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 tools have made promising progress, their scalability and applicability remain challenging. Recently, Large Language Models (LLMs), such as GPT-4 and CodeLlama, have demonstrated remarkable performance on code-related tasks. However, it is unknown whether such LLMs can do complex reasoning over code. In this work, we explore whether pre-trained LLMs can detect security vulnerabilities and address the limitations of existing tools. We evaluate the effectiveness of pre-trained LLMs, in terms of performance, explainability, and robustness, on a set of five diverse security benchmarks spanning two languages, Java and C/C++, and covering both synthetic and real-world projects.

Overall, all LLMs show modest effectiveness in end-to-end reasoning about vulnerabilities, obtaining an average of 60% accuracy across all datasets. However, 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). Further, LLMs are relatively much better at detecting simpler vulnerabilities that typically only need local reasoning (e.g., Integer Overflows and NULL pointer dereference). We find that advanced prompting strategies that involve step-by-step analysis significantly improve performance of LLMs on real-world datasets (improving F1 score by up to 0.25 on average). Finally, we share our insights and recommendations for future work on leveraging LLMs for vulnerability detection.

1 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., [13] and [14]. A recent study by Microsoft showed that more than 70% of the vulnerabilities are still caused by well-understood memory safety issues [36]. The size and complexity of modern software systems are growing quickly, 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 [35], and static analyzers such as CodeQL [1] and Semgrep [41] have made promising strides. For example, in the last two years, researchers found over 300 security vulnerabilities through custom CodeQL queries [29, 42]. However, these techniques face challenges in scalability and applicability. Fuzzing does not scale to large applications, and fuzzing parts of applications requires manually crafting fuzz drivers. Moreover, it is hard to use on large critical programs with complex inputs, such as network servers, GUI-based programs, embedded firmware, boot loaders, and system services. On the other hand, static analysis relies heavily on manual API specifications, and skillfully crafted heuristics to balance precision and scalability. In light of these challenges, GitHub pays a bounty of over 7K USD for each CodeQL query that can find new critical security bugs [22].

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 [8], automated program repair [25, 48, 49], test generation [16, 28], code evolution [51], and fault localization [50]. 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 over previous generations in their capabilities—a phenomenon termed as *emergent behavior* [46]. For instance, GPT-4, which is presumably orders of magnitude larger than its 175-billion predecessor GPT-3.5, significantly outperforms GPT-3.5 on a wide range of code-understanding tasks [5]. Second, the diversity of LLMs has grown rapidly and now includes not only proprietary general-purpose ones such as GPT-4 but also open-sourced LLMs such as CodeLlama [39] and StarCoder [31] that are specialized for code. Finally, the reasoning capabilities of LLMs (and hence their applicability) may vary significantly across different programming languages. All these

*Equal contribution

factors open up a large exploration space for applying LLMs to the challenging task of vulnerability detection.

Our Work. We conduct the first comprehensive study of using LLMs for detecting security vulnerabilities. We study five state-of-the-art LLMs, including proprietary models such as GPT-3.5 and GPT-4, and open-source models like CodeLlama. We evaluate these models on five popular security vulnerability datasets.

We design a set of three prompting strategies for LLMs to elicit increasingly sophisticated forms of reasoning and explanations. 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).

A significant limitation of static vulnerability detection tools is the requirement of building the target project to enable them to find bugs. Further, they also require concrete specifications of APIs (e.g., sources, sanitizers, and sinks). In contrast, LLMs have an internal model of APIs already seen during the pre-training phase and hence they do not require compiled or complete codebases to run. Inspired by this insight, we additionally design a new prompting strategy, called *Dataflow analysis-based prompt*, that simulates a source-sink-sanitizer based dataflow analysis on the target code snippet before predicting if it is vulnerable. The Dataflow analysis-based prompt, similar to a classical dataflow-based static analyzer asks the LLM to first infer the sources, sinks, and sanitizers in the code snippet and check for any *unsanitized* data flows between the source and sink. This style of prompting is similar to chain-of-thought reasoning and simulates step-by-step reasoning by LLMs.

A key challenge in evaluating LLMs is data leakage: LLMs may perform well on a dataset because such samples were already present in their pre-training data. Hence, we implement three semantics-preserving adversarial attacks for code and evaluate whether they significantly impact the performance of LLMs.

Research Questions. We study the following research questions:

- **RQ1:** How do different pre-trained LLMs perform in detecting security vulnerabilities across different languages and datasets? (Section §3.1)
- **RQ2:** How do different prompting strategies affect the performance of LLMs? (Section 3.2)
- **RQ3:** How does the performance of LLMs vary across different vulnerability classes? (Section 3.3)
- **RQ4:** Can adversarial attacks impact the performance of LLMs in detecting vulnerabilities? (Section §3.4)
- **RQ5:** How do LLMs compare to state-of-the-art static analysis tools? (Section 3.5)
- **RQ6:** How do LLMs compare to state-of-the-art deep-learning-based tools? (Section 3.6)

Results and Findings. We choose five state-of-the-art LLMs: GPT-4, GPT-3.5, CodeLlama-34B, CodeLlama-13B, and CodeLlama-7B and evaluate their effectiveness in detecting security vulnerabilities across five vulnerability datasets: OWASP [38], Juliet Java [27], Juliet C/C++ [26], CVEFixes Java [2], and CVEFixes C/C++ [2]. These datasets cover 5000 vulnerable and non-vulnerable code samples, across 25 CWEs. Our findings are summarized as follows:

(1) LLMs across all sizes have modest vulnerability detection ability, with an average accuracy of only about 60% across all datasets.
(2) Using prompts that focus on detecting specific CWEs improves the performance of LLMs. The dataflow analysis-based prompt further improves results for larger LLMs by allowing step-by-step reasoning. The improvement is significant for real-world datasets (an increase of up to 0.25 F1 score). However, we observe that LLMs often infer the relevant source, sink, and sanitizers correctly but fail in end-to-end logical reasoning.
(3) LLMs are relatively better at detecting vulnerabilities that require local reasoning without complex code structures (such as Integer Overflow and NULL Pointer Dereference). However, they struggle to detect more complex vulnerabilities in real-world programs.
(4) Larger LLMs like GPT-4 show mild degradation in performance when subject to adversarial attacks on code (average drop 8.6%). In contrast, smaller LLMs, like CodeLlama-7B, show a significant drop in performance (average drop 39%) in the presence of perturbations.
(5) LLMs lag behind state-of-the-art static analysis tools like CodeQL in average accuracy (by 15%) and precision (by 26%) across datasets. However, LLMs like GPT-4 show some promising partial abilities, e.g., by identifying source/sink specifications or doing context-based reasoning using natural language information, which can potentially be useful for static analysis.
(6) We obtain mixed results comparing LLMs to prior deep-learning-based tools: while the deep-learning-based tools only slightly outperform LLMs on synthetic datasets (by 0.07 in F1 score), all approaches struggle equally on real-world datasets often producing close to 50% accuracy only (same as a random baseline).

