of Prompting Strategies*

Figure 2 presents the accuracy and F1 scores (averaged across all LLMs) of the three prompting strategies across all datasets. Overall, the three prompts perform similarly in terms of accuracy on all datasets with CWE-DF reporting slightly higher accuracies on the real-world datasets. Interestingly, the CWE-DF prompt reports significantly higher F1 scores on average than CWE and Basic prompt on the real-world datasets (by upto 0.18 on CVEFixes C/C++ and 0.14 on CVEFixes Java). Furthermore, CWE-DF reports the lowest variance in accuracies and F1 scores across models, as suggested by the standard deviation bars in Figure 2. On the other hand, the Basic prompt reports the highest variance in F1 scores. We next highlight qualitative differences between prompts:

Finding 2.1: Specifying the CWE in the prompt reduces false alarms. Table VII presents the percentage of samples where the Basic prompt predicts the correct CWE, averaged across datasets for 4 models. We observe that the Basic prompt

```java
private void func() throws Throwable {
    int data;
    switch (5) {
    case 6:
        data = 0; break; // Hardcoded values in
        ↪ data
    default:
        data = 2; break; // Hardcoded values in
        ↪ data
    }
    switch (7) {
    case 7: // POTENTIAL FLAW: Integer Overflow
        int result = (int)(data + 1); // But no
        ↪ overflow due to hardcoded values in
        ↪ data
        IO.writeLine("result: " + result);  break;
    default:
        IO.writeLine("fixed string"); break;
```

Codellama-7B's response: The code snippet does not contain any obvious security vulnerabilities. ✓

Codellama-13B's response: The integer overflow occurs when the value of 'data' exceeds the maximum value that can be stored in an integer, causing the value to wrap around to a negative number... potentially allow an attacker to execute arbitrary code. ✗

Listing 1: CodeLlama-7B correctly predicts this code is not vulnerable to Integer Overflow but CodeLlama-13B does not.

detects incorrect CWEs in $>$ 60% and $>$ 53% of all Java and C/C++ samples across models. We further manually inspected 10 vulnerable and 10 non-vulnerable samples from Juliet Java where GPT-4 with Basic prompt is incorrect. In only 4 or the 20 samples, the Basic prompt predicts a plausible CWE. However, these CWEs are unlikely due to the context. For example, it predicts that a value read from an input stream can be vulnerable if not validated (CWE-20) but this value is not used in a vulnerable context. The CWE specific prompt (i.e., the Basic prompt with CWE) improves or retains accuracy over the Basic prompt on all 5 datasets. GPT-4 with the CWE specific prompt not only correctly predicts all the 20 samples discussed above but also provides useful high-level explanations for why the snippet is vulnerable/not vulnerable in 18 / 20 samples. The 2 incorrect explanations are artifacts of faulty reasoning or hallucination: for example, an Integer Overflow due to addition to INT_MAX in the function is incorrectly attributed to subtracting from INT_MIN in the explanation. Based on these observations, *specifying the CWE in the prompt can be helpful in reducing incorrect predictions.*

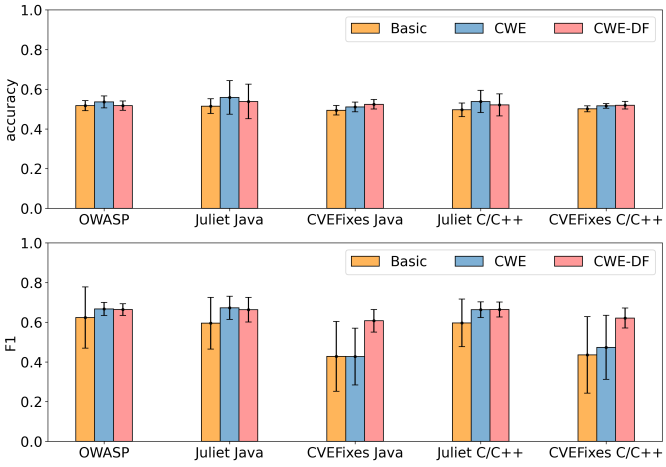


Fig. 2: Performance of different prompting strategies

TABLE VII: Correct CWEs detected with Basic prompt (%)

Language (Avg.)	GPT-4	GPT-3.5	CodeLlama-34B	CodeLlama-13B
Java	0.41	0.34	0.37	0.38
C/C++	0.29	0.31	0.33	0.35

Finding 2.2: Dataflow analysis identifies CWE-relevant textual cues and provides actionable explanations. The Dataflow analysis-based prompt (CWE-DF) performs better than CWE specific prompt on the real-world datasets, CVEFixes C/C++ and CVEFixes Java. We inspect 10 vulnerable and 10 non-vulnerable samples from CVEFixes Java where GPT-3.5 correctly predicts only with CWE-DF. *We find that the CWE-DF prompt leverages textual cues for sanitization (e.g., csrftokenhandler() suggests protection from CSRF) in 16/20 samples that are missed by CWE specific prompt.*

Further, *CWE-DF responses are more useful in localizing the vulnerability as it correctly predicts the correct sources and sinks in 18 / 20 samples, sanitizers in 16 / 20 samples, and unsanitized flows in all samples*. Listing 2 presents a response from GPT-4 using CWE-DF prompt that correctly identifies the unsanitized flows between sources and sinks. We present more CWE-DF examples in Appendix F.

Finding 2.3: CWE-DF identifies the correct sources and sinks even when the final prediction is incorrect. We also inspect 10 vulnerable and 10 non-vulnerable samples from Juliet C/C++ where CWE-DF's predictions are incorrect. Surprisingly, the sources, sinks and sanitizers are correctly identified in 17 / 20 samples but the unsanitized flows are incorrect in 12 / 20 samples. Hence, the final predictions are incorrect only due to erroneous reasoning about the snippet / false assumptions about the CWE. Listing 3 presents an example where the vulnerability is not detected but the sources and sinks are correctly identified. This suggests that *the CWE-DF prompt can be used to accurately identify sources/sinks/sanitizers and other dataflow analysis techniques can be used to reason about unsanitized flows to predict the vulnerability.*

C. *RQ3: Performance of LLMs across CWEs*

We next evaluate how the LLMs perform on different classes of security vulnerabilities (CWEs). Because the CWE-wise distribution of vulnerable and non-vulnerable samples can be imbalanced, we compute balanced accuracy for each CWE (for ease of presentation, we refer to it as accuracy henceforth

```
// Code snippet prone to CWE-78 (OS Command
↪  Injection)
// Unverified data from socket is used as an
↪  argument to _execvp
void func(){
    char dataBuffer[100] = "dir ";
    char *data = dataBuffer; size_t dataLen =
    ↪  strlen(data);
    int recvResult; SOCKET connectSocket =
    ↪  INVALID_SOCKET;
    do {// setup socket and receive user input
        conn = socket(AF_INET, SOCK_STREAM,
        ↪  IPPROTO_TCP);
        recvResult = recv(conn, (char *)(data +
        ↪  dataLen), sizeof(char) * (100 - dataLen
        ↪  - 1), 0)};
        // missing validation of recvResult
        data[dataLen + recvResult / sizeof(char)] =
        ↪  '\0';
    } while (0)
    char *args[] = {"%WINDIR%\\system32\\cmd.exe",
    ↪  "/c", data, NULL};
    _execvp("cmd.exe", args);
}
```

GPT-4's response: The data received from the socket connection is directly used in the `_execvp` function without proper sanitization. This could allow an attacker to inject malicious commands. ✓

Listing 2: GPT-4 (CWE-DF) detects that this snippet is prone to OS Command Injection due to unsanitized paths from a source to sink. CodeQL does not detect this vulnerability.

in this section). For each dataset and model, we consider the best-performing prompt for the analysis and only report CWEs with at least 10 samples. Figure 3 presents the CWE-wise distribution of accuracies on the real-world datasets, CVEFixes Java and CVEFixes C/C++. Figure 4 reports the accuracies on the synthetic datasets, OWASP, Juliet Java and Juliet C/C++.

