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 Solk-Breslin  
University of Pennsylvania  
Philadelphia, USA  
alia@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-known vulnerabilities, e.g., [13] and [14].

Recent developments 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 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 CodexQ 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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Size</th>
<th>Vul/Non-Vul</th>
<th>CWEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWASP [38]</td>
<td>Java</td>
<td>2740</td>
<td>1415/1325</td>
<td>11</td>
</tr>
<tr>
<td>SARD Juliet (C/C++) [36]</td>
<td>C/C++</td>
<td>81,280</td>
<td>40,640/40,460</td>
<td>118</td>
</tr>
<tr>
<td>SARD Juliet (Java) [27]</td>
<td>Java</td>
<td>35,940</td>
<td>17,970/17,970</td>
<td>112</td>
</tr>
<tr>
<td>CVEFixes [2]</td>
<td>Java</td>
<td>3926</td>
<td>1461/2465</td>
<td>68</td>
</tr>
</tbody>
</table>

#### 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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Languages</th>
<th>CVE Metadata</th>
<th>Two-Sided</th>
<th>Multi-Lang</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigVol [17]</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reveal [7]</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DiverseVol [9]</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVEFixes [2]</td>
<td>C/C++, Java, ...</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 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: \[
\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Cases}}.
\]

- **Precision**: Precision represents what proportion of cases that a tool detects as a vulnerability is a correct detection. It is computed as: \[
\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}.
\]

- **Recall**: Recall represents what proportion of vulnerabilities the tool can detect. It is computed as: \[
\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}.
\]

- **F1 score**: The F1 score is a harmonic mean of precision and recall. It is computed as: \[
\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.
\]

#### 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

<table>
<thead>
<tr>
<th>Model Class</th>
<th>Model Version</th>
<th>Size</th>
<th>Context Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-4</td>
<td>gpt-4-0613</td>
<td>N/A</td>
<td>8k</td>
</tr>
<tr>
<td>GPT-3.5</td>
<td>gpt-3.5-turbo-0613</td>
<td>N/A</td>
<td>4k</td>
</tr>
<tr>
<td>CodeLlama-34B</td>
<td>CodeLlama-34B-Instruct</td>
<td>34B</td>
<td>16k</td>
</tr>
<tr>
<td>CodeLlama-13B</td>
<td>CodeLlama-13B-Instruct</td>
<td>13B</td>
<td>16k</td>
</tr>
<tr>
<td>CodeLlama-7B</td>
<td>CodeLlama-7B-Instruct</td>
<td>7B</td>
<td>16k</td>
</tr>
</tbody>
</table>

#### 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.

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 placeholder here contains both the ID of the CWE and the name. For instance, for CWE-22, the 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 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 CVEs) 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 determinstic 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.
Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities

**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 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.

**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>
<thead>
<tr>
<th>Language (Avg.)</th>
<th>GPT-4</th>
<th>GPT-3.5</th>
<th>CL-34B</th>
<th>CL-13B</th>
<th>CL-7B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>0.41</td>
<td>0.34</td>
<td>0.37</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td>C/C++</td>
<td>0.29</td>
<td>0.31</td>
<td>0.33</td>
<td>0.35</td>
<td>0.47</td>
</tr>
</tbody>
</table>

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 adding 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 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
Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities

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

Figure 3: Accuracy Across CWEs on Java datasets.

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.

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 adoption 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](image)

**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](image)

**Table 6:** GPT-4 vs CodeQL on synthetic datasets

<table>
<thead>
<tr>
<th>Dataset/Model</th>
<th>CodeQL</th>
<th>GPT-4 (CWE-DF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>OWASP</td>
<td>0.63</td>
<td>0.58</td>
</tr>
<tr>
<td>Juliet Java</td>
<td>0.94</td>
<td>0.92</td>
</tr>
<tr>
<td>Juliet C/C++</td>
<td>0.72</td>
<td>0.99</td>
</tr>
</tbody>
</table>

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 un sanitized...
flows and accurately reasons that the snippet is vulnerable. More the other hand, identifies the correct sources, sinks, and unsanitized
mands that take a string of arguments (that CodeQL only detects OS Command Injection for system com-
looks for very specific cases of command injection
This happens because CodeQL only
predicts that the snippet is prone to OS Command Injection while
vulnerability: NO (CWE-190)

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: 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.