Contributions. To summarize, we make the following contributions in this paper:

- **Empirical Study:** We conduct the first large comprehensive study on how state-of-the-art LLMs perform in detecting security vulnerabilities across five datasets and two programming languages (C/C++ and Java).
- **Prompting Strategies:** We design three prompting strategies for LLMs, inspired by the recent advances in natural language processing and traditional program analysis techniques, that elicit different reasoning styles from LLMs and also provide human-readable explanations for their predictions.
- **Robustness of LLMs:** We study how the performance of LLMs is impacted by semantics-preserving adversarial attacks on code.
- **Comparison with other vulnerability detection tools**: We contrast the performance of LLMs against popular static analysis and deep-learning-based vulnerability detection tools.
- **Insights:** We perform a rigorous manual analysis of LLMs' predictions and highlight vulnerability patterns that impact the performance of these models.

2 APPROACH

2.1 Datasets

For our study, we select five diverse vulnerability datasets from two languages: C++ and Java. For each language, we also select both synthetic and real-world benchmarks. Table 1 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. We describe each dataset next.

Table 1: Details of Selected Datasets

Dataset	Language	Size	Vul/Non-Vul	CWEs
OWASP[38]	Java	2740	1415/1325	11
SARD Juliet (C/C++) [26]	C/C++	81,280	40,640/40,640	118
SARD Juliet (Java) [27]	Java	35,940	17,970/17,970	112
CVEFixes [2]	C/C++	19,576	8223/11,347	131
CVEFixes [2]	Java	3926	1461/2465	68

2.1.1 *OWASP (Synthetic).*

The Open Web Application Security Project (OWASP) benchmark [38] 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. OWASP contains 2740 test cases representing 11 unique classes of security vulnerabilities (also known as Common Weakness Enumeration or CWE).

2.1.2 *Juliet (Synthetic).*

Juliet [4] is a widely-used vulnerability dataset developed by NIST. Juliet comprises thousands of synthetically generated test cases representing various known vulnerability patterns. It contains 81,280 C/C++ programs covering 118 unique CWEs, and 35,940 Java programs covering 112 unique CWEs. For our paper, we use the latest version, Juliet 1.3 [3, 26, 27].

2.1.3 *CVEFixes (Real-World).*

Bhandari et al. [2] 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. CVEFixes consists of methods extracted from 5495 vulnerability-fixing commits. These methods span multiple programming languages such as C/C++, Java, Python, and JavaScript. For our work, we extracted all C/C++ and Java methods from CVEFixes. We collected 19,576 C/C++ and 3926 Java methods (both vulnerable and non-vulnerable), covering 131 and 68 different CWEs, respectively.

While many real-world datasets have been proposed in the literature, we needed a dataset for our study 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 2 shows a comparison of existing real-world vulnerability datasets. We selected CVEFixes because it is the only dataset that fits our criteria.

Table 2: Comparison of Real-World Datasets

Dataset	Languages	CVE Metadata	Two-Sided	Multi-Lang
BigVul [17]	C/C++	✓	✗	✗
Reveal [7]	C/C++	✗	✗	✗
DiverseVul [9]	C/C++	✗	✓	✗
DeepVD [45]	C/C++	✗	✗	✗
CVEFixes [2]	C/C++, Java, ...	✓	✓	✓

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

2.3 Large Language Models

We choose the most popular state-of-the-art pre-trained Large Language Models (LLMs) for our evaluation. We choose OpenAI models: GPT-4 (`gpt-4`) and GPT-3.5 (`gpt-3.5-turbo`). GPT-3.5 allows up to 4096 input tokens while GPT-4 (which presumably is much larger) allows up to 8192 tokens in the input prompt. Since these models are closed-source, we also evaluate CodeLlama models [39], which were recently open-sourced by Meta. We select three versions of these models: CodeLlama-7B (CL-7B), CodeLlama-13B (CL-13B), and CodeLlama-34B (CL-34B), containing 7 billion, 13 billion, and 34 billion parameters, respectively. We use the Hugging Face APIs [24] to access CodeLlama models. We use the "Instruct" version of CodeLlama models—these models are fine-tuned to follow user instructions and hence can better adapt to specific reasoning tasks. Table 3 presents LLM details.

Table 3: Details of LLMs

Model Class	Model Version	Size	Context Window
GPT-4	`gpt-4-0613`	N/A	8k
GPT-3.5	`gpt-3.5-turbo-0613`	N/A	4k
CodeLlama-34B	`CodeLlama-34B-Instruct`	34B	16k
CodeLlama-13B	`CodeLlama-13B-Instruct`	13B	16k
CodeLlama-7B	`CodeLlama-7B-Instruct`	7B	16k

2.4 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* and the *user prompt*. 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 [40], 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 different prompting strategies to study the reasoning abilities of LLMs:

2.4.1 Basic prompt. We design a very simple prompt (shown in Listing 4 in the appendix) with the goal of understanding 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 4: Dataset Processing and Selection

	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

2.4.2 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. The `<CWE>` placeholder here contains both the ID of the CWE and the name. For instance, for CWE-22, the user prompt would start with "*Is the following code snippet prone to CWE-22 (Improper Limitation of a Pathname to a Restricted Directory (Path Traversal))?*" followed by the target code snippet.