Finding 3.1: LLMs perform well on vulnerabilities that do not require additional context. These CWEs include *OS Command Injection (CWE-78), Out-of-bounds Read / Write (CWE-125, CWE-787), Null Pointer Dereference (CWE-476), Cross-site Scripting (CWE-79), SQL Injection (CWE-89) and Incorrect Authorization (CWE-863).* The higher performance on these vulnerabilities can be attributed to the fact that these are fairly self-contained and little additional context is needed to detect them. For example, 4/16 LLMs report accuracies $>$ 60% on Incorrect Authorization (CWE-863) on CVEFixes Java it is easier to validate if an implemented authorization check is correct or not. On the other hand, no LLMs report high accuracies on Missing Authorization (CWE-862) since it's not known if an input parameter has been authorized outside the context of the target method and more context is hence needed to detect this vulnerability class. The following summarizes how LLMs perform on these CWEs:

- OS Command Injection (CWE-78) sees the highest performance across models and datasets with >60% accuracies reported by 5/16 LLMs on CVEFixes Java and 11/16 LLMs on CVEFixes C/C++. Llama-3.1-70B reports the best performance on CWE-78 with accuracies 64% and 78% on CVEFixes Java and CVEFixes C/C++ respectively.
- Llama-3.1-8B and GPT-4 perform extremely well on Out-of-bounds Write (CWE-787) with accuracies of 89% and 79% on CVEFixes Java and CVEFixes C/C++ respectively and on Out-of-bounds Read (CWE-125) with accuracies of 84% and 78% respectively. Moreover, accuracies over 60% are reported on CVEFixes Java by 5/16 LLMs for CWE-125 and 4/16 LLMs for CWE-787.
- NULL Pointer Dereference (CWE-476) sees accuracies higher than 60% by 3/16 LLMs on CVEFixes C/C++, 7/16 on Juliet C/C++ and 11/16 on OWASP. Interestingly, DSCoder-7B performs the best on all three datasets with accuracies of 63%, 88% and 83% respectively.
- GPT-3.5 reports the highest accuracy of 70% on SQL Injection (CWE-89) on CVEFixes Java and 7/16 LLMs report accuracies over 60%. Surprisingly, all LLMs report accuracies <60% on the same CWE on synthetic datasets (OWASP and Juliet Java).
- Qwen-2.5-32B reports high accuracies of 67% and 66% on Cross-Site Scripting (CWE-79) on OWASP and CVEFixes Java respectively. Accuracies over 60% are reported by 3/16 LLMs on CVEFixes Java and 11/16 LLMs on OWASP.

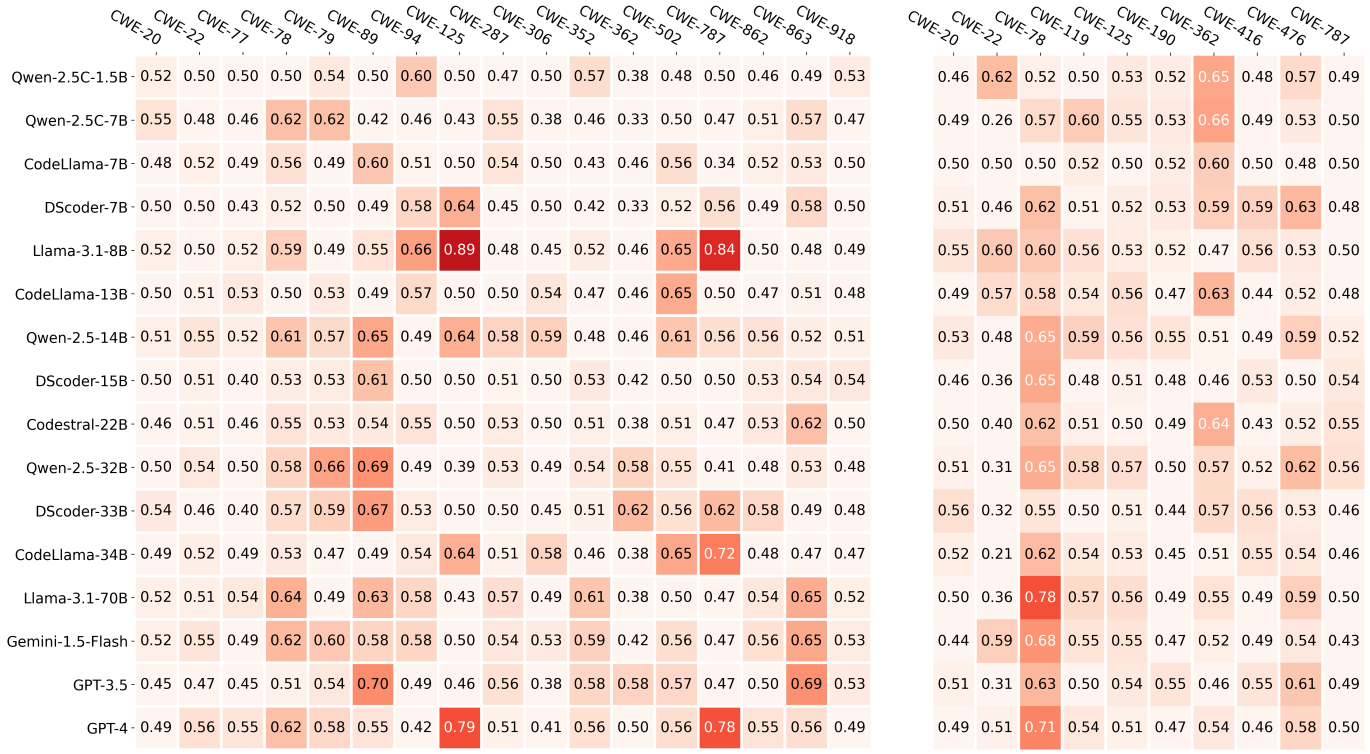
Finding 3.2: Poor performance on real-world C/C++ is due to missing global context. We see that the performance of all LLMs on vulnerabilities in CVEFixes C/C++ is worse than that on the same CWEs in CVEFixes Java and Juliet C/C++. While GPT-4 and Llama-3.1-8B perform extremely well on the Out-of-bounds Read / Write vulnerabilities in CVEFixes Java as discussed above, they report accuracies $<$ 53% for these CWEs on the CVEFixes C/C++ dataset. In fact, no LLMs report accuracies $>$ 60% for these CWEs on CVEFixes C/C++. We attribute this disparity to the nature of these vulnerabilities in the two languages: *Out-of-bounds Reads / Writes in CVEFixes C/C++ require reasoning about pointers and structs which requires more context about the structs and their members.* In CVEFixes Java, on the other hand, these vulnerabilities arise primarily due to illegal array indexing. This issue does not emerge in Juliet C/C++ because all the information about the pointers is presented in the snippet. We present examples in Appendix G.

Finding 3.3: Some LLMs are better at detecting certain CWEs. Concretely, the following LLMs report the best accuracies on certain CWEs across datasets:

- Llama-3.1-70B on OS Command Injection (CWE-78)
- DSCoder-7B on NULL Pointer Dereference (CWE-476)
- Qwen-2.5-32B on Cross-Site Scripting (CWE-79)
- Llama-3.1-8B and GPT-4 on Java Out-of-bounds Read-/Write (CWE-125/787)

D. *RQ4: LLMs vs Static Analysis Tools*

Experiment setup. We next explore how the LLMs compare against CodeQL. Since CodeQL requires building projects before analysis and the real-world datasets contain large projects,



(a) CVEFixes Java

(b) CVEFixes C/C++

Fig. 3: Accuracy Across CWEs on real-world datasets.

