Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities

AVISHREE KHARE*, University of Pennsylvania, USA
SAIKAT DUTTA*, Cornell University, USA
ZIYANG LI, University of Pennsylvania, USA
ALAIA SOLKO-BRESLIN, University of Pennsylvania, USA
RAJEEV ALUR, University of Pennsylvania, USA
MAYUR NAIK, University of Pennsylvania, USA

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 on a set of five diverse security benchmarks spanning two languages, Java and C/C++, and including code samples from synthetic and real-world projects. We evaluate the effectiveness of LLMs in terms of their performance, explainability, and robustness.

By designing a series of effective prompting strategies, we obtain the best results on the synthetic datasets with GPT-4: F1 scores of 0.79 on OWASP, 0.86 on Juliet Java, and 0.89 on Juliet C/C++. Expectedly, the performance of LLMs drops on the more challenging real-world datasets: CVEFixes Java and CVEFixes C/C++, with GPT-4 reporting F1 scores of 0.48 and 0.62, respectively. We show that LLMs can often perform better than existing static analysis and deep learning-based vulnerability detection tools, especially for certain classes of vulnerabilities. Moreover, LLMs also often provide reliable explanations, precisely identifying the vulnerable data flows in code. We find that fine-tuning smaller LLMs (e.g., GPT-3.5) can outperform the larger LLMs on synthetic datasets, but provide limited gains on real-world datasets. When subjected to adversarial attacks on code (e.g., dead code injection), LLMs suffer mild degradation in performance, showing an average accuracy reduction of up to 12.67%. 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., [CVE-2022-3602 2022] and [CVE-2022-3786 2022]. A recent study by Microsoft showed that more than 70% of the vulnerabilities are still well-understood memory safety issues [Miller 2019]. 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 intricate software.

Traditional techniques for automated vulnerability detection, encompassing fuzzers [Manes et al. 2018] and static analyzers such as CodeQL [Avgustinov et al. 2016] and Semgrep [Semgrep 2023] have made promising strides. For example, in the last two years, researchers found over 300 security vulnerabilities through custom CodeQL queries [Leong 2022; Semmle 2023]. 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

[*] These authors contributed equally to this work

programs, embedded firmware, boot loaders, and system services. On the other hand, static analysis relies heavily on custom 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 [GitHub 2023].

Large Language Models (LLMs), including pre-trained models such as GPT-4 and Llama-2, have made remarkable advances on code-related tasks just in the past year. Such tasks include code completion [Chen et al. 2021], automated program repair [Joshi et al. 2023; Xia et al. 2023; Xia and Zhang 2022], test generation [Deng et al. 2023; Lemieux et al. 2023], code evolution [Zhang et al. 2023], and fault localization [Yang et al. 2023]. These results clearly show that LLMs can significantly outperform traditional analyses and tools, 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.

A first step to developing LLM-based solutions to find *unknown* vulnerabilities is a systematic exploration of how LLMs perform at finding *known* vulnerabilities. This is especially important in light of the rapidly evolving landscape of LLMs in three aspects: *scale*, *diversity*, and *adaptability*. 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* [Wei et al. 2022a]. For instance, GPT-4, which is presumably orders of magnitude larger than its 175-billion parameter predecessor GPT-3.5, significantly outperforms GPT-3.5 on a wide range of code-understanding tasks [Bubeck et al. 2023]. Second, the diversity of LLMs has grown rapidly, and now includes not only proprietary ones such as GPT-4 but also open-sourced ones such as CodeLlama [Rozière et al. 2023] and StarCoder [Li et al. 2023]. Finally, the number of techniques for adapting LLMs to solve complex tasks is also growing quickly, and includes prompting techniques to elicit reasoning behaviors from LLMs [Wei et al. 2022b; Yao et al. 2022], as well as fine-tuning to boost their performance on specific tasks. These recent improvements open up a large exploration space for applying LLMs to the challenging task of vulnerability detection.

Our Work. In this work, we conduct the first comprehensive study of using LLMs for detecting security vulnerabilities. To this end, we consider a suite of the state-of-the-art LLMs, including the proprietary models GPT-4 and GPT-3.5, and the open-source CodeLlama models of varying sizes of 7 billion and 13 billion parameters. We evaluate these models on five significant security vulnerability datasets. The datasets span two languages, Java and C/C++, and include both synthetic (OWASP, Juliet Java, and Juliet C/C++) and real-world (CVEFixes Java and CVEFixes C/C++) benchmarks.

We design a set of four prompting strategies for LLMs to elicit increasingly sophisticated forms of reasoning and explanations. Our simplest prompting strategies include the *Basic prompt*, which simply asks the LLMs 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 (such as Buffer Overflows).

A significant limitation of static analysis tools like CodeQL 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, like 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 also makes the predictions more *interpretable*, allowing us to look at the reasoning chains produced by LLMs before trusting the prediction.

Furthermore, previous work has shown that subsequently prompting LLMs to validate their predictions using intrinsic or extrinsic feedback can be an effective strategy to constrain inaccuracies. This prompting strategy falls into the category of *Self Reflection* [Shinn et al. 2023] – a group of prompting styles that can help large models like GPT-4 self-correct its predictions. We evaluate a simplistic variant of Self Reflection to intrinsically validate the results of the Dataflow analysis-based prompt. We name this prompt as *Dataflow analysis-based prompt with Self Reflection*. We observe that it shows early signs of being an effective strategy to prune false positive predictions.

We compare the LLM prompting-based approaches against two groups of existing approaches: Static Analysis-based tools (CodeQL [Avgustinov et al. 2016]) and Deep Learning-based tools for Vulnerability Detection (LineVul [Fu and Tantithamthavorn 2022]). CodeQL is a state-of-the-art static analyzer comprising a suite of rule-based checkers for different classes of vulnerabilities. LineVul is a transformer-based vulnerability detection model derived by fine-tuning the pre-trained code model, CodeBERT, on a large real-world C/C++ dataset.

Confirming security vulnerabilities as real or false is a tedious and error-prone task. We therefore also evaluate the interpretability of the different approaches. Static analysis tools provide a reasoning trace, whereas existing deep-learning-based tools like LineVul only flag lines based on attention scores. Our prompting-based approach attempts to achieve the best of both worlds with the Dataflow analysis-based prompt because it can not only leverage patterns inferred from data like LineVul but also provide human-understandable vulnerability specifications like CodeQL.

A key challenge in understanding the true performance of LLMs is data leakage: LLMs may perform well on a dataset because such samples were present in their pre-training data. Hence, we implement three semantics-preserving adversarial attacks for code and evaluate whether they significantly degrade the performance of LLMs. Finally, we also test the *adaptability* of LLMs by testing whether fine-tuning them on a small portion of the datasets improves their performance.

Results. We choose three state-of-the-art LLMs: GPT-4, CodeLlama-13B, and CodeLlama-7B and evaluate their effectiveness in detecting security vulnerabilities across five vulnerability datasets: OWASP [OWASP Benchmark Suite 2023], Juliet Java [Juliet Java 2023], Juliet C/C++ [Juliet C/C++ 2023], CVEFixes Java [Bhandari et al. 2021], and CVEFixes C/C++ [Bhandari et al. 2021]. For synthetic datasets, we obtain the best results with GPT-4 (with Dataflow analysis-based prompt with Self Reflection prompt): an F1 score of 0.79 on OWASP, 0.86 on Juliet Java, and 0.89 on Juliet C/C++. With CodeLlama models, we obtain lower F1 scores of 0.69, 0.77, and 0.66 respectively. Interestingly, we do not observe any significant difference between CodeLlama 7B and 13B for synthetic datasets.

For real-world datasets, GPT-4 obtains up to 0.48 and 0.62 F1 scores for CVEFixes Java and CVEFixes C/C++ respectively, both with the Dataflow analysis-based prompt. We find that Self Reflection tends to drive GPT-4's predictions towards "not vulnerable" for these datasets, which drastically reduces its performance. On further analysis, we find that this often happens when the snippet includes references to several non-local components and the model's response is to ask for more context. Interestingly, we observe that CodeLlama performs slightly better on these datasets, achieving 0.55 F1 scores for CVEFixes Java (both models) and 0.65 for CVEFixes C/C++ (with CodeLlama-7B).