2.4.3 Dataflow analysis-based prompt. In addition to the straightforward one-step analysis in the previous two prompts, we also study whether providing specific step-by-step analysis instructions can help the LLMs do better reasoning over code and make better predictions. 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 [47]. 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, compared to the other prompts, this prompt generates many more tokens and is hence more costly. We provide the full prompts in Appendix A.2.

2.4.4 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. Few-shot prompting reported poorer results than the base CWE specific prompt while requiring more tokens. Our analysis of the few-shot prompts suggests that providing more examples might not be a useful strategy for vulnerability detection. It might be more useful to use prompts that instead elicit reasoning or explanations of some kind before detecting if the given snippet is vulnerable. With Chain-of-thought prompting, 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. This setup did not yield better results than the Dataflow analysis-based prompt. Moreover, the reasoning chains obtained by Chain-of-thought prompting (both zero-shot and few-shot) were not as elaborate as those from the Dataflow analysis-based prompt thus limiting the ease of debugging. We provide the details of this experiment in Appendix A.3.

2.5 Dataset Processing and Selection

We perform a data processing and cleaning step for each dataset before evaluating them with LLMs. We remove or anonymize information such as commits, benchmark IDs, or vulnerability names that may provide obvious hints about the vulnerability. We skip benchmarks that are spread across multiple files, due to limitations of prompt size. Table 4 presents the details of our selection stages.

Due to the prohibitively large cost of running pre-trained LLMs, we select a subset of samples from the original datasets. We select samples corresponding to vulnerability types (or CWEs) listed in MITRE's Top 25 Most Dangerous Software Weaknesses [37]. Due to prompt size limitations, we filter out code snippets larger than 2048 tokens. Finally, because the datasets are quite large, we further randomly select 500 vulnerable and 500 non-vulnerable samples per dataset. Finally, we end up with 5000 samples across five datasets. We provide more details for each dataset in Appendix A.1.

2.6 Experimental Setup

Experiments with GPT-3.5 and GPT-4. We use the OpenAI public API to perform the experiments with GPT-3.5 and GPT-4. We use the `ChatCompletions` API endpoint to query the models with the prompts discussed in Section 2.4. 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. In all our experiments, we use the top-1 prediction from the models.

Experiments with CodeLlama. We run all CodeLlama experiments on two sets of machines: one with Intel Xeon machine, with 40 CPUs, four GeForce RTX 2080 Ti GPUs, and 750GB RAM, and another Intel Xeon machine with 48 CPUs, 8 A100s, and 1.5T RAM. Similar to GPT-4, we set the temperature to 0, the maximum tokens to 1024, and use top-1 prediction for evaluation.

3 RESULTS

3.1 RQ1: Effectiveness of LLMs

We evaluate the performance of pre-trained LLMs on five open-source datasets discussed in Section 2.1. presents the best accuracy and F1 scores (across prompts) of GPT-4, GPT-3.5, and CodeLlama models (CL-7B, CL-13B, CL-34B) on all datasets. The more detailed metrics for all prompts are presented in Appendix A.4.

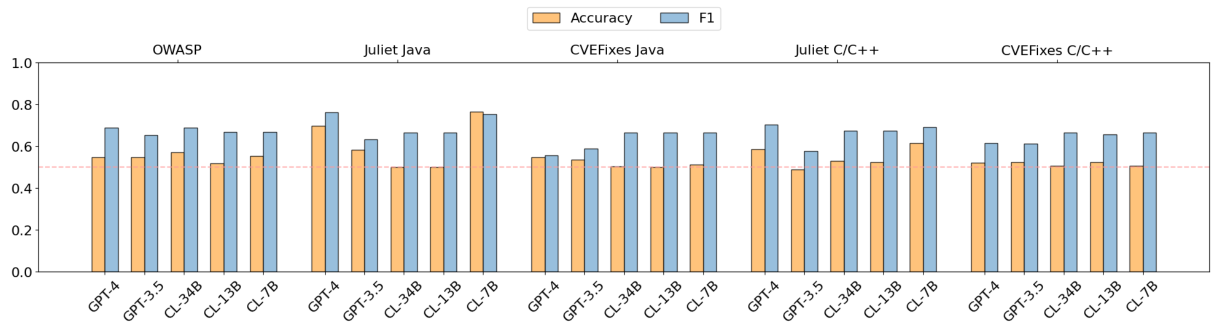


Figure 1: Effectiveness of LLMs in Predicting Security Vulnerabilities (Java and C++). We report the highest accuracy and F1 scores per model per dataset (across all prompting strategies).

Modest Vulnerability Detection Performance Across LLMs. The average of the highest accuracies across all datasets is 60.6% with the maximum being 77% for CL-7B (with Basic prompt) on the Juliet Java dataset. In terms of F1 score, GPT-4 (with CWE-DF) performs the best on the synthetic datasets: OWASP (F1: 0.69), Juliet Java (F1: 0.76), and Juliet C/C++ (F1: 0.70). We observe that CodeLlama models, despite being much smaller than GPT-4, are competitive on most datasets, even with the Basic prompt. CL-34B reports the highest F1 scores (F1: 0.67) on the real-world datasets, CVEFixes Java and CVEFixes C/C++, with the CWE-DF prompt.

Performance does not improve with scale. While GPT-4 performs strictly better than GPT-3.5 on 3 of the 5 datasets, the CodeLlama models roughly exhibit similar performance. Surprisingly, **CL-7B performs better than the larger LLMs on the Juliet Java and Juliet C/C++ datasets even with the Basic prompt**. Listing 1 presents a representative example where only CL-7B correctly predicts that an integer overflow vulnerability (CWE-190) cannot occur in the given context. Since the `data` variable can only contain value 2, adding 1 to it in Line 11 cannot cause an integer overflow. CL-7B recognizes this and predicts that the snippet is not vulnerable. CL-13B and GPT-4 (CWE-DF) ignore the hard-coded values and predict that the snippet is vulnerable, indicating that smaller LLMs may sometimes perform better than larger models.