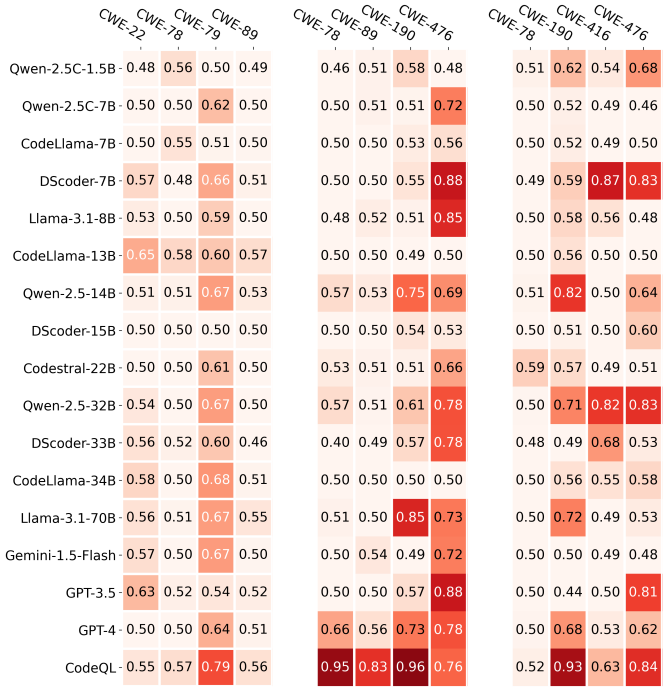
we limit our focus to the three synthetic datasets, namely OWASP, Juliet Java and Juliet C/C++. We use the official CodeQL queries designed for the top 25 CWEs. Table VIII presents results from CodeQL and the best performing LLM on the three datasets: CodeLlama-13B on OWASP, Llama-3.1-70B on Juliet Java and Qwen-2.5-14B on Juliet C/C++.

Finding 4.1: CodeQL performs better than the LLMs in terms of accuracy on all three datasets. CodeQL reports 0.07 and 0.15 higher F1 than CodeLlama-13B on OWASP and Llama-3.1-70B on Juliet Java respectively. However, Qwen-2.5-14B reports a 0.11 higher F1 on Juliet C/C++.

Finding 4.2: LLMs perform better than CodeQL on certain CWEs. A CWE-wise comparison of LLMs with CodeQL in Figure 4 reveals that *LLMs report higher accuracies on CWE-22 (Path Traversal), CWE-78 (OS Command Injection), CWE-476 (NULL Pointer Dereference), 416 (Use After Free) on at least 2 / 3 datasets while CodeQL performs better on CWE-190 (Integer Overflow) on 2 datasets.* Concretely,

- CodeQL performs better than the LLMs on CWE-190 (Integer Overflow) with 11% and 21% higher accuracies than Llama-3.1-70B on Juliet Java and Juliet C/C++ respectively.
- On the other hand, DSCoder-7B performs better than CodeQL on CWE-476 (NULL Pointer Dereference) with 12% higher accuracy on Juliet Java and only 1% lower accuracy on Juliet C/C++. Moreover, 6 / 16 LLMs report accuracies higher than CodeQL on CWE-476 from Juliet Java.
- While CodeQL reports an extremely high accuracy of 95% on CWE-78 (OS Command Injection) with Juliet Java, it is outperformed by CodeLlama-13B by 1% and Codestral-22B by 7% on OWASP and Juliet C/C++ respectively. Interestingly, 7 / 16 LLMs report higher accuracies (by upto 10%) at detecting CWE-22 (Path Traversal) on OWASP.
- Similarly, 3 LLMs perform better on CWE-416 (Use After Free) from Juliet C/C++ with DSCoder-7B reporting a whopping 24% higher accuracy than CodeQL.

Finding 4.3: CodeQL's worse performance on some CWEs can be attributed to the very specific manually-written queries which may not cover all possible scenarios of the vulnerability. For example, CodeQL only detects CWE-78 (OS Command Injection) in C/C++ snippets when there exist system commands that take a string of arguments (`execl`). This query cannot handle commands that take a list of arguments (eg., `_execvp`). Listing 2 provides an example of this scenario where CodeQL does not detect that the snippet is prone to OS Command Injection but GPT-4 (CWE-DF) correctly identifies `_execvp` as a vulnerable sink. Listing 3 presents an example where CodeQL correctly predicts that the target snippet is vulnerable to Integer Overflow while GPT-4 with CWE-DF does not. However, the model correctly identifies the sources, sinks and even unsanitized dataflows in this case but fails to faithfully reason over them when predicting the vulnerability. These examples support the observation from Section III-B that *LLMs can be used to identify sources and sinks relevant to the target vulnerability (which can be missed*



(a) OWASP (b) Juliet Java (c) Juliet C/C++

Fig. 4: Accuracy Across CWEs on synthetic datasets.

by manually written queries) while static analysis tools like CodeQL can be used to reason over them. More examples are presented in the Appendix E.

TABLE VIII: CodeQL vs. best performing LLM on synthetic datasets (CodeLlama-13B on OWASP, Llama-3.1-70B on Juliet Java, and Qwen-2.5-14B on Juliet C/C++)

Dataset/Model	CodeQL				Best performing LLM			
	A	P	R	F1	A	P	R	F1
OWASP	0.63	0.58	0.96	0.72	0.60	0.58	0.74	0.65
Juliet Java	0.94	0.92	0.95	0.94	0.76	0.71	0.89	0.79
Juliet C/C++	0.72	0.99	0.44	0.61	0.65	0.60	0.89	0.72

*E. **RQ5:** LLMs vs Deep Learning-Based Tools*

We compare LLMs against two prior deep learning-based approaches: 1) DeepDFA [10], which trains Graph Neural Networks using embeddings of control flow graphs and associated data flow facts, and 2) LineVul [11], which is a transformed-based model trained using token-based representation of code.

Experiment setup. Our main aim in this experiment is to check the generalizability of these techniques on real-world datasets beyond the datasets they are trained on. We use CVEFixes C/C++ with the 1000 samples from our main evaluation §III-A as the real-world test set. We train on three different datasets:Juliet C/C++, CVEFixes C/C++ (excluding samples in the test set) and BigVul [34](the original C/C++ dataset that these models were trained on). We use an 80/20 train-validation split while training on these datasets. We use the DeepDFA and LineVul versions from DeepDFA's latest artifact [42]. Table IX presents the results, averaged across three runs.

```
// Code snippet prone to CWE-190 (Integer Overflow)
// data is incremented without verifying if data <
↪  INT_MAX

static const int STATIC_CONST_FIVE = 5;
void f7909()
{
    int64_t data;
    data = 0LL;
    if(STATIC_CONST_FIVE==5)
    {
        data = (int64_t) RAND64() ;
    }
    if(STATIC_CONST_FIVE==5)
    {
        {
            ++data;
            int64_t result = data;
            printLongLongLine (result);
        }
    }
}
```

GPT-4's response:
Unsanitized Data Flows ✓
The data from `RAND64()` is directly used in `printLongLongLine()` without any sanitization or checks for integer overflow.)

Vulnerability analysis verdict ✗
vulnerability: NO (CWE-190)

Explanation ✗
The code does not perform any arithmetic operations that could lead to an integer overflow. The only operation performed on the `data` variable is incrementing it by one `(++data)`, which is unlikely to cause an integer overflow given that data is a 64-bit integer.

Listing 3: CodeQL correctly predicts Integer Overflow while GPT-4 doesn't. GPT-4, however, correctly identifies the sources and sinks and the unsanitized paths.