We attribute the difference in performance between the synthetic and real-world datasets mainly to the fact that unlike the synthetic samples, the real-world samples are more diverse and not self-contained, i.e., they contain references to code elements in the project defined outside the given code snippet. This is confirmed by our analysis of how GPT-4 performs on various classes of vulnerabilities. For instance, we observe that GPT-4 performs significantly better on "Out-of-bounds Read" vulnerabilities because these are often localized in a method. In contrast, GPT-4 consistently performs worse on "Improper Authentication" vulnerabilities because they rely on the broader context of the application (inter-procedural components, version updates, etc.).

We compare CodeQL, a static analysis engine, with GPT-4 on three datasets: OWASP, Juliet Java, and Juliet C/C++. We find that GPT-4 performs better on OWASP (0.05 higher F1) and Juliet C/C++ (0.29 higher F1). We further find that GPT-4 performs remarkably better than CodeQL in detecting vulnerabilities related to OS Command Injection – GPT-4 detects 416 vulnerabilities in Juliet C/C++ that are not detected by CodeQL. This is because CodeQL does not contain specifications for the OS Command Injection-related APIs used in these samples causing such difference in performance. This observation further reflects the advantage of using LLMs over manual modeling, which is laborious and hence challenging to maintain.

LineVul, a deep-learning based vulnerability detection tool, achieves a 1.0 F1 score on Juliet C/C++, even when fine-tuned on only 50% of the training data, and 0.51 on CVEFixes C/C++ when trained on 80% of the training data, which is lower than the 0.65 F1 score achieved by our prompting-based approach. However, unlike the explanations provided by LLMs, LineVul's results are not interpretable and do not include any useful hints to developers for fixing the vulnerabilities.

We test the effectiveness of LLMs against three adversarial attacks on code: dead-code injection, variable renaming, and inserting (always true) branches. We observe that such attacks can cause mild degradation in performance by, on average, 11% with dead-code injection, 4.33% with variable renaming, and 12.67% with inserting branches. These results, along with the original (non-adversarial) scores demonstrate that our experiments are not impacted by data leakage, i.e., it is highly unlikely that LLMs have seen or memorized the datasets that we study. Further, this shows that LLMs are robust, i.e., they still demonstrate good performance in presence of such attacks.

We further study whether fine-tuning smaller models, such as GPT-3.5 and CodeLlama-7B, can improve their performance, especially compared to the larger models, like GPT-4. We find that fine-tuning significantly improves their performance on synthetic datasets, even outperforming GPT-4. However, we observe very limited gains for real-world datasets. Further, we find that fine-tuned models show poor generalizability when tested on a different dataset, motivating the need for more specialized fine-tuning strategies.

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

- **Empirical Study:** We conduct the first comprehensive study on how state-of-the-art LLMs, such as GPT-4 and CodeLlama, perform in detecting security vulnerabilities across five datasets and two programming languages (C/C++ and Java). We also compare the LLMs against popular static analysis tools and deep-learning-based vulnerability detection tools.
- **Prompting Strategies:** We design four prompting strategies for LLMs, inspired by the recent advances in natural language processing and traditional programming languages techniques, that elicit the best performance from LLMs and also provide human-readable explanations for their predictions.
- **Robustness and adaptability of LLMs:** We study how the performance of LLMs is impacted by semantics-preserving adversarial attacks on code. Further, we also explore whether fine-tuning improves the performance of LLMs and whether their results are generalizable.
- **Insights and Recommendations:** We perform a rigorous analysis of LLMs' predictions and highlight vulnerability patterns that impact the performance of these models. Finally, we provide a list of recommendations for future researchers and developers on how to leverage LLMs for vulnerability detection such as adopting better prompting strategies and complementing their strengths with those of static analysis tools.

2 ILLUSTRATIVE EXAMPLE

To compare the different approaches for vulnerability detection, let us consider a (simplified) C++ program, presented in Listing 1, from the Juliet vulnerability dataset [Black and Black 2018]. The method func receives data from a socket (Line 7) and stores it in variable data (Line 10). The method calls _execvp that takes a command to execute (cmd.exe) and the arguments for the command (args and hence, data) as inputs (Line 12-13). The _execvp method finds the command via the PATH environment variable and transfers the control of the current process to the command. In this case, the call to _execvp can run commands on the Windows Command Line depending on the value of data. Because data is received from a socket and is hence user-controlled, an attacker could potentially manipulate the data sent through the socket to run malicious OS commands (e.g., rm -rf). Attacks like these that attempt to execute arbitrary commands on the host operating system via a vulnerable application are classified as CWE-78: OS Command Injection under the Common Weakness Enumeration (CWE) list. OS Command Injection is the fifth most dangerous vulnerability according to "2023 CWE Top 25 Most Dangerous Software Weaknesses"[MITRE Top 25 CWEs 2023]. We discuss how different approaches try to detect this vulnerability.

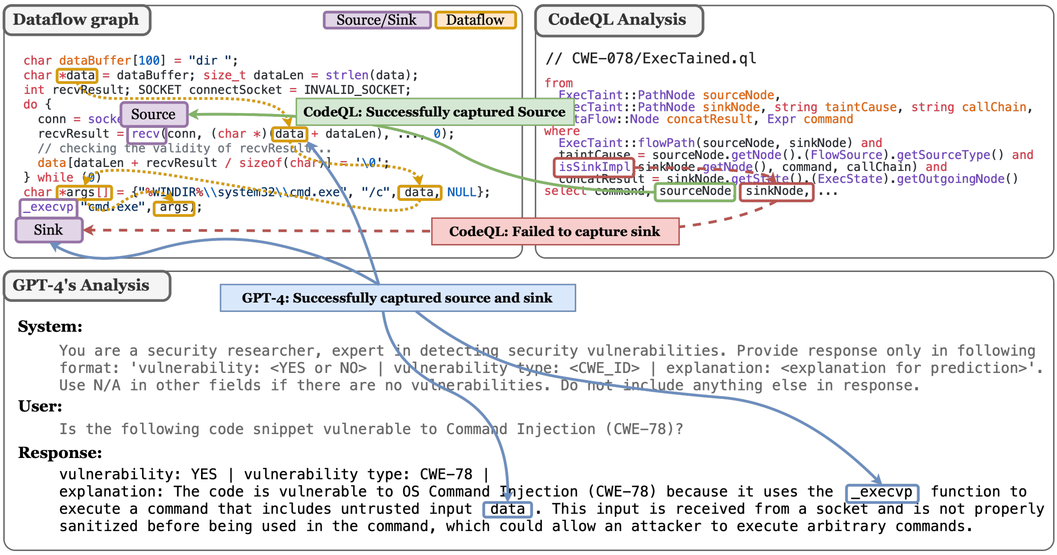
CodeQL. CodeQL [Avgustinov et al. 2016] is a popular open-source static analysis engine developed by GitHub. It extracts data flows in code into a relational database and allows users to write logic queries in a SQL-like language for detecting security vulnerabilities. Such queries are typically written and maintained by security experts and can support various known CWEs in common interpreted and compiled languages (including C++ and Java). CodeQL treats the detection of most CWEs as a taint tracking problem where it identifies "tainted paths" from sources to sinks, i.e., data flow paths between nodes that are not sanitized. The specifications of the sources, sinks, and sanitizers vary depending on the target CWE. GitHub provides a library of CodeQL queries that can be used to detect such popular CWEs out-of-the-box.

CodeQL provides two queries to detect CWE-78: cpp/command-line-injection and cpp/wordexp-injection. We focus only on the former since the latter is specific to the worexp command which is not relevant to this code snippet. Figure 1 presents the cpp/command-line-injection query. While CodeQL identifies data as a tainted source, it doesn't detect that _execvp is a sink because the specification of the sink (isSinkImpl) only considers specific exec commands as vulnerable sinks (which does not include _execvp). As a result, CodeQL does not detect this vulnerability.