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).
### 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] andCppCheck [12] use syntactic and simple semantic analysis techniques to find vulnerabilities in C++ code. More advanced tools like CodeQL [1], Infer [18], andCodeChecker [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] andIVDetect [32] leveraged Graph Neural Networks (GNNs) for modeling dataflow graphs, control flow graphs, abstract syntax trees and program dependency graphs. Other works explored alternative model architectures: VulDeePecker [33] andSySeVR [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] andContraFlow [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], andfault localization [50]. However, unlike these approaches, we find that LLMs have limited vulnerability detection abilities.

Recently, Li et al. [30] developed Lift, 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 runCodeLlama 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.

---

**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++**

<table>
<thead>
<tr>
<th>Model</th>
<th>Train/Prompt</th>
<th>Test</th>
<th>A</th>
<th>P</th>
<th>R</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepDFA</td>
<td>BigVul</td>
<td>BigVul</td>
<td>0.98</td>
<td>0.53</td>
<td>0.92</td>
<td>0.67</td>
</tr>
<tr>
<td>LineVul</td>
<td>BigVul</td>
<td>BigVul</td>
<td>0.99</td>
<td>0.96</td>
<td>0.88</td>
<td>0.92</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-DF</td>
<td>CVEFixes C/C++</td>
<td>0.52</td>
<td>0.51</td>
<td>0.76</td>
<td>0.61</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>CVEFixes C/C++</td>
<td>CVEFixes C/C++</td>
<td>0.51</td>
<td>0.55</td>
<td>0.17</td>
<td>0.23</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>Juliet C/C++</td>
<td>CVEFixes C/C++</td>
<td>0.53</td>
<td>0.53</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>BigVul</td>
<td>CVEFixes C/C++</td>
<td>0.52</td>
<td>0.52</td>
<td>0.76</td>
<td>0.62</td>
</tr>
<tr>
<td>LineVul</td>
<td>CVEFixes C/C++</td>
<td>CVEFixes C/C++</td>
<td>0.59</td>
<td>0.58</td>
<td>0.65</td>
<td>0.62</td>
</tr>
<tr>
<td>LineVul</td>
<td>Juliet C/C++</td>
<td>CVEFixes C/C++</td>
<td>0.50</td>
<td>0.50</td>
<td>0.91</td>
<td>0.64</td>
</tr>
<tr>
<td>LineVul</td>
<td>BigVul</td>
<td>CVEFixes C/C++</td>
<td>0.50</td>
<td>0.63</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-DF</td>
<td>Juliet C/C++</td>
<td>0.59</td>
<td>0.55</td>
<td>0.98</td>
<td>0.70</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>Juliet C/C++</td>
<td>Juliet C/C++</td>
<td>0.77</td>
<td>0.74</td>
<td>0.82</td>
<td>0.78</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>CVEFixes C/C++</td>
<td>Juliet C/C++</td>
<td>0.64</td>
<td>0.72</td>
<td>0.46</td>
<td>0.55</td>
</tr>
<tr>
<td>DeepDFA</td>
<td>BigVul</td>
<td>Juliet C/C++</td>
<td>0.74</td>
<td>0.70</td>
<td>0.82</td>
<td>0.77</td>
</tr>
<tr>
<td>LineVul</td>
<td>Juliet C/C++</td>
<td>Juliet C/C++</td>
<td>1.0</td>
<td>1.0</td>
<td>0.99</td>
<td>1.0</td>
</tr>
<tr>
<td>LineVul</td>
<td>CVEFixes C/C++</td>
<td>Juliet C/C++</td>
<td>0.51</td>
<td>0.62</td>
<td>0.14</td>
<td>0.22</td>
</tr>
<tr>
<td>LineVul</td>
<td>BigVul</td>
<td>Juliet C/C++</td>
<td>0.43</td>
<td>0.42</td>
<td>0.39</td>
<td>0.41</td>
</tr>
</tbody>
</table>