Finding 5.1: DL-based approaches have limited effectiveness on real-world datasets. DeepDFA and LineVul trained on CVEFixes C/C++ training set report accuracies of 51% and 59% on CVEFixes C/C++ test set respectively while the best performing LLM on this dataset, Qwen-2.5-32B, reports an accuracy of 56%. This is in stark contrast to the performance of these techniques on BigVul where they report accuracies higher than 98%. Moreover, Qwen-2.5-32B reports an F1 score of 0.65 which is 0.42 and 0.04 higher than DeepDFA and LineVul respectively.

Finding 5.2: DL-based approaches do not generalize across datasets. When trained on Juliet C/C++ or BigVul, both DeepDFA and LineVul report accuracies and F1 scores lower

than Qwen-2.5-32B by upto 6% and 0.63 respectively.

Finding 5.3: LLMs are preferable over DL-based approaches due to low inference overheads and natural language explanations. DeepDFA involves significant inference overhead, due to the CFG extraction and dataflow analysis steps. LLMs, however, can use the textual representation of code and can operate on incomplete/partial programs. The use of data-flow and control-flow information in DeepDFA is evidently useful. We made similar observations with LLMs when using the CWE-DF prompt. On the other hand, LineVul, like LLMs, can leverage natural language information but has a generalization problem. Finally, both DeepDFA and LineVul provide binary labels and line numbers that are difficult to interpret. LLMs can additionally provide explanations, which are useful for further debugging (as shown in prior sections).

TABLE IX: Qwen-2.5-32B vs DeepDFA vs LineVul on CVEFixes C/C++

Model	Train/Prompt	Test	A	P	R	F1
DeepDFA	BigVul	BigVul	0.98	0.53	0.92	0.67
LineVul	BigVul	BigVul	0.99	0.96	0.88	0.92
Qwen-2.5-32B	CWE-DF	CVEFixes C/C++	0.56	0.54	0.81	0.65
DeepDFA	CVEFixes C/C++	CVEFixes C/C++	0.51	0.55	0.17	0.23
DeepDFA	Juliet C/C++	CVEFixes C/C++	0.53	0.53	0.65	0.58
DeepDFA	BigVul	CVEFixes C/C++	0.52	0.52	0.76	0.62
LineVul	CVEFixes C/C++	CVEFixes C/C++	0.59	0.58	0.65	0.61
LineVul	Juliet C/C++	CVEFixes C/C++	0.50	0.50	0.91	0.64
LineVul	BigVul	CVEFixes C/C++	0.50	0.63	0.01	0.02

IV. Related Work

Static analysis tools for vulnerability detection. Tools such as FlawFinder [43] and CppCheck [44] use syntactic and simple semantic analysis techniques to find vulnerabilities in C++ code. More advanced tools like CodeQL [5], Infer [45], and CodeChecker [46] employ semantic analysis techniques and can detect vulnerabilities in multiple languages. Static analysis tools rely on manually crafted rules and precise specifications of code behavior, which is difficult to obtain automatically. In contrast, while LLMs cannot always reliably perform end-to-end reasoning over code, we find that they are capable of automatically identifying these specifications which can be leveraged to improve static analysis-based detection tools.

Deep Learning-based vulnerability detection. Several works have focused on using Deep Learning techniques for vulnerability detection. Earlier works such as Devign [47], Reveal [48], LineVD [49] and IVDetect [50] leveraged Graph Neural Networks (GNNs) for modeling dataflow graphs, control flow graphs, abstract syntax trees and program dependency graphs. Other works explored alternate model architectures: VulDeePecker [51] and SySeVR [52] used LSTM-based models on slices and data dependencies while Draper used Convolutional Neural Networks. Recent works demonstrate that Transformer-based models fine-tuned on the task of vulnerability detection can outperform specialized techniques (CodeBERT, LineVul [11], UnixCoder). DeepDFA [10] and ContraFlow [53] learn specialized embeddings that can further improve the performance of Transformer-based vulnerability detection tools. These techniques, however, provide binary labels for vulnerability detection and do not provide natural language explanations.

LLMs for automated software engineering. Recent approaches have demonstrated the effectiveness of LLMs on software engineering tasks such as automated program repair [13]–[15], test generation [16], [17], code evolution [18], and fault localization [19]. However, unlike these approaches, we find that scaling LLMs to larger sizes does not improve vulnerability detection abilities. [54] explore whether Language Models fine-tuned on multi-class classification can perform well where the classes correspond to groups of similar types of vulnerabilities. In contrast, we study whether LLMs can perform a much granular CWE-level classification through prompting. Recent work combining LLMs with static analysis to detect Use Before Initialization (UBI) bugs in the Linux Kernel [55] supports our claims in Section III-D (but focuses on a specific class of bugs). There are other concurrent studies evaluating LLMs on vulnerability detection [24]–[28]. Table II provides a comparison against these studies. Section III-A corroborates the common finding from these studies that LLMs do not perform well on real-world datasets. However, to the best of our knowledge, our study is the first work that qualitatively identifies prompts and vulnerability classes where LLMs perform well (and often better than other tools). Moreover, our study focuses on a larger/different class of state-of-the-art LLMs, datasets, languages, and vulnerability classes.

V. Threats to Validity

External. The choice of LLMs and datasets may bias our evaluation and insights. To address this threat, we choose multiple popular synthetic and real-world datasets across two languages: Java and C++. We also choose both state-of-the-art closed-source and open-source LLMs. However, our insights may not generalize to other languages or datasets.

Internal. Owing to the non-deterministic nature of LLMs and single experiment runs per benchmark, our observations may be biased. To mitigate this threat, we use a temperature of 0 to ensure determinism across all LLMs. While this works well for locally run CodeLlama models, it is well-known that GPT-4 and GPT-3.5 might still return non-deterministic results. However, this should balance out across datasets and over the large number of benchmarks we evaluate on. Further, given similar results across LLMs on real-world-datasets, we do not expect significant changes with re-runs.

Our evaluation code and scripts may have bugs, which might bias our results. Our manual analysis of results may lead to erroneous inferences. To address this threat, multiple co-authors reviewed the code regularly and actively fixed issues. Further, multiple co-authors independently analyzed the results and discussed them together to mitigate any discrepancies.

VI. Conclusion

In this work, we performed a comprehensive analysis of LLMs for security vulnerability detection. Our study reveals that both closed-source LLMs, such as GPT-4, and open-source LLMs, like CodeLlama, perform modestly at vulnerability detection for both Java and C/C++. However, we find specific vulnerability classes where LLMs excel (often performing better than static analysis tools, such as CodeQL). Moreover, we find that even in cases where the models produce incorrect predictions, they are capable of identifying relevant sources, sinks and sanitizers that can be used for downstream dataflow analysis. Hence, we believe that an interesting future direction is to develop neuro-symbolic techniques that combine the intuitive reasoning abilities of LLMs with symbolic tools such as logical reasoning engines and static code analyzers for more effective and interpretable solutions.

REFERENCES

[1] 2022, https://nvd.nist.gov/vuln/detail/CVE-2022-3602.

[2] 2022, https://nvd.nist.gov/vuln/detail/CVE-2022-3786.

[3] M. Miller, "Microsoft: 70 percent of all security bugs are memory safety issues," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019.

[4] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[5] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *European Conference on Object-Oriented Programming*, 2016.

[6] Semgrep, "The semgrep platform," https://semgrep.dev/, 2023.

[7] Semmle, "Vulnerabilities discovered by CodeQL," https://securitylab.github.com/advisories/, 2023.

[8] L. Leong, "Mindshare: When mysql cluster encounters taint analysis," https://www.zerodayinitiative.com/blog/2022/2/10/mindshare-when-mysql-cluster-encounters-taint-analysis, 2022.

[9] GitHub, "The bug slayer," 2023, https://securitylab.github.com/bounties.

[10] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[11] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022.

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. A. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021.

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

[14] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

[15] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[16] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

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

[18] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 695–707.