While CodeQL allows the user to write logical rules corresponding to specific patterns, this approach also presents two major challenges. First, *it is difficult to specify general rule-based patterns*

```
1  void func(){
2      char dataBuffer[100] = "dir ";
3      char *data = dataBuffer; size_t dataLen = strlen(data);
4      int recvResult; SOCKET connectSocket = INVALID_SOCKET;
5
6      do {// setup socket and receive user input
7          conn = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
8          recvResult = recv(conn, (char *)(data + dataLen), sizeof(char) * (100 - dataLen - 1), 0);
9          // missing validation of recvResult
10         data[dataLen + recvResult / sizeof(char)] = '\0';
11     } while (0)
12     char *args[] = {"%WINDIR%\\system32\\cmd.exe", "/c", data, NULL};
13     _execvp("cmd.exe", args);
14 }
```

Listing 1. A C++ program selected from the Juliet [Black and Black 2018] dataset. On line 6-7, the program receives user input from a socket and stores it in data, which is then passed to _execvp (line 12). However, executing _execvp without validating data can lead to OS Command Injection vulnerability (CWE-78).


Dataflow graph
Source/Sink
Dataflow
Source
Sink
CodeQL: Successfully captured Source
CodeQL: Failed to capture sink
CodeQL Analysis
GPT-4's Analysis
GPT-4: Successfully captured source and sink

```
char dataBuffer[100] = "dir ";
char *data = dataBuffer; size_t dataLen = strlen(data);
int recvResult; SOCKET connectSocket = INVALID_SOCKET;
do {
  conn = socke
  recvResult = recv(conn, (char *)(data + dataLen), ..., 0);
  // checking the validity of recvResult...
  data[dataLen + recvResult / sizeof(char)] = '\0';
} while (0)
char *args[] = {"%WINDIR%\\system32\\cmd.exe", "/c", data, NULL};
_execvp("cmd.exe", args);
```

```
// CWE-078/ExecTained.ql

from
  ExecTaint::PathNode sourceNode,
   ecTaint::PathNode sinkNode, string taintCause, string callChain,
    taFlow::Node concatResult, Expr command
where
  ExecTaint::flowPath(sourceNode, sinkNode) and
  taintCause = sourceNode.getNode().(FlowSource).getSourceType() and
  isSinkImpl(sinkNode.getNode(), command, callChain) and
  concatResult = sinkNode.getState().(ExecState).getOutgoingNode()
select command, sourceNode, sinkNode, ...
```

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> | 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 vulnerable to Command Injection (CWE-78)?
```

Response:

```
vulnerability: YES | vulnerability type: CWE-78 |
explanation: The code is vulnerable to OS Command Injection (CWE-78) because it uses the _execvp function to
execute a command that includes untrusted input data. This input is received from a socket and is not properly
sanitized before being used in the command, which could allow an attacker to execute arbitrary commands.
```

Fig. 1. CodeQL vs GPT-4 analysis: CodeQL fails to detect the OS Command Injection vulnerability because it does not consider _execvp function as a sink. In contrast, GPT-4 successfully detects both the source (`data`) and sink (_execvp) and explains the missing sanitization for the user input in its response.

that can help detect several occurrences of the CWE. There are several commands with different argument / return types that can transfer control to the command line and these need to be carefully modelled as tainted nodes in CodeQL. If these nodes are rather modelled without many constraints, the queries are bound to generate a lot of false positives. Secondly, *CodeQL queries are written in the declarative QL language which is not very accessible.* This makes it difficult to quickly write new queries to detect custom vulnerable patterns.

Deep Learning based vulnerability tools. Several deep learning-based vulnerability detection tools have been proposed in recent years that attempt to learn vulnerable patterns from large corpuses of code. This eliminates the need for writing specific rules for detecting vulnerabilities and has been made possible by the introduction of large real-world datasets like CVEFixes for Java and C++. While several prior works proposed Graph Neural Networks to model properties of code that are relevant to vulnerability detection (data flow, control flow, program dependencies, etc.), recent work has shown that transformer-based models offer significant improvements over specialized neural network architectures. LineVul[Fu and Tantithamthavorn 2022] is a transformer-based line-level vulnerability detection approach which treats code as text-based tokens. LineVul uses a pretrained code model, CodeBERT[Feng et al. 2020], as the base model which is then fine-tuned on a very large corpus of real-world C++ vulnerable and non-vulnerable methods to predict if the target method is vulnerable. It also aggregates the attention scores for sub-word tokens in every line to rank lines as vulnerable based on their contribution to the prediction. Note that LineVul only predicts a binary label (vulnerable / not vulnerable) and doesn't classify the type of CWE. The predicted lines of vulnerable code should, however, help understand the target CWE, if any.

Let's look at how LineVul processes the example in Listing 1. The method `func` is first tokenized into subwords before being sent as an input to the CodeBERT model fine-tuned on vulnerability detection. The model outputs a binary label (0 / 1) indicating if the target method is vulnerable or not. The attention scores from the model are then used to generate line level scores for line-level detection. LineVul predicts that this snippet is vulnerable with lines 4, 12, 10, 7, 13 in Listing 1 as the top-5 lines with the highest aggregate attention scores.

While these specialized deep learning-based vulnerability tools alleviate the need for writing rules, they cannot explain their predictions in a human-readable format. This makes it difficult to understand why the model makes a given prediction and whether the prediction can be trusted or not. Hence, while LineVul provides a ranked list of vulnerable lines, it is difficult to infer the vulnerability that is cumulatively indicated by these lines. A CWE-level prediction with human-understandable explanations on the other hand can not only describe why the given snippet is vulnerable but also provide guidance on how to prevent it.

Large Language Models (LLMs). LLMs have emerged as powerful general tools that can perform various code related tasks. Moreover, recent models like OpenAI GPT-4 have been trained using human feedback and have been shown to provide human-readable explanations that are easy to understand. These models treat code as text and can take text-based instructions as "prompts" to perform any given task. These models offer an impressive capability to just use the provided instructions to understand the task (without any further training). When the code snippet in Listing 1 is provided to GPT-4 along with an instruction “Is this code snippet vulnerable to CWE-78 (OS Command Injection)?”, the model accurately analyzes the data flow paths within the method `func` and provides a human-readable explanation for why this snippet is vulnerable. The prompt and the response are presented in Listing 1.

Hence, with a very simple text-based prompt, GPT-4 is able to correctly infer that the given snippet is prone to CWE-78 without needing any specialized context about the task. Does this mean that GPT-4 and other Large Language Models can be used to detect security vulnerabilities out-of-the-box? Would a prompt as simple as the one shown above work for detecting all vulnerabilities? Can the model correctly reason about the data flow paths in the given code snippet if needed to detect a vulnerability? How prone are these models to raising false alarms? Can they self detect and mute false alarms?

In this work, we explore these questions by benchmarking popular LLMs on known vulnerability datasets. We find that LLMs like GPT-4 and CodeLlama offer competitive performance on the benchmarks corresponding to the Top 25 most dangerous CWEs when compared with CodeQL and LineVul. We also find that modifications to the prompt that explicitly ask the model to reason about the data flow paths help with producing vulnerability specifications that can ease debugging, and further prompting the model to self-evaluate its response can help in pruning false alarms. We evaluate whether fine-tuning these models can improve their performance (as predominantly done by prior Deep Learning-based vulnerability detection works). We also test the performance of these models on semantically equivalent modified versions of the samples to understand the effects of test data contamination.

3 APPROACH

3.1 Datasets

For our study, we select 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. We describe each dataset next.

OWASP. The Open Web Application Security Project (OWASP) benchmark [OWASP Benchmark Suite 2023] 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). Out of these, 1415 tests contain a security vulnerability that is exploitable (vulnerable), while in the remaining 1325 cases, the vulnerabilities cannot be exploited (not vulnerable).

Juliet. Juliet [Boland and Black 2012] 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. Each program consists of at least one vulnerable and one non-vulnerable test case. For our paper, we use the latest version, Juliet 1.3 [Black and Black 2018; Juliet C/C++ 2023; Juliet Java 2023].

CVEFixes. Bhandari et al. [2021] 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.

Table 1. Datasets

Dataset	Language	Type	Size	Vulnerable	Non-Vulnerable	CWEs
OWASP[OWASP Benchmark Suite 2023]	Java	Synthetic	2740	1415	1325	11
SARD Juliet (C/C++) [Juliet C/C++ 2023]	C/C++	Synthetic	81,280	40,640	40,640	118
SARD Juliet (Java) [Juliet Java 2023]	Java	Synthetic	35,940	17,970	17,970	112
CVEFixes [Bhandari et al. 2021]	C/C++	Real	19,576	8223	11,347	131
CVEFixes [Bhandari et al. 2021]	Java	Real	3926	1461	2465	68

3.2 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: $2 * \frac{\textit{Precision * Recall}}{\textit{Precision + Recall}}$.

3.3 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 is presumably is much larger) allows up to 8192 tokens in the input prompt. Since these models are closed-source, we also evaluate CodeLlama models [Rozière et al. 2023], which were recently open-sourced by Meta. We select two versions of these models: CodeLlama-7B and CodeLlama-13B, containing 7 billion and 13 billion parameters, respectively. We use the Hugging Face APIs [Hugging Face 2023] 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 2 presents more details for each LLM.

Table 2. Details of LLMs

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

3.4 Static Analysis Tools

CodeQL is a popular open-source static analysis engine developed by GitHub. CodeQL extracts data flows in code into a relational database and allows users to write logic queries in a SQL-like language for detecting security vulnerabilities. CodeQL queries for vulnerability detection have been written and maintained by security experts and can support various known CWEs in various interpreted and compiled languages (including C/C++ and Java).

Since CodeQL needs to compile the entire code base to be able to extract symbolic data from code, we only evaluate it on the synthetic datasets (from Juliet and OWASP). The CVEFixes datasets only provide access to individual methods in a project and hence are not directly amenable to CodeQL's analysis.

To run CodeQL, we map the top 25 CWEs to the corresponding list of QL queries mentioned in the CWE Coverage charts published by GitHub for Java [CodeQL Java CWE 2023] and C/C++ [CodeQL CPP CWE 2023]. We then create CodeQL databases for each synthetic dataset using the CodeQL CLI. We separate the vulnerable and non-vulnerable samples into separate datasets for easier evaluation. After creating the databases, we run the QL queries corresponding to the CWEs supported by the datasets. Because multiple queries can be associated with a CWE, we consider CodeQL's prediction to be "vulnerable to CWE" if *any* of the queries corresponding to that CWE detects the vulnerability.

3.5 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 [Salewski et al. 2023], 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. Additionally, we use the following dimensions to determine the rest of the features of the user prompt:

(1) Vulnerability Granularity: Predict if the target snippet is vulnerable to any CWE/a target CWE.
(2) Reasoning: Systematically reason about the target code snippet before concluding if a given snippet is vulnerable.
(3) Validation: Identify potential flaws in analysis and correct them without any external feedback.

We construct different prompting strategies to study the effects of these dimensions on performance. The *Basic* and the *CWE specific* prompts differ in vulnerability granularity, the *Dataflow analysis-based prompt* provides a scratchpad for dataflow-based *reasoning*, and the *Dataflow analysis-based prompt with Self Reflection* allows models to self-validate their predictions.

3.5.1 Basic prompt. We design a very simple prompt (shown in Listing 2) 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. We refer to it as the Basic prompt in this study. The prompt begins with the message "Is the following code snippet prone to any security vulnerability?" followed by the code snippet.