Trends within model classes. The models here can be divided into two broad classes: the GPTx models (GPT-4 and GPT-3.5) and the CodeLlama models (CL-7B, CL-13B, CL-34B). The CodeLlama models display a higher divide between the accuracy and F1 scores than the GPTx models with F1 scores much higher than the accuracies (which are close to 50%). This is because the **CodeLlama models tend to almost always predict that the given code snippet is vulnerable** (as is evident with a high f1 score but lower accuracy on most datasets).

Finding 1: LLMs *across all sizes* have modest vulnerability detection ability, with an average accuracy of only about 60% across *all datasets.*

Implication 1: Beyond prompting, future techniques need to specialize general-purpose or code LLMs on high-quality vulnerability datasets to improve their performance.

```
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 the 'data' variable 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: CL-7B correctly predicts that this snippet is not vulnerable to Integer Overflow but GPT-4 and CL-13B do not.

3.2 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. CWE-DF performs better than CWE and Basic prompt in terms of F1 score on the real-world datasets (CVEFixes C/C++ and CVEFixes Java). We next highlight qualitative differences between various prompts:

The Basic prompt often detects incorrect vulnerability classes. Table 5 presents the percentage of samples predicted as vulnerable with the Basic prompt where the predicted CWE is correct, averaged over all datasets with the same programming language. We can observe that all models predict an incorrect vulnerability in > 60% and > 53% of all Java and C/C++ samples predicted as vulnerable respectively. This suggests that the Basic prompt detects incorrect CWEs in roughly half of the cases.

Table 5: Correct CWEs detected with the Basic prompt (%)

Language (Avg.)	GPT-4	GPT-3.5	CL-34B	CL-13B	CL-7B
Java	0.41	0.34	0.37	0.38	0.39
C/C++	0.29	0.31	0.33	0.35	0.47

Specifying the CWE in the prompt reduces false alarms. In 16 out of the 25 model-dataset combinations, the CWE specific prompt improves or retains both the accuracy and F1 score over the Basic prompt. GPT-4 with the CWE specific prompt on Juliet Java reports a 13% higher accuracy and a 0.1 higher F1 score than those with the Basic prompt. We manually inspect 10 vulnerable and 10 non-vulnerable samples from this dataset where the CWE specific prompt is correct and the Basic prompt is not. **We find that not including the CWE in the Basic prompt results in predictions discussing other incorrect CWEs.** The Basic prompt incorrectly predicts that the 10 non-vulnerable samples are vulnerable to other CWEs of which we only find 3 to be plausible but 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. Of the 10 vulnerable samples, Basic prompt predicts a plausible CWE for one sample while incorrectly predicting that the remaining 9 samples are not vulnerable. On the other hand, the CWE specific prompt 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.

Dataflow analysis identifies CWE-relevant textual cues and provides more elaborate explanations. The Dataflow analysis-based prompt (CWE-DF) prompt performs better than CWE specific prompt both in terms of accuracy and F1 score in 12 out of 25 model-dataset combinations and reports a higher F1 score in 23 out of 25

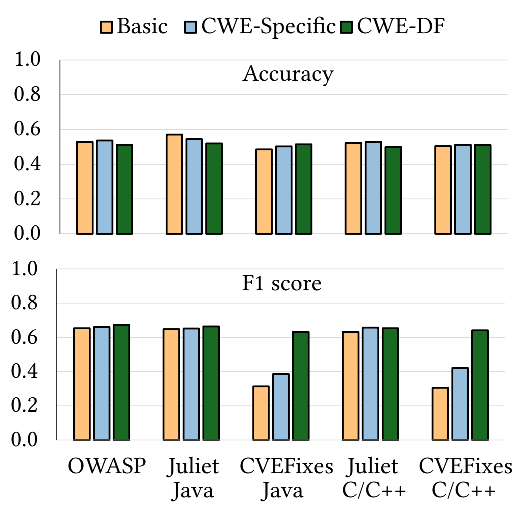


Figure 2: Performance of different prompting strategies