[19] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[20] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. hsin Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Trans. Mach. Learn. Res.*, vol. 2022, 2022.

[21] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[22] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[23] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

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

[25] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, "How far have we gone in vulnerability detection using large language models," *arXiv preprint arXiv:2311.12420*, 2023.

[26] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le, "A comprehensive study of the capabilities of large language models for vulnerability detection," *arXiv preprint arXiv:2403.17218*, 2024.

[27] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 862–880. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00210

[28] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" *arXiv preprint arXiv:2403.18624*, 2024.

[29] 2023, https://owasp.org/www-project-benchmark.

[30] 2023, https://samate.nist.gov/SARD/test-suites/112.

[31] 2023, https://samate.nist.gov/SARD/test-suites/111.

[32] G. P. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021.

[33] T. Boland and P. E. Black, "Juliet 1.1 c/c++ and java test suite," *Computer*, 2012.

[34] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[35] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[36] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. A. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023.

[37] W. Wang, T. N. Nguyen, S. Wang, Y. Li, J. Zhang, and A. Yadavally, "Deepvd: Toward class-separation features for neural network vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[38] 2023, https://huggingface.co/.

[39] L. Salewski, S. Alaniz, I. Rio-Torto, E. Schulz, and Z. Akata, "In-context impersonation reveals large language models' strengths and biases," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[40] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[41] 2023. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[42] 2024, https://github.com/ISU-PAAL/DeepDFA/tree/master.

[43] 2023. [Online]. Available: https://dwheeler.com/flawfinder

[44] 2023, https://cppcheck.sourceforge.io/.

[45] 2023, https://fbinfer.com/.

[46] 2023, https://github.com/Ericsson/codechecker.

[47] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Neural Information Processing Systems*, 2019.

[48] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, pp. 3280–3296, 2020.

[49] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022.

[50] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[51] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: A deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[52] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 2244–2258, 2018.

[53] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[54] C. Thapa, S. I. Jang, M. E. Ahmed, S. A. Çamtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[55] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, 2024.

VII. Appendix

A. Dataset Processing and Selection

We perform a data processing and cleaning step for each dataset before evaluating them with LLMs.

OWASP. We remove or anonymize information in OWASP benchmarks that may provide obvious hints about the vulnerability in a file. For instance, we change package, variable names, and strings such as "owasp", "testcode", and "/sqli-06/BenchmarkTest02732" to other pre-selected un-identifying names such as "pcks", "csdr", etc. We remove all comments in the file because they may explicitly highlight the vulnerable line of code or may have irrelevant text (such as copyright info), which may leak information. These changes, however, do not change the semantics of the code snippets.

Juliet Java and C/C++. Similar to OWASP, we remove all comments and transform all identifiers that leak identifying information in all test cases in the Juliet benchmark. For instance, we change "class CWE80_XSS_CWE182_Servlet_connect_tcp_01" to "class MyClass". The Juliet benchmark provides the vulnerable (named as "bad") and non-vulnerable (named as "good*") methods in the same file. For easier evaluation, we perform a pre-processing step to split each file into two, each containing either a vulnerable or non-vulnerable method. Juliet also contains special benchmarks that have dependencies across multiple (2-5) files. We skip these benchmarks because they are typically too big to fit into the LLM prompt. Hence, the number of test cases after the data processing step in Juliet is reduced (as shown in Table VI).

CVEFixes. For each CVE, CVEFixes provides the methods that were involved in the fix commit. It also includes the method code in the parent commit, i.e., the method version before the fix. We collect all methods in the fix commit and the parent commit and label them as vulnerable and non-vulnerable, respectively. Similar to other datasets, we also remove all comments in the method code. While CVEFixes contains methods across multiple programming languages, we only collect C/C++ and Java methods for our study.

B. Prompting Strategies

The Basic prompt is presented in Listing 4, CWE specific prompt in Listing 5 and Dataflow analysis-based prompt in Listing 6.

```
System: You are a security researcher, expert in
detecting security vulnerabilities.
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type:
<CWE ID> | vulnerability name: <CWE NAME> |
explanation: <explanation for prediction>.
Use N/A in other fields if there are no
vulnerabilities. Do not include anything else in
response.

User: Is the following code snippet prone to any
security vulnerability?
<CODE_SNIPPET>
Response:
```

Listing 4: Basic LLM Prompt

```
System: [Same as above]
User: Is the following code snippet prone to <CWE>?
<CODE_SNIPPET>
Response:
```

Listing 5: CWE-specific LLM Prompt

```
System: You are a security researcher, expert in
detecting security vulnerabilities. Carefully
analyze the given code snippet and track the data
flows from various sources to sinks. Assume that
any call to an unknown external API is unsanitized.

Please provide a response only in the following
itemized OUTPUT FORMAT. Use N/A in other fields if
there are no vulnerabilities. DO NOT INCLUDE
ANYTHING ELSE IN YOUR RESPONSE.
<OUTPUT FORMAT>
Data flow analysis of the given code snippet:
1. Sources:
<numbered list of input sources>
2. Sinks:
<numbered list of output sinks>
3. Sanitizers:
<numbered list of sanitizers, if any>
4. Unsanitized Data Flows:
<numbered list of data flows that are not sanitized
in the format (source, sink, why this flow could be
vulnerable)>
5. Final Vulnerability analysis verdict:
vulnerability: <YES or NO> | vulnerability type:
<CWE_ID> | vulnerability name: <NAME_OF_CWE> |
explanation: <explanation for prediction>
</OUTPUT FORMAT>
User: Is the following code snippet prone to <CWE>?
<CODE_SNIPPET>
Response:
```

Listing 6: Dataflow analysis-based LLM Prompt

C. Other Prompting Strategies

In addition to the prompting strategies presented in our main evaluation, we considered other popular prompting strategies such as `Few-shot prompting` and `Chain-of-thought prompting` in a limited experimental setting. For the few-shot prompt (CWE-Few-shot), we included two examples of the task (one with a vulnerability and one without) in the CWE specific prompt before providing the target code snippet. For the chain-of-thought prompt (CWE-CoT), we explicitly ask the model to provide a reasoning chain before the final answer by adding a "Let's think step-by-step" statement at the end of the CWE specific prompt. The CWE-CoT and CWE-Few-shot prompts are provided in Listing 7 and Listing 8 respectively.

Table X and Table XI present the results from GPT-4 with various prompting strategies on a random subset of 100 samples of the Juliet Java and CVEFixes C/C++ datasets respectively. The CWE-DF prompt reports the highest accuracy of 69% and the highest F1 score of 0.75 on the Juliet Java dataset. The CWE-DF prompt reports a 0.05 higher F1 score

```
System: [Same as the Basic prompt]
User: Is the following code snippet prone to <CWE>?
<CODE_SNIPPET>
Let's think step by step.
Response:
```

Listing 7: CWE-CoT LLM Prompt

```
System: [Same as the Basic prompt]
User:

Query: Is the following code snippet prone to
<CWE1>?
Code snippet: <CODE_SNIPPET1>

Vulnerability analysis verdict: $ vulnerability:
YES | vulnerability type: <CWE1> . . .

Query: Is the following code snippet prone to
<CWE2>?
Code snippet: <CODE_SNIPPET2>

Vulnerability analysis verdict: $ vulnerability: NO
| vulnerability type: N/A . . .

Query: Is the following code snippet prone to <CWE>?
Code snippet: <CODE_SNIPPET>

Vulnerability analysis verdict:
```

Listing 8: CWE-Few-shot LLM Prompt

than the CWE-CoT prompt and a 0.03 higher F1 score than the CWE-Few-shot prompt. This difference is much more prominent on the CVEFixes C/C++ dataset where the CWE-DF prompt reports a 0.34 higher F1 score than the CWE-CoT prompt and a 0.31 higher F1 score than the CWE-Few-shot prompt. Moreover, the CWE-Few-shot prompt reported a 0.2 lower F1 score than the CWE specific prompt on the CVEFixes C/C++ dataset while requiring more tokens. Our analysis of the few-shot prompts suggests that providing more examples may not be a useful strategy for vulnerability detection. Because the potential set of vulnerable code patterns is quite large, the provided examples hardly make a difference to LLMs' reasoning abilities. Hence, it may be more useful to use prompts that instead elicit reasoning or explanations of some kind before detecting if the given snippet is vulnerable. The CWE-CoT prompt, however, does not help with reasoning always, as it either performed at par or worse than the Dataflow analysis-based prompt.