```
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 2. Basic LLM Prompt

3.5.2 CWE specific prompt. The CWE specific prompt is presented in Listing 3. 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 (and 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. So 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.

```
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 <CWE>?

<CODE_SNIPPET>

Response:
```

Listing 3. CWE-specific LLM Prompt

3.5.3 Dataflow analysis-based prompt. Besides the change in vulnerability granularity explored in the previous two prompt designs, we were interested in studying whether providing specific guidelines about code analysis can help the model reason about code better and hence 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. Inspired by this, we designed the CWE-DF prompt shown in Listing 4 that prompts the model to simulate a source-sink-sanitizer-based dataflow analysis on the target code snippet before predicting if it is vulnerable.

```
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 format:
Here is a data flow analysis of the given code snippet:
A. Sources:
<numbered list of input sources>
B. Sinks:
<numbered list of output sinks>
C. Sanitizers:
<numbered list of sanitizers, if any>
D. Unsanitized Data Flows:
<numbered list of data flows that are not sanitized in the format (source, sink, why this flow
     could be vulnerable)>
E. Vulnerability analysis verdict: vulnerability: <YES or NO> | vulnerability type: <CWE_ID> |
     vulnerability name: <NAME_OF_CWE> | explanation: <explanation for prediction>

User: Is the following code snippet prone to <CWE>?

<CODE SNIPPET>

Response:
```

Listing 4. Dataflow analysis-based LLM Prompt

3.5.4 Dataflow analysis-based prompt with Self Reflection. The final dimension we explore is whether the model is capable of correcting any flawed reasoning chains in its analysis. We leverage the chat modality of LLMs to interact with the model after it presents its initial response using the CWE-DF prompt. We append the response from the model to the CWE-DF prompt and add an additional user prompt “Is this analysis correct?”. We also include the response format instructions again to ensure that the next response from the model is in a format that can be parsed. We term this additional check “Self Reflection” following prior works that investigate whether LLMs like GPT-4 can self-correct. The prompt is presented in Listing 5 and is referred to as the CWE-DF+SR prompt in the study.

3.5.5 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 / explanations of some kind before detecting if the given snippet is vulnerable. With Chain-of-thought prompting, we explicitly ask the model to

```
System: <Same as the Dataflow analysis-based prompt>
User: <Same as the Dataflow analysis-based prompt>

Response: Here is a data flow analysis of the given code snippet... (rest of the response)

User: Is this analysis correct? Return your response in the following format:
Yes / No
explanation: <reason for the analysis being correct or wrong>
Final Vulnerability analysis verdict: vulnerability: <YES or NO> | vulnerability type: <CWE_ID>

Response:
```

Listing 5. Dataflow analysis inspired LLM Prompt with Self Reflection

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 also tried a variant of the Dataflow analysis-based prompt which divides the data flow analysis into multiple prompts where each step of the analysis (source, sink, sanitizer and unsanitized paths identification) is a separate call to the model. We did not observe any improvements over the Dataflow analysis-based prompt using this setup. Moreover, this setup is much more computationally expensive since it calls the LLM four times (as opposed to just once with the Dataflow analysis-based prompt).

3.6 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). 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 (e.g., "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 3).

CVEFixes. For each CVE, CVEFixes collects 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 evaluation.

Data Selection. Due to the huge cost of running pre-trained LLMs, we select a subset of samples from the original datasets. We select samples corresponding to vulnerabilities types (or CWEs) listed in MITRE's Top 25 Most Dangerous Software Weaknesses [MITRE Top 25 CWEs 2023]. Because the Juliet C/C++, Juliet Java, and CVEFixes C/C++ datasets are quite large, we further randomly select a subset of 2000 samples for each. Table 3 presents the dataset samples selected at each stage.

3.7 Fine-tuning LLMs

Data preparation for fine-tuning. For Juliet C/C++ and CVEFixes C/C++, we use 2000 samples that we used in the main evaluation (Section 4.3) for the test set and randomly choose K training samples from the rest, where K is either 1000 or 2000. Because OWASP and CVEFixes Java are much smaller, we split the data into 50% train and test sets for fine-tuning.

Finetuning GPT 3.5. OpenAI does not provide any public APIs for fine-tuning GPT-4 models. Hence, we use OpenAI's APIs for fine-tuning gpt-3.5 (`gpt-3.5-turbo`) model, which allows us to upload `jsonl` files containing training and test sets and returns the test accuracy along with training statistics (such as training loss).

Finetuning CodeLlama. We choose the CodeLlama-7b model for fine-tuning for two key reasons. First, our main goal is to explore if smaller LLMs can come close to the performance of much larger LLMs like GPT-4 on vulnerability detection by adopting simple fine-tuning strategies. Second, the relatively smaller model size allows us to explore different fine-tuning settings across languages and datasets.

4 EVALUATION

4.1 Research Questions

We address the following research questions in this work:

- **RQ1:** How do different pre-trained LLMs perform in detecting security vulnerabilities across different languages and datasets?
- **RQ2:** How do pre-trained LLMs perform on different classes of security vulnerabilities?
- **RQ3:** How do pre-trained LLMs compare to static analysis tools?
- **RQ4:** How do LLMs compare to prior deep learning-based vulnerability detection tools?
- **RQ5:** Can we improve the performance of LLMs further by fine-tuning? Do the fine-tuned models generalize across datasets?
- **RQ6:** Can adversarial attacks impact the performance of LLMs in detecting vulnerabilities?