```
// 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);
}
```

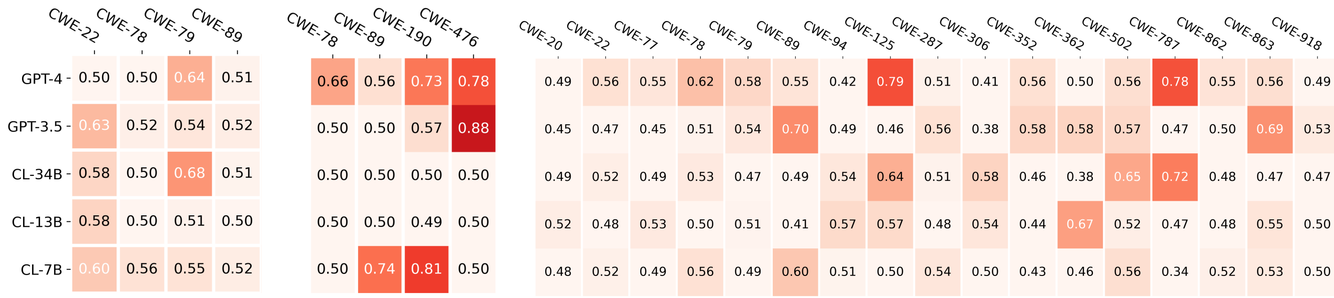
GPT-4's response: The data received from the socket connection is directly used in the _execvp function without proper sanitization. This could allow an attacker to inject malicious commands. ✓

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.

cases. The most significant improvement with the CWE-DF prompt over the CWE specific prompt is for the CVEFixes Java dataset and GPT-3.5 with 7% higher accuracy and 0.4 higher F1 score. We manually inspect 10 vulnerable and 10 non-vulnerable samples from this dataset where only the CWE-DF prompt is correct. **We find that the CWE-DF prompt leverages textual cues for sanitization (e.g., csrftokenhandler() suggests protection from CSRF) while the CWE specific prompt does not.** The CWE-DF prompt also assumes that any external parameter or call source is tainted by design and predicts that the sample is vulnerable more often (as is evident from higher F1). Further, **the analysis from CWE-DF prompt is more detailed and useful in localization than the explanations from CWE specific prompt**.

CWE specific prompt's explanations for 6 /10 vulnerable samples and all the 10 non-vulnerable samples were incorrect due to ignorance of CWE-specific elements and missed textual cues of sanitization or test code. For example, the CWE specific prompt considers a snippet vulnerable even when the request is wrapped inside an `XssHttpRequestWrapper` for sanitization. In contrast, the CWE-DF prompt provides correct explanations for 18 / 20 samples. Moreover, it 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 A.7.

LLMs often identify correct sources, sinks, and sanitizers but reason about them incorrectly. We manually inspect 10 vulnerable and 10 non-vulnerable samples for the Juliet C/C++ dataset where the predictions from CWE-DF are incorrect. In 17 / 20 samples, the explanation is incorrect due to erroneous reasoning about the snippet / false assumptions about the CWE. For example, an explanation incorrectly states that a char overflow is not



(a) OWASP **(b) Juliet Java** **(c) CVEFixes Java**

Figure 3: Accuracy Across CWEs on Java datasets.

vulnerable to CWE-190 (Integer Overflow or wraparound) since the wraparound is valid in C. Surprisingly, the sources/sinks are incorrect only in 2 samples and the sanitizers are incorrect only in 3 samples. The unsanitized flows are, however, incorrect in 12 samples suggesting that **the prompt is capable of identifying the sources/sinks/sanitizers accurately in most cases but fails to reason about the unsanitized flows and the target vulnerability.** Listing 3 presents an example where the vulnerability is not detected but the sources and sinks are correctly identified.

Finding 2: Specifying the CWE in the prompt reduces false positives based on other incorrect CWEs. Dataflow-based predictions provide mention accurate sources, sinks, and sanitizers even when the deductive logical reasoning is incorrect.

Implication 2: While step-by-step analysis improves performance, future approaches should combine LLMs with symbolic tools that can handle the logical reasoning aspects of analysis.

3.3 RQ3: Performance of LLMs across CWEs

We next evaluate how the LLMs perform on different classes of security vulnerabilities (CWEs). For each dataset and model, we consider the best-performing prompt for the analysis and only report CWEs with at least 10 samples. 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 in this section). Figure 3 presents the CWE-wise distribution of accuracies on the OWASP, Juliet Java, and CVEFixes Java datasets. Figure 4 reports the accuracies on the Juliet C/C++ and CVEFixes C/C++ datasets respectively. For each model, we consider the prompt with the best accuracy on the dataset and break ties using the F1 score.

LLMs perform well on vulnerabilities that do not require additional context. We find that there are certain CWEs where multiple LLMs perform better on: **Out-of-bounds Read / Write (CWE-125, CWE-787), Null pointer dereference (CWE-476) and Integer Overflow (CWE-190)**. 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. Concretely, GPT-3.5 consistently performs well on NULL Pointer Dereference across all datasets with accuracies of 80.8%, 87.9% and 60.5% on the Juliet C/C++, Juliet Java and CVEFixes C/C++ datasets respectively. GPT-4 also performs consistently better on NULL Pointer Dereference and Integer Overflow on the synthetic datasets with accuracies of 62.1% and 68.4% on Juliet C/C++ and 78.5% and 73% on Juliet Java respectively. CL-7B reports high accuracies on Integer Overflow on the synthetic datasets with accuracies of 81% on Juliet Java and 70% on Juliet C/C++ respectively. GPT-4 and CL-34B perform extremely well on Out-of-bounds Read (78.6% and 64.3%) and Out-of-bounds Write (78.1% and 71.9%) on the CVEFixes Java dataset.

(a) Juliet C/C++ **(b) CVEFixes C/C++**

Figure 4: Accuracy Across CWEs on C/C++ datasets.

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++. For instance, while GPT-4 and CL-34B 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. 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 A.8.

Some LLMs are better at detecting certain CWEs. GPT-4 reports high accuracies on CWE-78 (OS Command Injection) in 3/5 datasets (CVEFixes C/C++ with 70.8%, CVEFixes Java with 61.7%, and Juliet Java with 63.8%) but the other models do not. Among the CodeLlama models, CL-13B performs well on CWE-362 (Race Condition) in CVEFixes C/C++ with 63.1% and CVEFixes Java with 66.7%. CWE-89 (SQL Injection) is another vulnerability where only CL-7B reports high accuracies on 2/3 datasets (Juliet Java with 74% and CVEFixes Java with 64%).

Finding 3: LLMs perform better on vulnerabilities that only need reasoning about local entities (Out-of-bounds Read / Write, Integer Overflow, NULL pointer dereference). This locality can vary across languages: Array indexing in Java can be locally reasoned about while struct pointer accesses in C/C++ need additional context.

Implication 3: Techniques for language-specific and vulnerability-specific adaption of LLMs should be developed to improve their detection abilities.

3.4 RQ4: Impact of Adversarial Attacks

Experiment setup. We implement three semantics-preserving attacks, proposed by Gao et al. [21]: dead-code injection, variable renaming, and branch insertion (illustrated in Table 11 in appendix). We only select test samples where the original prediction of LLMs (with the CWE prompt) was correct. We select 100 test samples per dataset for this experiment and measure the reduction in accuracy after the attack. For this experiment, we select two models, GPT-4 and CL-7B, and four datasets. Figure 5 presents our results.

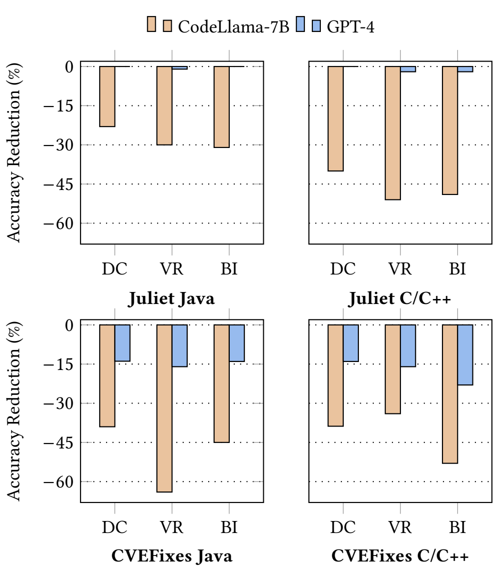


Figure 5: Accuracy degradation of CodeLlama-7B and GPT-4 under three kinds of adversarial attacks (DC: Deadcode insertion, VR: Variable Rename, BI: Branch Insertion). Lower values indicate more prone to attack.

Results. We find GPT-4 suffers mild degradation in accuracy (average 8.6%) across all datasets and attacks. For GPT-4, the degradation is more significant for real-world datasets (up to 23%) compared to synthetic ones (up to only 2%). In contrast, for CodeLlama-7B, the accuracy significantly reduces by almost 40-50% in many cases and by 39% on average, especially for C++.

Among the three attacks, branch insertion (27% avg. reduction) and variable renaming (23% avg. reduction) have the highest impact on LLM performance. On further analysis of incorrect predictions, we observe that under these attacks, GPT-4 either fails to infer any sources, sinks, or sanitizers (even if they exist) or the reasoning chain breaks (which worked previously). Interestingly, under these attacks, CodeLlama-7B fails to follow the instructions in the system prompt and does not perform proper analysis using the dataflow analysis steps, causing it to output an incorrect response.

Finding 4: More powerful LLMs like GPT-4 remain robust to adversarial attacks and only suffer mild degradation in performance. Whereas smaller LLMs like CodeLlama-7B show a significant drop in performance in the presence of perturbations. Overall, the impact of attacks is more pronounced for real-world datasets.

Implication 4: Developers of code LLMs should consider adversarial training methods during fine-tuning to ensure model robustness to such attacks.

3.5 RQ5: LLMs vs Static Analysis Tools

Experiment setup. We next explore how GPT-4 (CWE-DF) compares against CodeQL. Since CodeQL requires building projects before analysis and the real-world datasets contain large projects, we limit our focus to the three synthetic datasets, namely OWASP and Juliet Java and Juliet C/C++. In this study, we run the official CodeQL queries designed for the top 25 CWEs. Table 6 presents results from CodeQL and GPT-4 on the three datasets. **CodeQL performs better than GPT-4 (CWE-DF) in terms of accuracy on all three datasets.** GPT-4 reports a 0.09 higher F1 than CodeQL on Juliet C/C++ while CodeQL reports a 0.03 and 0.18 higher F1 on OWASP and Juliet Java respectively.

Table 6: GPT-4 vs CodeQL on synthetic datasets

Dataset/Model	CodeQL				GPT-4 (CWE-DF)			
	A	P	R	F1	A	P	R	F1
OWASP	0.63	0.58	0.96	0.72	0.55	0.53	1.00	0.69
Juliet Java	0.94	0.92	0.95	0.94	0.70	0.63	0.98	0.76
Juliet C/C++	0.72	0.99	0.44	0.61	0.59	0.55	0.98	0.70

We manually analyze some examples where CodeQL is correct and GPT-4 isn't and vice versa. As noted in Section 3.2, **we find that even when GPT-4 (with CWE-DF) provides an incorrect prediction, the identified sources / sinks and even unsanitized flows are often correct.** 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. The model correctly identifies the sources, sinks and even unsanitized