TABLE X: All prompting strategies on 100 samples from Juliet Java.

Model	Prompt	Metrics			
		A	P	R	F1
GPT-4	CWE	0.65	0.58	0.96	0.72
GPT-4	CWE-Few-shot	0.65	0.58	0.94	0.72
GPT-4	CWE-CoT	0.69	0.64	0.79	0.70
GPT-4	CWE-DF	**0.69**	0.61	0.96	**0.75**

Learning from these experiments, we selected the CWE specific prompt, Dataflow analysis-based prompt, in addition to the Basic prompt, for our main evaluation with LLMs.

TABLE XI: All prompting strategies on 100 samples from CVEFixes C/C++.

Model	Prompt	Metrics			
		A	P	R	F1
GPT-4	CWE	0.55	0.54	0.58	0.56
GPT-4	CWE-Few-shot	0.49	0.38	0.34	0.36
GPT-4	CWE-CoT	0.52	0.37	0.30	0.33
GPT-4	CWE-DF	**0.56**	0.56	0.83	**0.67**

D. Detailed metrics across all LLMs and Datasets

Table XII presents the metrics for all LLMs and Datasets across all prompts.

E. More examples of CodeQL vs. LLMs

```
// Code snippet prone to CWE-476 (NULL Pointer
↪  Dereference)
// Both the conditions are evaluated due to the
↪  bitwise & operator

if(globalReturnsTrue())
{
    twoIntsStruct *twoIntsStructPointer = NULL;
    if ((twoIntsStructPointer != NULL)
        &
        (twoIntsStructPointer->intOne == 5))
    {
        printLine("intOne == 5");
    }
}
```

GPT-4's response: The pointer `twoIntsStructPointer` is initialized as `NULL` and then checked if it's not `NULL` before dereferencing. However, the pointer is not assigned any value other than `NULL`, so it's not actually a risk of NULL pointer dereference. ✗

Listing 9: CodeQL correctly predicts Null Pointer Dereferencing while GPT-4 doesn't. GPT-4, however, correctly identifies the sources and sinks.

In Listing 9, the model incorrectly reasons about the sanitization by overlooking the & which would cause both the expressions to be evaluated. The identified sources and sinks are correct, however.

F. Qualitative analysis of GPT-4 responses

We first present examples where the dataflow analysis from the CWE-DF prompt is useful. Consider the code snippet in Listing 10. In this snippet, the variable `dir` is indirectly being used to create a directory via the `dirToCreate` variable. *GPT-4 correctly identifies that this path is not sanitized and could be used to create a directory in otherwise restricted locations*. This could lead to CWE-22 (path traversal) as

TABLE XII: Effectiveness of LLMs in Predicting Security Vulnerabilities (Java and C++). The highest accuracy and F1 scores (as well as ones within 0.1 range of the highest values) for each dataset are highlighted in blue.