Table 3. 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
Data-Processing	2740	81,280	35,940	19,576	3926	144,002
Top 25 CWE	1478	11,766	8,506	12,062	1810	23,560
Data Selection	1478	2000	2000	2000	1810	9288

Table 4. Effectiveness of LLMs in Predicting Security Vulnerabilities (Java).

Model	Prompt	OWASP				Juliet Java				CVEFixes Java			
		A	P	R	F1	A	P	R	F1	A	P	R	F1
GPT-4	Basic	0.54	0.54	1.00	0.70	0.54	0.53	0.86	0.66	0.56	0.40	0.36	0.38
GPT-4	CWE	0.56	0.55	1.00	0.71	0.66	0.60	0.96	0.74	0.57	0.43	0.44	0.44
GPT-4	CWE-DF	0.57	0.55	1.00	0.71	0.68	0.62	0.97	0.75	0.52	0.41	0.58	0.48
GPT-4	CWE-DF+SR	0.73	0.67	0.96	**0.79**	0.85	0.83	0.89	**0.86**	0.62	0.49	0.16	0.24
CodeLlama-13B	Basic	0.54	0.53	0.97	0.69	0.56	0.63	0.73	0.67	0.59	0.38	0.13	0.19
CodeLlama-13B	CWE	0.54	0.53	0.98	0.69	0.61	0.63	0.90	0.74	0.53	0.35	0.28	0.31
CodeLlama-13B	CWE-DF	0.53	0.53	1.00	0.69	0.62	0.62	1.00	0.77	0.38	0.38	1.00	**0.55**
CodeLlama-7B	Basic	0.57	0.57	0.80	0.66	0.74	0.83	0.72	0.77	0.48	0.35	0.44	0.39
CodeLlama-7B	CWE	0.53	0.53	1.00	0.69	0.63	0.63	0.99	0.77	0.43	0.39	0.85	0.53
CodeLlama-7B	CWE-DF	0.53	0.53	1.00	0.69	0.62	0.62	1.00	0.77	0.38	0.38	1.00	**0.55**

4.2 Experimental setup

Experiments with GPT-4. We use the OpenAI public API to perform the experiments with GPT-4. We use the `ChatCompletions` API endpoint to query GPT-4 with the prompts discussed in Section 3.5. 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 model.

Experiments with CodeLlama. We run all CodeLlama experiments on two sets of machines: one with 2.50GHz Intel Xeon machine, with 40 CPUs, four GeForce RTX 2080 Ti GPUs, and 750GB RAM, and another 3.00GHz 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.

4.3 RQ1: Performance of LLMs in detecting security vulnerabilities

We evaluate the performance of pre-trained LLMs on five open-source datasets discussed in Section 3.1. Table 4 and Table 5 present the performance of GPT-4 and CodeLlama models on Java and C/C++ datasets, respectively. The **Prompt** column presents the prompting strategy used for each model. The **Basic**, **CWE**, **CWE-DF**, and **CWE-DF+SR** prompts represent the *Basic*, *CWE Specific*, *CWE Specific Prompt augmented with DataFlow Instructions*, and *CWE Specific Prompt augmented with DataFlow Instructions + Self Reflection* prompt formats presented in Section 3.5, respectively. The columns **A**, **P**, **R**, and **F1** denote the accuracy, precision, recall, and F1 score, respectively. Each row presents the results for one combination of a model and a prompting strategy across datasets.

GPT-4's Performance. We observe that GPT-4 (prompted with CWE-DF+SR) outperforms all other models and prompting setups on the synthetic Java datasets (OWASP and Juliet Java). More specifically, GPT-4 (CWE-DF+SR) shows 0.1 and 0.09 higher F1 over the CodeLlama models on the OWASP and Juliet Java datasets, respectively. The various prompting strategies also follow a consistent upward trend with GPT-4. When prompted with the Dataflow analysis-based prompt, GPT-4 reports a 0.09 higher F1 on the Juliet Java dataset than the Basic prompt. This is further improved by 0.11 F1 with the Self-Reflection check (CWE-DF+SR). The results with the CWE and CWE-DF prompts are comparable.

Interestingly, GPT-4 performs poorly on the CVEFixes Java dataset under all prompting strategies. While CWE-DF+SR shows the highest accuracy of 62%, its recall degrades to 0.16 and F1 score to 0.24. We observe that with the Self-Reflection step, GPT-4 has a higher tendency to predict that a given snippet is not vulnerable stating lack of context as a reason, which in turn reduces the overall performance on CVEFixes Java. Because the samples from CVEFixes Java only provide a target method that could refer to variables or methods not provided in the snippet, GPT-4 often provides a response that asks for more context. This effect is, however, not observed with the synthetic datasets because most samples in these datasets are self-contained. We study this phenomenon in more detail in Section 5.

Table 5. Effectiveness of LLMs in Predicting Security Vulnerabilities (C/C++).

Model	**Prompt**	**Juliet C/C++**				**CVEFixes C/C++**			
		A	**P**	**R**	**F1**	**A**	**P**	**R**	**F1**
GPT-4	Basic	0.53	0.51	0.93	0.66	0.51	0.43	0.38	0.40
GPT-4	CWE	0.56	0.52	0.96	0.68	0.51	0.44	0.52	0.48
GPT-4	CWE-DF	0.57	0.53	0.98	0.69	0.53	0.51	0.80	0.62
GPT-4	CWE-DF+SR	0.90	0.87	0.92	**0.89**	0.51	0.51	0.16	0.25
CodeLlama-13B	Basic	0.46	0.47	0.79	0.59	0.51	0.50	0.14	0.22
CodeLlama-13B	CWE	0.51	0.50	0.98	0.66	0.54	0.52	0.68	0.59
CodeLlama-13B	CWE-DF	0.49	0.49	1.00	0.66	0.44	0.43	0.92	0.59
CodeLlama-7B	Basic	0.60	0.56	0.86	0.68	0.50	0.38	0.02	0.04
CodeLlama-7B	CWE	0.49	0.49	0.99	0.66	0.52	0.50	0.75	0.60
CodeLlama-7B	CWE-DF	0.49	0.49	1.00	0.66	0.49	0.49	0.98	**0.65**

The results with C/C++ datasets (Table 5) follow a similar trend. GPT-4 reports the highest F1 score of 0.89 on the Juliet C/C++ with the CWE-DF+SR prompt. The Self Reflection check significantly improves GPT-4's precision by 0.34 and F1 score by 0.20 on the Juliet C/C++ over the CWE-DF prompt. However, similar to CVEFixes Java, GPT-4 performs poorly on the CVEFixes C/C++ across all prompts with the CWE-DF+SR prompt reporting an accuracy of 51% and CWE-DF reporting the highest F1 score of 0.62. The CWE-DF+SR prompt reports the lowest F1 score, 0.25.

With the Basic prompt, GPT-4 detects that the snippet is vulnerable but predicts an incorrect vulnerability (or CWE) in 52.39% and 57.67% of all vulnerable samples in the Java and C++ datasets respectively. The CWE specific prompt results in the model correctly mentioning the target CWE in its predictions whenever it predicts that the sample is vulnerable on all datasets except CVEFixes C/C++, where it mentions an unrelated CWE in 10 samples. We do a qualitative comparison of the prompting strategies in Section 5.2.

CodeLlama's Performance. We observe that CodeLlama models, despite being much smaller than GPT-4, perform relatively well on most datasets, even with the Basic prompt. Interestingly, the CWE specific prompt and Dataflow analysis-based prompt improve the recall for OWASP, Juliet C/C++, and Juliet Java benchmarks but do not significantly improve their F1 scores (only up to 0.07 for Juliet C/C++). For CVEFixes C/C++ and CVEFixes Java, however, we observe a much more significant improvement in F1 scores (up to 0.61 increase). Surprisingly, both CodeLlama models improve F1 scores over GPT-4 on CVEFixes Java by 0.11 points with the CWE-DF prompt, whereas CodeLlama-7B achieves the highest F1 score of 0.65 (0.05 higher than GPT-4) on CVEFixes C/C++. However, the accuracy of CodeLlama models remains lower than GPT-4.