```
// 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.

dataflows in this case but fails to faithfully reason over them when predicting the vulnerability.

On the other hand, Listing 2 provides an example where GPT-4 correctly predicts that the snippet is prone to OS Command Injection while CodeQL does not. This happens because **CodeQL only looks for very specific cases of command injection**: We find that CodeQL only detects OS Command Injection for system commands that take a string of arguments (`execl`) but not for those that take a list of arguments (`_execvp`, used in this example). GPT-4, on the other hand, identifies the correct sources, sinks, and unsanitized flows and accurately reasons that the snippet is vulnerable. More examples are presented in the Appendix A.6.

> **Finding 5:** CodeQL performs better than GPT-4 on vulnerability detection across all CWEs. GPT-4 can, however, often accurately identify analysis-relevant sources and sinks that are missed by CodeQL's strict queries.

> **Implication 5:** Future techniques should combine LLMs' ability to infer relevant sources and sinks in code and its context understanding abilities to improve static analysis tools like CodeQL to obtain the best of both worlds.

3.6 RQ6: LLMs vs Deep-Learning-Based Tools

We compare LLMs against two prior deep learning-based approaches: 1) DeepDFA [43], which trains Graph Neural Networks using embeddings of control flow graphs and associated data flow facts, and 2) LineVul [20], which is a transformed-based model trained using token-based representation of code.

Experiment setup. We used the DeepDFA and LineVul versions from DeepDFA's latest artifact version [15]. For CVEFixes C/C++ and Juliet C/C++, we use the same test set as our main evaluation §3.1. We split the remaining dataset into training (80%) and validation (20%) sets. We also compare the results with the BigVul [17] dataset, used in the prior work. Because BigVul is a much larger real-world dataset, it can potentially improve the performance of the tools. We explore two evaluation settings: 1) train and test on the same dataset and 2) train on one and test on a different dataset to evaluate the generalizability of these models. Table 7 presents the results. Each cell reports the average results across three runs.

DL-based approaches have limited effectiveness on real-world datasets. We observe that both DeepDFA and LineVul obtain a maximum F1 score of 0.62 on CVEFixes C/C++, which is quite similar to the scores of GPT-4. For DeepDFA, while training on larger BigVul and Juliet C/C++ datasets improves its F1 scores, the accuracy scores remain close to 50%. On the other hand, LineVul is able to learn better even when using CVEFixes C/C++, which is relatively smaller. Interestingly, when trained on BigVul, LineVul obtains a poor F1 score of 0.02 on CVEFixes C/C++, which might indicate an over-fitting problem. We observe a similar pattern when LineVul is tested on Juliet C/C++ using CVEFixes C/C++ or BigVul training datasets. For Juliet C/C++, both tools obtain better scores compared to GPT-4. LineVul obtains a perfect score indicating that Juliet C/C++ has limited code patterns that are likely easier to learn. However, as we explain later, LineVul does not generalize well.

Generalizability across datasets. We observe that, for LineVul, the F1 scores drop drastically when trained on BigVul and tested on CVEFixes C/C++ or Juliet C/C++. Compared to LineVul, DeepDFA generalizes better, obtaining higher accuracy and F1 scores on Juliet C/C++ when trained using other datasets.