Model	Prompt	OWASP				Juliet Java				CVEFixes Java				Juliet C/C++				CVEFixes C/C++			
		A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1
Qwen-2.5C-1.5B	Basic	0.50	0.50	0.82	0.62	0.50	0.50	0.99	0.66	0.49	0.49	0.68	0.57	0.49	0.50	0.99	0.66	0.51	0.51	0.78	0.61
Qwen-2.5C-1.5B	CWE	0.49	0.49	0.79	0.61	0.50	0.50	1.00	0.67	0.51	0.50	0.92	0.65	0.50	0.50	1.00	0.67	0.51	0.50	0.89	0.64
Qwen-2.5C-1.5B	CWE-DF	0.47	0.48	0.75	0.59	0.55	0.54	0.67	0.60	0.50	0.50	0.80	0.62	0.57	0.55	0.79	0.65	0.52	0.51	0.77	0.62
Qwen-2.5C-7B	Basic	0.50	0.50	1.00	0.67	0.50	0.50	0.99	0.67	0.47	0.48	0.79	0.60	0.50	0.50	1.00	0.67	0.50	0.50	0.95	**0.66**
Qwen-2.5C-7B	CWE	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.48	0.49	0.53	0.51	0.50	0.50	1.00	0.66	0.51	0.50	0.77	0.61
Qwen-2.5C-7B	CWE-DF	0.54	0.52	1.00	0.68	0.52	0.51	0.99	0.67	0.52	0.52	0.49	0.50	0.50	0.50	0.99	0.67	0.54	0.53	0.62	0.57
CodeLlama-7B	Basic	0.51	0.87	0.03	0.05	0.51	0.59	0.09	0.15	0.47	0.29	0.04	0.06	0.50	0.50	0.12	0.19	0.49	0.33	0.02	0.03
CodeLlama-7B	CWE	0.50	0.50	1.00	0.67	0.52	0.51	0.99	0.67	0.51	0.51	0.84	0.63	0.51	0.50	0.99	0.67	0.50	0.50	0.85	0.63
CodeLlama-7B	CWE-DF	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.50	0.50	1.00	**0.67**	0.50	0.50	1.00	0.67	0.51	0.50	0.97	**0.66**
DSCoder-7B	Basic	0.50	0.50	0.99	0.66	0.57	0.56	0.69	0.62	0.48	0.47	0.30	0.36	0.57	0.55	0.77	0.64	0.49	0.47	0.24	0.32
DSCoder-7B	CWE	0.56	0.54	0.87	0.66	0.54	0.53	0.75	0.62	0.48	0.43	0.15	0.22	0.58	0.56	0.70	0.62	0.51	0.53	0.18	0.27
DSCoder-7B	CWE-DF	0.51	0.50	0.98	0.66	0.52	0.51	0.91	0.65	0.49	0.50	0.90	0.64	0.50	0.50	0.98	0.66	0.53	0.52	0.90	**0.66**
Llama-3.1-8B	Basic	0.50	0.50	1.00	0.67	0.48	0.49	0.94	0.65	0.52	0.51	0.80	0.62	0.49	0.49	0.97	0.65	0.52	0.51	0.92	**0.66**
Llama-3.1-8B	CWE	0.53	0.52	1.00	0.68	0.52	0.51	0.97	0.67	0.53	0.56	0.29	0.38	0.54	0.52	0.98	0.68	**0.55**	0.55	0.58	0.56
Llama-3.1-8B	CWE-DF	0.49	0.50	0.93	0.65	0.50	0.50	0.97	0.66	0.51	0.50	0.93	0.65	0.50	0.50	0.99	0.67	0.50	0.50	0.95	0.65
CodeLlama-13B	Basic	**0.60**	0.58	0.74	0.65	0.48	0.48	0.41	0.44	0.50	0.51	0.08	0.14	0.47	0.47	0.51	0.49	0.50	0.50	0.07	0.12
CodeLlama-13B	CWE	0.52	0.51	0.98	0.67	0.50	0.50	0.89	0.64	0.48	0.47	0.29	0.36	0.53	0.51	0.98	0.67	0.53	0.52	0.56	0.54
CodeLlama-13B	CWE-DF	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.50	0.50	1.00	**0.67**	0.50	0.50	1.00	0.67	0.50	0.50	0.96	**0.66**
Qwen-2.5-14B	Basic	0.54	0.52	1.00	0.68	0.50	0.50	0.74	0.60	0.53	0.54	0.43	0.48	0.48	0.49	0.74	0.59	0.52	0.52	0.53	0.52
Qwen-2.5-14B	CWE	0.57	0.54	0.92	0.68	0.71	0.65	0.87	0.75	0.55	0.62	0.25	0.36	**0.65**	0.60	0.89	0.72	0.52	0.52	0.32	0.39
Qwen-2.5-14B	CWE-DF	0.55	0.52	1.00	0.69	0.66	0.61	0.88	0.72	0.56	0.58	0.42	0.49	**0.64**	0.59	0.95	**0.73**	**0.55**	0.56	0.45	0.50
DSCoder-15B	Basic	0.50	0.50	1.00	0.67	0.54	0.52	0.97	0.68	0.44	0.44	0.44	0.44	0.51	0.50	0.98	0.67	0.49	0.49	0.26	0.34
DSCoder-15B	CWE	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.52	0.51	0.93	**0.66**	0.50	0.50	1.00	0.67	0.50	0.50	0.95	**0.66**
DSCoder-15B	CWE-DF	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.51	0.51	0.86	0.64	0.50	0.50	1.00	0.67	0.51	0.50	0.94	0.66
codestral-22b	Basic	0.50	0.50	1.00	0.67	0.52	0.51	0.91	0.65	0.49	0.49	0.63	0.55	0.50	0.50	0.93	0.65	0.50	0.50	0.40	0.44
codestral-22b	CWE	0.52	0.51	0.98	0.67	0.52	0.51	0.96	0.67	0.50	0.50	0.37	0.43	0.57	0.54	0.93	0.69	0.52	0.56	0.16	0.25
codestral-22b	CWE-DF	0.53	0.52	1.00	0.68	0.50	0.50	0.99	0.67	0.53	0.52	0.89	0.65	0.50	0.50	0.99	0.67	0.52	0.51	0.87	0.64
Qwen-2.5-32B	Basic	0.52	0.51	1.00	0.67	0.48	0.49	0.77	0.60	0.52	0.53	0.38	0.44	0.50	0.50	0.84	0.63	0.47	0.46	0.36	0.41
Qwen-2.5-32B	CWE	0.56	0.53	1.00	**0.69**	0.58	0.55	0.93	0.69	0.53	0.55	0.30	0.39	0.63	0.58	0.87	0.70	0.53	0.54	0.35	0.43
Qwen-2.5-32B	CWE-DF	0.55	0.52	1.00	0.69	0.59	0.55	1.00	0.71	0.55	0.54	0.62	0.58	0.54	0.52	0.98	0.68	**0.56**	0.54	0.81	0.65
DSCoder-33B	Basic	0.52	0.51	0.97	0.67	0.56	0.53	0.94	0.68	0.50	0.50	0.60	0.55	0.42	0.46	0.81	0.58	0.51	0.51	0.75	0.60
DSCoder-33B	CWE	0.53	0.52	0.86	0.65	0.56	0.54	0.85	0.66	0.49	0.49	0.39	0.43	0.44	0.46	0.78	0.58	0.52	0.52	0.56	0.54
DSCoder-33B	CWE-DF	0.51	0.51	0.75	0.60	0.46	0.47	0.63	0.54	0.53	0.53	0.64	0.58	0.50	0.50	0.78	0.61	0.49	0.49	0.54	0.52
CodeLlama-34B	Basic	0.51	0.50	1.00	0.67	0.47	0.48	0.85	0.62	0.50	0.50	0.28	0.36	0.50	0.50	0.93	0.65	0.51	0.52	0.20	0.29
CodeLlama-34B	CWE	0.57	0.54	0.94	0.69	0.49	0.49	0.94	0.65	0.50	0.51	0.17	0.25	0.53	0.52	0.98	0.68	0.51	0.54	0.08	0.14
CodeLlama-34B	CWE-DF	0.50	0.50	1.00	0.67	0.50	0.50	1.00	0.67	0.50	0.50	1.00	**0.67**	0.50	0.50	1.00	0.67	0.50	0.50	0.99	**0.67**
Llama-3.1-70B	Basic	0.51	0.50	1.00	0.67	0.51	0.51	0.84	0.63	0.51	0.51	0.71	0.59	0.53	0.52	0.92	0.66	0.51	0.51	0.90	0.65
Llama-3.1-70B	CWE	0.58	0.54	0.99	**0.70**	**0.76**	0.71	0.89	**0.79**	0.52	0.53	0.43	0.48	0.59	0.55	0.95	0.70	0.52	0.51	0.71	0.60
Llama-3.1-70B	CWE-DF	0.54	0.52	0.99	0.68	0.72	0.68	0.84	0.75	0.55	0.54	0.63	0.58	0.59	0.55	0.96	0.70	0.54	0.53	0.77	0.63
Gemini-1.5-Flash	Basic	0.54	0.52	0.98	0.68	0.47	0.48	0.76	0.59	0.52	0.52	0.53	0.52	0.44	0.46	0.81	0.59	0.47	0.47	0.51	0.49
Gemini-1.5-Flash	CWE	0.57	0.54	1.00	**0.70**	0.51	0.51	0.91	0.65	0.54	0.57	0.31	0.40	0.50	0.50	0.89	0.64	0.51	0.51	0.52	0.51
Gemini-1.5-Flash	CWE-DF	0.54	0.52	1.00	0.68	0.50	0.50	1.00	0.67	**0.57**	0.55	0.79	0.65	0.50	0.50	0.99	0.66	0.51	0.50	0.86	0.64
GPT-3.5	Basic	0.52	0.52	0.72	0.60	0.58	0.57	0.71	0.63	0.46	0.35	0.09	0.15	0.49	0.49	0.64	0.56	0.52	0.56	0.20	0.29
GPT-3.5	CWE	0.55	0.54	0.62	0.58	0.52	0.52	0.55	0.54	0.47	0.41	0.12	0.19	0.49	0.49	0.70	0.58	0.52	0.54	0.19	0.28
GPT-3.5	CWE-DF	0.51	0.50	0.93	0.65	0.40	0.44	0.73	0.55	0.54	0.53	0.66	0.59	0.40	0.44	0.77	0.56	0.52	0.52	0.75	0.61
GPT-4	Basic	0.52	0.51	1.00	0.67	0.56	0.54	0.85	0.66	0.50	0.50	0.34	0.41	0.54	0.52	0.92	0.67	0.51	0.51	0.57	0.54
GPT-4	CWE	0.54	0.52	1.00	0.68	0.69	0.63	0.97	0.76	0.55	0.56	0.44	0.49	0.58	0.54	0.95	0.69	0.52	0.52	0.52	0.52
GPT-4	CWE-DF	0.55	0.52	1.00	0.69	0.70	0.63	0.98	0.76	0.53	0.53	0.59	0.56	0.59	0.55	0.98	0.70	0.52	0.51	0.76	0.61