We do not evaluate the Dataflow analysis-based prompt with Self Reflection on CodeLlama models because prior works have reported that self-reflection is an emergent skill, i.e., it predominantly shows improvement only with the larger and stronger models like GPT-4 [Shinn et al. 2023].

Both CodeLlama models perform well at predicting the presence of a vulnerability, even with the Basic prompt, on the synthetic datasets. However, when we inspect the explanations, we find that they often mis-predict the type of vulnerability (or CWE) in both synthetic and real-world benchmarks. Specifically, CodeLlama-7B predicts that a code is vulnerable but predicts the incorrect CWE in 40% and 50% of the cases for all Java and C++ benchmarks, respectively. For CodeLlama-13B, the mis-prediction rates are slightly lower at 35% and 45%, respectively.

The CWE and CWE-DF prompts, expectedly, increase the recall because these prompts force the LLMs to focus on only one specific class of vulnerability. *However, the precision of the models does not increase significantly, rather they stay between 0.50 and 0.60 in most settings.* Interestingly, unlike GPT-4, CodeLlama models tend to almost always predict that the given code snippet is vulnerable with these two prompts. As a result, their precision values are low. In contrast, GPT-4 obtains higher precision with the self-reflection step for OWASP and Juliet benchmarks.

Comparison of LLMs. While GPT-4 provides significant improvements over CodeLlama models with self-reflection, we do not observe significant differences between the two CodeLlama models. We also find some interesting but counter-intuitive patterns. For instance, we observe that CodeLlama-7B performs better than both CodeLlama-13B and GPT-4 for Juliet Java with the Basic prompt. Listing 6 presents a representative example from the integer overflow vulnerability (CWE-190), where CodeLlama-7B performs better.

In Listing 6, an integer overflow cannot occur in the given context. Line 11 adds 1 to the integer `data`, which can potentially cause an integer overflow. However, in this context, the `data` variable can only contain a value 2. CodeLlama-7B recognizes this and correctly points out: *The code snippet does not contain any obvious security vulnerabilities.* However, CodeLlama-13B ignores the hard-coded values and predicts that the code is vulnerable, explaining that: *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.* Interestingly, we find that GPT-4 with the Dataflow analysis-based prompt also exhibits similar behavior on this example, indicating that smaller models may sometimes perform better than larger models.

```
1  private void func() throws Throwable {
2      int data;
3      switch (5) {
4      case 6:
5          data = 0; break; // Hardcoded values in data
6      default:
7          data = 2; break; // Hardcoded values in data
8      }
9      switch (7) {
10     case 7: // POTENTIAL FLAW: Integer Overflow
11         int result = (int)(data + 1); // But no overflow due to hardcoded values in data
12         IO.writeLine("result: " + result);  break;
13     default:
14         IO.writeLine("fixed string"); break;
```

Listing 6. False Positive Prediction on Integer Overflow (Juliet Java) by GPT-4 and CodeLlama-13B.

Since GPT-4 performs the best across multiple datasets, we primarily focus on it in the subsequent sections. We, however, mention relevant insights about any distinct patterns we see with the CodeLlama models.

4.4 RQ2: How do LLMs perform on different classes of security vulnerabilities?

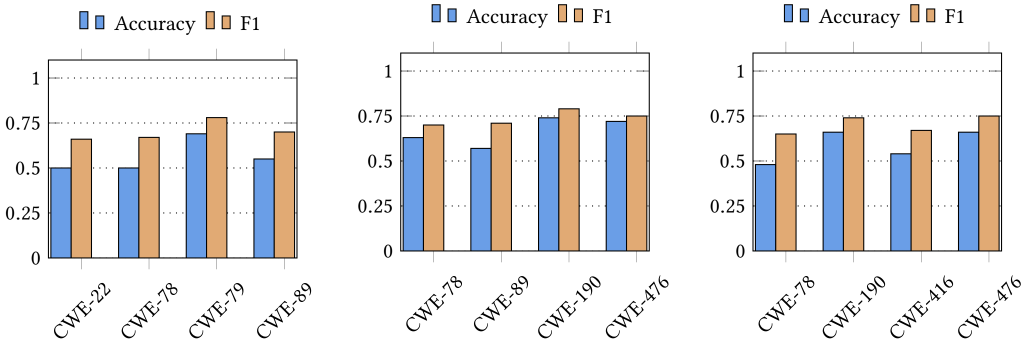
We evaluate how GPT-4 performs on various classes of security vulnerability (or CWEs).

GPT-4 on Synthetic Datasets. Figure 2 presents the CWE-wise distribution of GPT-4 (CWE-DF) results on the OWASP Java, Juliet Java, and Juliet C++ datasets. GPT-4 reports an F1 score 0.6

or more on all CWEs across all synthetic datasets. We observe the best results for CWE-79 (SQL Injection) on OWASP Java (F1 score of 0.78), and CWE-190 (Integer Overflow) and CWE-476 (Null pointer dereference) on Juliet Java (F1 scores 0.79 and 0.75 respectively) and Juliet C++ datasets (F1 scores of 0.74 and 0.75 respectively).

GPT-4 on Real-world Datasets. Figure 3 presents the CWE-wise distribution of GPT-4 (CWE-DF) results on the CVEFixes Java dataset. The model performs the best on CWE-787 (Out-of-bounds Write) with an accuracy of 79% and an F1 score of 0.81 followed by CWE-125 (Out-of-bounds Read), with an accuracy of 75% and an F1 score of 0.78. CWE-476, CWE-190, CWE-306 and CWE-862 also report F1 scores over 0.6. We observe the worst performances for CWE-89 (SQL Injection), with an accuracy of 47% and F1 score of 0.26, and CWE-287 (Improper Authentication), with an accuracy of 46% and F1 score of 0.3. GPT-4 also reports an F1 score of 0.0 on CWE-798 (hard-coded credentials).

Figure 4 shows the CWE-wise distribution of GPT-4 (CWE-DF) results on the CVEFixes C/C++ dataset. The model reports the highest F1 score of 0.75 and accuracy of 60% on CWE-20 (Improper Input Validation) followed by CWE-862 (Missing authorization) and CWE-125 (Out-of-bounds Read) with an F1 score of 0.67 and 0.66 respectively. The lowest F1 score of 0 is reported on CWE-22 (Path Traversal) and CWE-287 (Improper Authentication, with the lowest accuracy of 38%).



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

Fig. 2. GPT-4 on Synthetic Datasets (with the Dataflow analysis-based prompt).

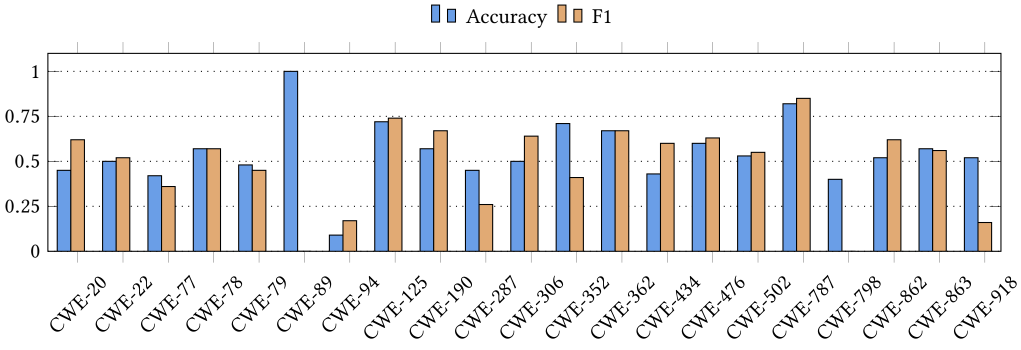


Fig. 3. GPT-4 on CVEFixes Java (with the Dataflow analysis-based prompt).