Trade-offs. 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).

Finding 6: Deep Learning-based tools, similar to LLMs, have poor effectiveness on vulnerability detection, especially when dealing with real-world datasets. Further, prior transformer-based approaches suffer from poor generalization across datasets.

Implication 6: Future techniques should explore a deeper combination of pre-trained LLMs with dataflow-style analysis to build more effective, interpretable, and general vulnerability detection tools.

Table 7: GPT-4 vs DeepDFA vs LineVul on CVEFixes C/C++ and Juliet 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
GPT-4	CWE-DF	CVEFixes C/C++	0.52	0.51	0.76	0.61
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
GPT-4	CWE-DF	Juliet C/C++	0.59	0.55	0.98	0.70
DeepDFA	Juliet C/C++	Juliet C/C++	0.77	0.74	0.82	0.78
DeepDFA	CVEFixes C/C++	Juliet C/C++	0.64	0.72	0.46	0.55
DeepDFA	BigVul	Juliet C/C++	0.74	0.70	0.82	0.77
LineVul	Juliet C/C++	Juliet C/C++	1.0	1.0	0.99	1.0
LineVul	CVEFixes C/C++	Juliet C/C++	0.51	0.62	0.14	0.22
LineVul	BigVul	Juliet C/C++	0.43	0.42	0.39	0.41

4 RELATED WORK

Static analysis tools for vulnerability detection. Static analysis tools search for pre-defined vulnerability patterns in code. Tools such as FlawFinder [19] and CppCheck [12] use syntactic and simple semantic analysis techniques to find vulnerabilities in C++ code. More advanced tools like CodeQL [1], Infer [18], and CodeChecker [11] 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 observe that LLMs can automatically identify such specifications by leveraging statistically learned rules from training data. Further, due to their semantic understanding of natural language, LLMs can also perform more contextual reasoning. These abilities can potentially be leveraged to improve static analysis tools.

Deep Learning-based vulnerability detection. Several works have focused on using Deep Learning techniques for vulnerability detection. Earlier works such as Devign [52], Reveal [6], LineVD [23] and IVDetect [32] 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 [33] and SySeVR [34] 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 [20], UnixCoder). DeepDFA [43] and ContraFlow [10] learn specialized embeddings that can further improve the performance of Transformer-based vulnerability detection tools. To the best of our knowledge, these techniques provide binary labels for vulnerability detection and cannot classify the type of vulnerability. Thapa et al. [44] 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 some of the largest Language Models, such as GPT-4, and perform a much granular CWE-level classification, generate human-readable informal specifications and explore various prompting techniques that allow using the LLMs out-of-the-box.

LLMs for automated software engineering. Several recent approaches have demonstrated that LLMs can be effectively leveraged to improve the state-of-the-art performance in various traditional software engineering tasks such as automated program repair [25, 48, 49], test generation [16, 28], code evolution [51], and fault localization [50]. However, unlike these approaches, we find that LLMs have limited vulnerability detection abilities.

Recently, Li et al. [30] developed Llift, an approach that combines LLMs with static analysis to detect Use Before Initialization (UBI) bugs in Linux kernel. While they focus on a specific class of bugs, their approach supports the observations we make in Section 3.5 on the complementary nature of static analysis and LLMs. To the best of our knowledge, our work is the first comprehensive and general study of vulnerability detection abilities of LLMs across a broad range of vulnerabilities (25 CWEs) across two languages.

5 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 not studied in this paper.

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 zero 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, due to the large number of benchmarks we evaluate, the non-determinism should balance out across the datasets. Further, given the poor effectiveness of all LLMs across the board, we do not expect our results to significantly change 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 of the paper independently analyzed the results and discussed them together to mitigate any discrepancies.

6 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++. Their performance on vulnerability detection is lower when dealing with real-world code. However, we find that even in cases where the models produce incorrect predictions, they identify relevant sources, sinks and sanitizers for 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] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *European Conference on Object-Oriented Programming*.

[2] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. 2021. 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).

[3] Paul E Black and Paul E Black. 2018. *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology.

[4] Tim Boland and Paul E Black. 2012. Juliet 1.1 C/C++ and Java test suite. *Computer* (2012).

[5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]

[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48 (2020), 3280–3296.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *ArXiv* abs/2107.03374 (2021).

[9] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (2023).

[10] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. 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).

[11] Code Checker 2023. https://github.com/Ericsson/codechecker.

[12] CPPCheck 2023. https://cppcheck.sourceforge.io/.

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

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

[15] DeepDFA artifact 2024. https://github.com/ISU-PAAL/DeepDFA/tree/master.

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 423–435.

[17] 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* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[18] Fb Infer 2023. https://fbinfer.com/.

[19] FlawFinder 2023. https://dwheeler.com/flawfinder

[20] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[21] Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete Adversarial Attack to Models of Code. 7, PLDI (2023). https://doi.org/10.1145/3591227

[22] GitHub. 2023. The Bug Slayer. https://securitylab.github.com/bounties.

[23] David Hin, Andrey Kan, Huaming Chen, and Muhammad Ali Babar. 2022. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)* (2022).

[24] Hugging Face 2023. https://huggingface.co/.

[25] Harshit Joshi, José Cambronero 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*.

[26] Juliet C/C++ 2023. https://samate.nist.gov/SARD/test-suites/112.

[27] Juliet Java 2023. https://samate.nist.gov/SARD/test-suites/111.

[28] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.

[29] Lucas Leong. 2022. Mindshare: When MySQL Cluster Encounters Taint Analysis. https://www.zerodayinitiative.com/blog/2022/2/10/mindshare-when-mysql-cluster-encounters-taint-analysis.

[30] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024).

[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[32] Yi Li, Shaohua Wang, and Tien Nhut Nguyen. 2021. 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).

[33] Zhuguo Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2020. VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing* (2020).

[34] Z. Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19 (2018), 2244–2258.

[35] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2018. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140* (2018).

[36] Matt Miller. 2019. 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/.

[37] MITRE Top 25 CWEs 2023. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[38] OWASP Benchmark Suite 2023. https://owasp.org/www-project-benchmark.