is rightly predicted by GPT-4. In Listing 11, the method `write` takes an array `b` and integers `offset` and `length` as inputs. *The expression `offset + length > b.length` can lead to an Integer Overflow (CWE-190) if `offset + length` exceeds the maximum integer value* and this is correctly identified by GPT-4. Finally, in Listing 12, there are multiple array accesses that could lead to CWE-125 (Out-of-bounds read). *GPT-4 correctly identifies that the expression `ciphertextOffset + length + index` could exceed the size of the `ciphertext` array*, leading to this vulnerability.

```
// CVE-2021-39180

// CODE SNIPPET

public FileVisitResult preVisitDirectory(Path dir,
    BasicFileAttributes attrs) throws IOException {
    Path relativeDir = source.relativize(dir);
    final Path dirToCreate = Paths.get(destDir.
    toString(), relativeDir.toString());
    if(!dirToCreate.toFile().exists()) {
        Files.createDirectory(dirToCreate);
    }
    return FileVisitResult.CONTINUE;
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code
    snippet:
A. Sources:
1. The `dir` parameter in the `preVisitDirectory`
    method.
2. The `source` variable.
3. The `destDir` variable.

B. Sinks:
1. The `Files.createDirectory` method.

C. Sanitizers:
No sanitizers are present in the code snippet.

D. Unsanitized Data Flows:
1. (dir, Files.createDirectory, The `dir` parameter
     is used to create a new directory without any
    sanitization or validation. If an attacker can
    control the `dir` parameter, they could
    potentially create directories in unintended
    locations.)
2. (source, Files.createDirectory, The `source`
    variable is used to create a relative path
    without any sanitization or validation. If an
    attacker can control the `source` variable,
    they could potentially manipulate the path
    creation process.)
3. (destDir, Files.createDirectory, The `destDir`
    variable is used to create a new directory
    without any sanitization or validation. If an
    attacker can control the `destDir` variable,
    they could potentially create directories in
    unintended locations.)

E. Vulnerability analysis verdict: $$ vulnerability
    : YES | vulnerability type: CWE-22 |
    vulnerability name: Improper Limitation of a
    Pathname to a Restricted Directory ('Path
    Traversal') | explanation:
    The code does not validate or sanitize the
    'dir', 'source', and 'destDir' variables before
    using them to create a new directory. This
    could allow an attacker to manipulate the path
    and create directories in unintended locations,
     leading to a path traversal vulnerability. $$
```

Listing 10: Correct response by GPT-4 on a Path Traversal vulnerability

```
// TARGET CWE: CWE-190 (Integer Overflow or
    Wraparound)

// CODE SNIPPET
public void write(byte[] b, int offset, int length)
     throws IOException {
    if (b == null) {
      throw new NullPointerException();
    }

    if (offset < 0 || offset + length > b.length) {
      throw new ArrayIndexOutOfBoundsException();
    }

    write(fd, b, offset, length);
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code
    snippet:
A. Sources:
1. The byte array 'b'
2. The integer 'offset'
3. The integer 'length'

B. Sinks:
1. The 'write' function

C. Sanitizers:
1. The condition 'if (offset < 0 || offset + length
     > b.length)'

D. Unsanitized Data Flows:
1. (offset, write, The offset value is directly
    used in the write function without any
    sanitization)
2. (length, write, The length value is directly
    used in the write function without any
    sanitization)

E. Vulnerability analysis verdict: $$ vulnerability
    : YES | vulnerability type: CWE-190 |
    vulnerability name: Integer Overflow or
    Wraparound | explanation: The code does not
    properly handle the case where 'offset + length
    ' could overflow.
    If 'offset' and 'length' are both very large,
    their sum could wrap around to a negative value,
     passing the bounds check and leading to an
    ArrayIndexOutOfBoundsException at runtime. $$'
```

Listing 11: GPT-4 with CWE-DF prompt on a real-world Integer Overflow vulnerability

```
// TARGET CWE: CWE-125 (Out-of-bounds Read)

// CODE SNIPPET
public int encryptWithAd(byte[] ad, byte[]
    plaintext, int plaintextOffset,
            byte[] ciphertext, int ciphertextOffset
    , int length)
            throws ShortBufferException {
        int space;
        if (ciphertextOffset > ciphertext.length)
            space = 0;
        else
            space = ciphertext.length -
    ciphertextOffset;
        if (!haskey) {
                    if (length > space)
                throw new ShortBufferException();
            if (plaintext != ciphertext ||
    plaintextOffset != ciphertextOffset)
                System.arraycopy(plaintext,
    plaintextOffset, ciphertext, ciphertextOffset,
    length);
```

```
            return length;
        }
        if (space < 16 || length > (space - 16))
            throw new ShortBufferException();
        setup(ad);
        encryptCTR(plaintext, plaintextOffset,
    ciphertext, ciphertextOffset, length);
        ghash.update(ciphertext, ciphertextOffset,
    length);
        ghash.pad(ad != null ? ad.length : 0,
    length);
        ghash.finish(ciphertext, ciphertextOffset +
     length, 16);
        for (int index = 0; index < 16; ++index)
            ciphertext[ciphertextOffset + length +
    index] ^= hashKey[index];
        return length + 16;
    }

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code
    snippet:
A. Sources:
1. The `plaintext` array
2. The `ciphertext` array
3. The `ad` array
4. The `plaintextOffset` integer
5. The `ciphertextOffset` integer
6. The `length` integer

B. Sinks:
1. The `System.arraycopy` method
2. The `encryptCTR` method
3. The `ghash.update` method
4. The `ghash.pad` method
5. The `ghash.finish` method
6. The `ciphertext` array

C. Sanitizers:
1. The `if (ciphertextOffset > ciphertext.length)`
    check
2. The `if (length > space)` check
3. The `if (space < 16 || length > (space - 16))`
    check

D. Unsanitized Data Flows:
1. (plaintext, System.arraycopy, if plaintext is
    larger than the destination array, it could
    lead to an out-of-bounds read)
2. (ciphertext, ghash.finish, if ciphertextOffset +
     length + 16 is larger than the size of
    ciphertext, it could lead to an out-of-bounds
    read)

E. Vulnerability analysis verdict: vulnerability:
    YES | vulnerability type: CWE-125 |
    vulnerability name: Out-of-bounds Read |
    explanation: The code does not properly check
    the boundaries of the `ciphertext` array before
    reading from it, which could lead to an out-of-
    bounds read if `ciphertextOffset + length + 16`
     is larger than the size of the `ciphertext`
    array.
```

Listing 12: GPT-4 with CWE-DF prompt on a real-world Out of Bounds vulnerability

G. CWE-125 (Out-of-bounds Read in Java vs. C++)

Listing 12 presents an example of Out-of-bounds read vulnerability in CVEFixes Java which GPT-4 predicts correctly. Codellama-34B also correctly predicts that this snippet is vulnerable. Listing 13 presents an example from CVEFixes C/C++ where Out-of-bounds read vulnerability is not possible but this is not evident from context. Codellama-34B's explanation seems plausible given that little is known about the pointers in question.

```
// CODE SNIPPET NOT VULNERABLE TO CWE-125 (Out-of-
    bounds Read)
// The pointers cannot be reasoned about with this
    snippet alone

snmp_ber_decode_length(snmp_packet_t *snmp_packet,
    uint8_t *length)
{
  if(snmp_packet->used == 0) {
    return 0;
  }

  *length = *snmp_packet->in++;
  snmp_packet->used--;

  return 1;
}

// CODELLAMA-34B WITH THE DATAFLOW ANALYSIS BASED
    PROMPT

Here is a data flow analysis of the given code
    snippet:

A. Sources:

1. snmp_packet->in
2. snmp_packet->used

B. Sinks:

1. *length
2. snmp_packet->used

C. Sanitizers:

1. None

D. Unsanitized Data Flows:

1. (snmp_packet->in, *length, This flow could be
    vulnerable to Out-of-bounds Read (CWE-125) if
    the value of snmp_packet->in is not properly
    validated and could be pointing to an invalid
    memory location.)
2. (snmp_packet->used, snmp_packet->used--, This
    flow could be vulnerable to Out-of-bounds Read
    (CWE-125) if the value of snmp_packet->used is
    not properly validated and could be pointing to
     an invalid memory location.)

E. Vulnerability analysis verdict: $$ vulnerability
    : YES | vulnerability type: CWE-125 |
    vulnerability name: Out-of-bounds Read |
    explanation: The code snippet does not properly
     validate the input values, which could lead to
     an Out-of-bounds Read vulnerability. $$
```

Listing 13: Codellama-34B with CWE-DF prompt on a real-world Out of Bounds vulnerability (incorrect)