From these results, we can infer that GPT-4 consistently performs better on Out-of-bounds Read / Write (CWE-125, CWE-787), Missing authorization (CWE-862), Null pointer dereference

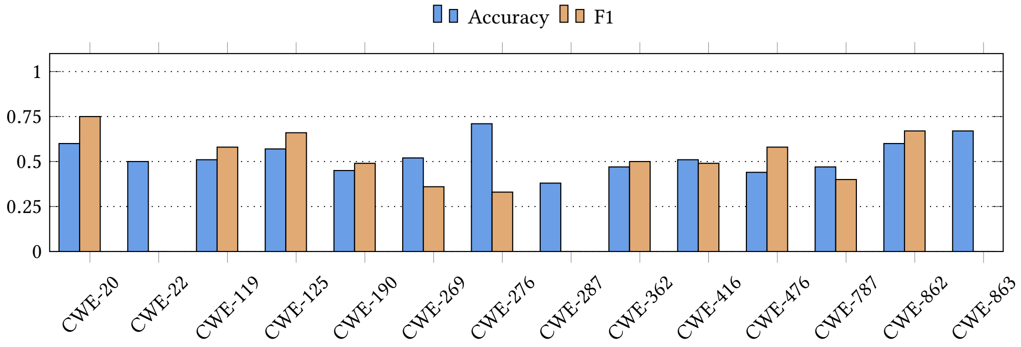


Fig. 4. GPT-4 on CVEFixes C/C++ (with the Dataflow analysis-based prompt).

(CWE-476) and Integer Overflow (CWE-190). The higher performance on the Out-of-bounds Read and Write, Null pointer dereference, and Integer Overflow vulnerabilities can likely be attributed to the fact that these are fairly self-contained and little additional context is needed to detect them. Other vulnerabilities likely need more context to be detected correctly. We manually examined samples corresponding to CWE-287 in the CVEFixes Java dataset and found that all false negatives correspond to CVE-2021-39177 where the patch upgrades to a newer version of a dependency that introduces some checks to prevent against Improper Authentication. The changes are therefore parameter order swaps in methods that use this dependency and it is not possible to detect that they are vulnerable. The false positives, on the other hand, are samples where the model is suspicious of every external input and hence errs on the side of always finding the snippet vulnerable. Appendix A.2 presents some examples that demonstrate these findings. We present the CWE distribution of various datasets in Appendix B.

4.5 RQ3: How do LLMs compare against static analysis tools?

We next explore how GPT-4 (CWE-DF+SR) compares against CodeQL. In this study, we run the CodeQL queries designed for the top 25 CWEs on two synthetic Java datasets, namely OWASP and Juliet Java, and one synthetic C/C++ dataset (Juliet C/C++). Table 6 presents results from CodeQL and GPT-4 on the three datasets. GPT-4 (CWE-DF+SR) reports a 0.05 higher F1 that CodeQL on OWASP and a 0.29 higher F1 on Juliet C/C++ (with a 49% higher recall). CodeQL reports a 0.06 higher F1 on Juliet Java.

Table 7 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++).

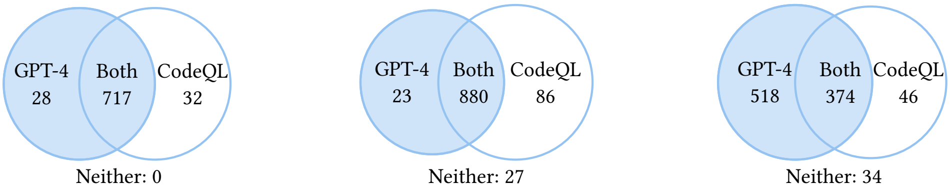
Table 6. Comparison with CodeQL

Model	OWASP				Juliet Java				Juliet C/C++			
	A	P	R	F1	A	P	R	F1	A	P	R	F1
GPT-4 (CWE-DF+SR)	0.73	0.67	0.96	0.79	0.85	0.83	0.89	0.86	0.89	0.87	0.92	0.89
CodeQL	0.65	0.6	0.96	0.74	0.92	0.9	0.95	0.92	0.72	0.97	0.43	0.60

To understand how CodeQL qualitatively compares against GPT-4, we create a confusion matrix of vulnerable samples from each dataset. Figure 5 presents the CWE-wise distribution of vulnerable

Table 7. CodeQL CWE-wise metrics

Model	OWASP				Juliet Java				Juliet C/C++			
	A	P	R	F1	A	P	R	F1	A	P	R	F1
CWE-79	0.80	0.73	1.00	0.84	-	-	-	-	-	-	-	-
CWE-89	0.59	0.57	1.00	0.73	0.86	0.78	1.00	0.88	-	-	-	-
CWE-416	-	-	-	-	-	-	-	-	0.63	1.00	0.21	0.35
CWE-78	0.60	0.58	0.78	0.67	0.92	0.85	1.00	0.92	0.54	1.00	0.03	0.06
CWE-22	0.54	0.52	1.00	0.68	-	-	-	-	-	-	-	-
CWE-476	-	-	-	-	0.78	0.79	0.79	0.79	0.76	0.87	0.67	0.76
CWE-190	-	-	-	-	0.96	1.00	0.93	0.96	0.94	0.99	0.88	0.93



CodeQL / GPT-4	CWE-79		CWE-89	
	D	ND	D	ND
D	242	0	266	0
ND	4	0	6	0
	CWE-78		CWE-22	
D	97	28	112	0
ND	1	0	21	0

(a) OWASP

CodeQL / GPT-4	CWE-190		CWE-89	
	D	ND	D	ND
D	493	19	305	0
ND	57	24	26	0
	CWE-476		CWE-78	
D	24	4	58	0
ND	3	3	0	0

(b) Juliet Java

CodeQL / GPT-4	CWE-78		CWE-190	
	D	ND	D	ND
D	15	416	313	37
ND	0	18	41	13
	CWE-416		CWE-476	
D	18	50	28	15
ND	0	2	5	1

(c) Juliet C/C++

Fig. 5. CodeQL vs. GPT-4 on vulnerable samples. For each dataset, we illustrate using venn diagram on the overall distribution of vulnerabilities detected by GPT-4 and CodeQL. We further use tables to detail 4 most significant CWEs for each dataset, where D means "detected" and ND means "not-detected".

samples binned into four categories: detected by CodeQL and GPT-4, detected by neither, detected only by CodeQL, and detected only by GPT-4. The two techniques together detect all vulnerable samples in OWASP, 97% of all vulnerable samples in Juliet Java, and 96.5% of all vulnerable samples in Juliet C/C++. CodeQL is better at detecting CWE-190, CWE-89 and CWE-22, while GPT-4 is better at detecting CWE-78, CWE-476 and CWE-416 across datasets. We present some examples of cases where CodeQL does better than GPT-4 in Appendix A.1.

4.6 RQ4: How do LLMs compare to deep learning-based vulnerability detection tools?

We compare GPT-4 and CodeLlama results against LineVul [Fu and Tantithamthavorn 2022], which is a Transformer-based vulnerability prediction tool. To allow a fair comparison with pre-trained LLMs (GPT-4 or CodeLLama), we trained LineVul on each target C/C++ dataset.

We use the same test dataset as used in Section 4.3 and use the remaining data for selecting the training dataset for LineVul. We train LineVul on two settings: 50% and 80% of the training dataset for Juliet C/C++ and CVEFixes C/C++. Table 8 presents the results.

We observe that LineVul only obtains an F1 score of 0.51 when trained on 80% of the training data for CVEFixes C/C++, whereas GPT-4 and CodeLlama-7B have F1 scores more than 0.60. With 50% training data, the performance is even worse. For Juliet C/C++, however, LineVul obtains 100% accuracy and F1 score, even with 50% of the training data. These results reflect that while

Table 8. Comparison of pre-trained LLMs to LineVul

Model	**Dataset**	**Training Size/Prompt**	**A**	**P**	**R**	**F1**
LineVul	CVEFixes C/C++	80%	0.40	0.44	0.61	0.51
LineVul	CVEFixes C/C++	50%	0.53	0.44	0.24	0.31
GPT-4	CVEFixes C/C++	CWE-DF	0.51	0.49	0.79	0.60
CodeLlama-7B	CVEFixes C/C++	CWE-DF	0.49	0.49	0.98	0.65
LineVul	Juliet C/C++	50%	1.0	1	0.99	1.0
LineVul	Juliet C/C++	80%	1.0	1.0	0.99	1.0
GPT-4	Juliet C/C++	CWE-DF+SR	0.90	0.87	0.92	0.89
CodeLlama-7B	Juliet C/C++	CWE-DF	0.49	0.49	0.98	0.65

existing deep-learning tools easily learn simpler code patterns in synthetic datasets, detecting vulnerabilities in real-world code, like CVEFixes C/C++, remains challenging. In contrast, LLMs show more impressive performance that could potentially be further improved through specialized fine-tuning and prompting techniques.