---

**Avishree Khare**, **Saikat Dutta**, **Ziyang Li**, **Alaia Solko-Breslin**, **Rajeev Alur**, and **Mayur Naik**
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


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. 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>
<thead>
<tr>
<th>Listing 5: CWE-specific LLM Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System:</strong> 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.</td>
</tr>
<tr>
<td><strong>User:</strong> Is the following code snippet prone to &lt;CWE&gt;?</td>
</tr>
<tr>
<td><strong>&lt;CODE_SNIPPET&gt;</strong></td>
</tr>
<tr>
<td><strong>Response:</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 6: Dataflow analysis-based LLM Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System:</strong> 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.</td>
</tr>
<tr>
<td><strong>User:</strong> Is the following code snippet prone to &lt;CWE&gt;?</td>
</tr>
<tr>
<td><strong>&lt;CODE_SNIPPET&gt;</strong></td>
</tr>
<tr>
<td><strong>Response:</strong></td>
</tr>
</tbody>
</table>

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: All prompting strategies on 100 samples from Juliet Java.

<table>
<thead>
<tr>
<th>Model</th>
<th>Prompt</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE</td>
<td>0.65</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-Few-shot</td>
<td>0.65</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-CoT</td>
<td>0.69</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-DF</td>
<td>0.69</td>
</tr>
</tbody>
</table>

Table 9: All prompting strategies on 100 samples from CVE-Fixes C/C++.

<table>
<thead>
<tr>
<th>Model</th>
<th>Prompt</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE</td>
<td>0.55</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-Few-shot</td>
<td>0.49</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-CoT</td>
<td>0.52</td>
</tr>
<tr>
<td>GPT-4</td>
<td>CWE-DF</td>
<td>0.56</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dead-code Injection</td>
<td>int obj = new Object(...); if (obj != null) {...}</td>
</tr>
<tr>
<td>Variable Renaming</td>
<td>String var3 = (String) names.nextElement(); String[] values = request.getParameterValues(var3);</td>
</tr>
<tr>
<td>Branch Insertion</td>
<td>boolean var11 = true; if (var11) { configManager.init(); }</td>
</tr>
</tbody>
</table>

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 Dereferenc- ing) // 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 Dereferenc- ing 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();
    }
}
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. $$
\]

```java
write(fd, b, offset, length);
```

Listing 9: GPT-4 with CWE-DF prompt on a real-world Integer Overflow vulnerability

```java
// 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; // space = ciphertext.length - ciphertextOffset;
    if (space < 16 || length > (space - 16))
        throw new ShortBufferException();
    else
        ghash.update(ciphertext, ciphertextOffset, length);
    System.arraycopy(plaintext, plaintextOffset, ciphertext, ciphertextOffset, length);
    return length;
}
```

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

```java
if (plaintext != ciphertext || plaintextOffset != ciphertextOffset)
    throw new ShortBufferException();
```

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:

A. Functions:
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)
The code does not properly check for path traversal vulnerabilities. This could allow an attacker 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.

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 `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: 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.

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

A. Sources:
1. `snmp_packet->in`
2. `snmp_packet->used`

B. Sinks:
1. `snmp_packet->in`
2. `snmp_packet->used`

C. Sanitizers:
1. None

D. Unsanitized Data Flows:
1. `(snmp_packet->in, 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.
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: $\textit{vulnerability: YES | vulnerability type: CWE-125 | vulnerability name: Out-of-bounds Read | explanation: The code snippet does not properly validate the input values, which could lead to an Out-of-bounds Read vulnerability.}$

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