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[40] Leonard Salewski, Stephan Alaniz, Isabel Rio-Torto, Eric Schulz, and Zeynep Akata. 2024. In-Context Impersonation Reveals Large Language Models' Strengths and Biases. *Advances in Neural Information Processing Systems* 36 (2024).

[41] Semgrep. 2023. The Semgrep Platform. https://semgrep.dev/.

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

[43] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[44] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Ahmet Çamtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. *Proceedings of the 38th Annual Computer Security Applications Conference* (2022).

[45] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

[46] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed Huai hsin Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Trans. Mach. Learn. Res.* 2022 (2022).

[47] 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.

[48] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. 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*.

[49] Chunqiu Steven Xia and Lingming Zhang. 2022. 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*. 959–971.

[50] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[51] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. 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*. 695–707.

[52] Yaqin Zhou, Shangqing Liu, J. Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Neural Information Processing Systems*.

A APPENDIX

A.1 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 4).

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.

A.2 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

A.3 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.

Table 8 and Table 9 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 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 8: 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**

Table 9: All prompting strategies on 100 samples from CVE-Fixes 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**

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.

A.4 Detailed metrics across all LLMs and Datasets

Table 10 presents the metrics for all LLMs and Datasets across all prompts.

A.5 Adversarial Attack Examples

Table 11: Illustration of three different semantics-preserving adversarial attacks (in gray) on Java programs.

Attack	Example
Dead-code Injection	`int obj = new Object(...);` `if (obj == null) { /* deadcode */ }`
Variable Renaming	`String var3 = (String) names.nextElement();` `String[] values = request.getParameterValues(var3);`
Branch Insertion	`boolean var11 = true;` `if (var11) { configManager.init(); }`

Table 11 presents the three adversarial attacks that we implemented: dead-code injection, variable renaming, and branch insertion. Each attack contains multiple code patterns, which are randomly chosen when perturbing a given code snippet. We implement these attacks for Java and C/C++. For a given code snippet, we only apply an attack only once.

A.6 More Details on CodeQL vs. GPT-4

Table 12 presents the CWE-wise distribution of CodeQL results. CodeQL shows relatively better performance on Juliet Java (with an F1 score over 0.75 across CWEs) than Juliet C/C++ (with two CWEs reporting F1 scores under 0.4). On Juliet Java, CodeQL reports the highest performance on CWE-78 with an F1 score of 0.92. Interestingly, CWE-78 reports the worst results on the other two datasets (F1 score of 0.67 with OWASP and 0.03 with Juliet C/C++).

```
// 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 7: CodeQL correctly predicts Null Pointer Dereferencing while GPT-4 doesn't. GPT-4, however, correctly identifies the sources and sinks .

In Listing 7, 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.

A.7 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 8. 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 is rightly predicted by GPT-4. In Listing 9, 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 10, 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.

```
// 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();
```

Table 10: 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
GPT-4	Basic	0.52	0.51	1.00	0.68	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	0.50	0.57	0.54
GPT-4	CWE	0.54	0.52	1.00	**0.69**	0.69	0.63	0.97	**0.76**	**0.55**	0.56	0.44	0.50	0.58	0.54	0.95	0.69	**0.52**	0.52	0.52	0.52
GPT-4	CWE-DF	0.55	0.53	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
GPT-3.5	Basic	0.53	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**	0.55	0.19	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	**0.51**	0.54	0.19	0.28
GPT-3.5	CWE-DF	0.51	0.51	0.93	0.66	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
CL-34B	Basic	0.51	0.51	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	0.51	0.19	0.28
CL-34B	CWE	**0.57**	0.54	0.94	**0.69**	0.49	0.49	0.94	0.65	0.50	0.51	0.16	0.25	0.53	0.52	0.98	0.68	0.51	0.54	0.08	0.14
CL-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	1.00	**0.67**
CL-13B	Basic	0.52	0.51	0.98	0.67	0.47	0.48	0.72	0.58	0.50	0.50	0.13	0.21	0.46	0.48	0.79	0.59	0.51	0.52	0.22	0.31
CL-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	**0.52**	0.52	0.56	0.54
CL-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
CL-7B	Basic	0.56	0.54	0.81	0.65	**0.77**	0.79	0.72	0.75	0.47	0.46	0.42	0.44	**0.62**	0.58	0.86	0.69	0.47	0.35	0.07	0.11
CL-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	0.84	0.63
CL-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	0.98	**0.66**

Table 12: CodeQL vs GPT-4 (CWE-DF) CWE-wise comparison

Dataset	CWE	CodeQL				GPT-4 (CWE-DF)			
		A	P	R	F1	A	P	R	F1
OWASP	22	0.52	0.50	1.00	0.67	0.48	0.48	1.00	0.64
OWASP	78	0.56	0.53	0.77	0.63	0.48	0.48	1.00	0.65
OWASP	89	0.57	0.54	1.00	0.70	0.52	0.52	1.00	0.68
OWASP	79	0.79	0.71	1.00	0.83	0.65	0.59	1.00	0.74
Juliet Java	78	0.94	0.86	1.00	0.92	0.57	0.47	1.00	0.64
Juliet Java	89	0.88	0.85	1.00	0.92	0.69	0.68	1.00	0.81
Juliet Java	476	0.76	0.76	0.76	0.76	0.79	0.75	0.88	0.81
Juliet Java	190	0.97	1.00	0.92	0.96	0.71	0.62	0.97	0.75
Juliet C/C++	78	0.52	1.00	0.04	0.08	0.50	0.50	1.00	0.67
Juliet C/C++	476	0.80	0.95	0.75	0.84	0.71	0.75	0.86	0.80
Juliet C/C++	416	0.63	1.00	0.26	0.41	0.52	0.51	0.97	0.67
Juliet C/C++	190	0.93	0.99	0.87	0.93	0.68	0.60	0.98	0.74

```
    }

    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 9: 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
```

```
// 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 8: Correct response by GPT-4 on a Path Traversal vulnerability

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

A.8 CWE-125 (Out-of-bounds Read in Java vs. C++)

Listing 10 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 11 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 11: Codellama-34B with CWE-DF prompt on a real-world Out of Bounds vulnerability (incorrect)