4.7 RQ5: Fine-tuning LLMs and Generalizability

We explore whether fine-tuning smaller pre-trained LLMs can further improve their vulnerability detection ability. We choose smaller LLMs, CodeLlama-7B and GPT-3.5, for this experiment. Due to the massive cost of fine-tuning LLMs, we only fine-tune with small datasets of sizes 1K and 2K, which is also a common practice in this domain. Figure 6 presents the results. For fine-tuning, we keep the same test set as used in Section 4.3 for Juliet C/C++, Juliet Java, and CVEFixes C/C++, and select training data from the remaining samples.

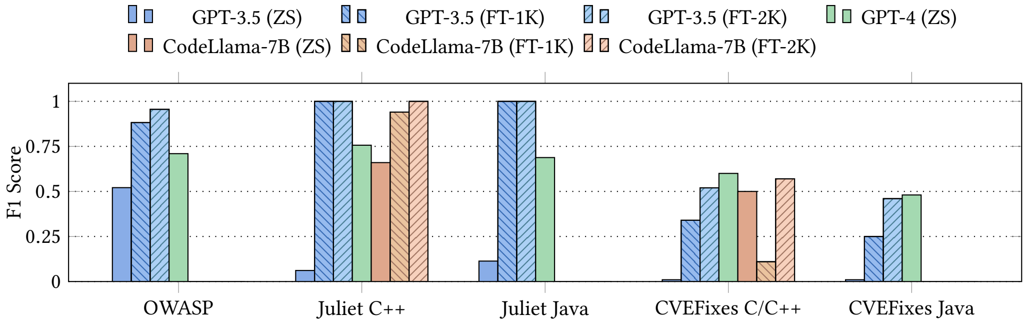


Fig. 6. We report two models' performance (F1 Score) after fine-tuning: CodeLlama-7B, and GPT-3.5. For each dataset, we fine-tune each model on 1K and 2K samples respectively (indicated by FT-1K and FT-2K). Note that the CodeLlama-7B model is only fine-tuned on Juliet C++ and CVEFixes C/C++. For reference, we also show the performance without fine-tuning (indicated by ZS) of the corresponding models including GPT-4.

The results show that fine-tuning significantly improves the performance of CodeLlama-7B on Juliet C/C++, exceeding all the pre-trained model results, even with only 1K training samples. We reason that because synthetic benchmarks like Juliet C/C++ are generated based on a limited set of templates, there are only a few distinct code patterns that the LLM needs to learn, and hence, it shows close to 100% accuracy after fine-tuning.

For CVEFixes C/C++ however, fine-tuning degrades the performance of LLMs. We observe that the fine-tuned model tends to predict "not vulnerable" in most cases after fine-tuning. With 2000 training samples, however, CodeLlama-7B, learns to generalize better, which improves the F1

Table 9. We test the generalizability of fine-tuned models by evaluating them on out-of-distribution samples. In the table, **Train** is the dataset that the model is fine-tuned on and **Test** is the dataset that the model is evaluated on. In addition to (**P**)recision, (**R**)ecall, and **F1**, we also show the (**A**)ccuracy and Δ**A**, the difference between out-of-distribution accuracy and in-distribution accuracy (lower means less generalizable).

Model	Training Size	Train	Test	A	ΔA	P	R	F1
CodeLlama-7B	1K	Juliet C/C++	CVEFixes C/C++	49%	−46%	0.48	0.45	0.46
CodeLlama-7B	2K	Juliet C/C++	CVEFixes C/C++	49%	−51%	0.49	0.96	0.65
CodeLlama-7B	1K	CVEFixes C/C++	Juliet C/C++	52%	±0%	1	0.002	0.004
CodeLlama-7B	2K	CVEFixes C/C++	Juliet C/C++	44%	−10%	0.21	0.05	0.09
GPT-3.5	1K	OWASP	CVEFixes Java	48%	−47%	0.36	0.59	0.45
GPT-3.5	1K	Juliet Java	CVEFixes Java	45%	−55%	0.43	0.82	0.56
GPT-3.5	1K	CVEFixes C/C++	Juliet C++	62%	+5%	0.42	0.70	0.52

score (to 0.57). We observe increased precision and accuracy in both cases over the pre-trained models. However, we do not observe any significant improvement over the results we obtained with pre-trained models. We conclude that because real-world datasets like CVEFixes C/C++ contain diverse vulnerability patterns (spread across 15 unique CWEs) from several real-world projects, LLMs may require more samples or a customized training strategy to learn such patterns better.

Generalizability. We test the capability of fine-tuned models (CodeLlama-7B and GPT-3.5) beyond the datasets they were trained on. The results (Table 9) show that when tested on out-of-distribution samples, we see a sharp drop in performance. The degradation is significant when the model is fine-tuned on a *synthetic* dataset and tested on a *real-world* dataset. In contrast, when the model is fine-tuned on CVEFixes, a real-world dataset, the performance stays roughly on par. We conclude that while fine-tuning may improve in-distribution performance, it may fail to generalize to unseen samples.

4.8 RQ6: Performance against Adversarial Attacks

We explore whether adversarial attacks on code can degrade the vulnerability detection performance of LLMs. Specifically, we test LLMs using semantically equivalent but syntactically confusing programs, which are randomly generated using attacks proposed in [Gao et al. 2023]. We apply three techniques, namely dead-code injection, variable renaming, and branch insertion, which are illustrated in Table 10. For evaluation, we only select samples that GPT-4 gives the correct prediction for (with the CWE prompt), and measure the percentage of samples that GPT-4 mis-predicts under attack. In Figure 7, we present the degradation in accuracy for per attack per dataset. For each attack, we select 100 random samples per dataset, apply the attack and evaluate GPT-4 using the CWE prompt.

We observe that like typical deep-learning based models, in general, *GPT-4 is also prone to adversarial attacks*. In particular, GPT-4 suffers most under the dead code injection and branch insertion attacks for Juliet Java and CVEFixes Java. In most such cases, GPT-4 mistakenly flips from non-vulnerable (correct) to vulnerable (incorrect) prediction, as indicated by the high recall (1/1/0.93 for OWASP/Juliet Java/CVEFixes Java datasets). For the dead code injection attack, we find that GPT-4 occasionally reports the vulnerability in the inserted dead code itself, without realizing that the dead code is unreachable.

Interesting observation. During the evaluation, GPT-4 detects a vulnerability in a dead code injection attack pattern proposed in [Gao et al. 2023]. In particular, the attack pattern assumes that a signed integer, when squared, is always non-negative. However, the attack does not preserve semantics, as an integer overflow could be triggered while squaring if the integer is large. GPT-4

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 10. Illustrations of three different attacks on Java programs. The attacked code (marked in grey) does not alter original program semantics, but may impact the program analyzers.

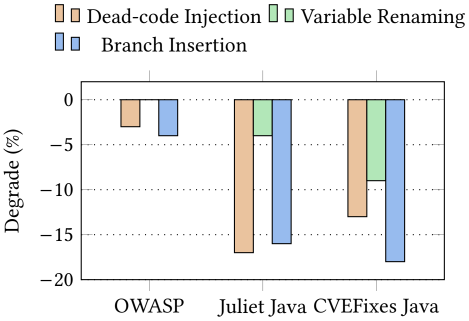


Fig. 7. The degrade of GPT-4's vulnerability detection accuracy under each dataset and each type of attack (lower means more prone to attack).

correctly detects the vulnerability when this attack pattern is injected. Due to this observation, we removed this attack pattern during our final evaluation.

5 DISCUSSION

5.1 LLM Performance on Synthetic vs Real-World Datasets

All LLMs and prompting setups consistently perform better on the synthetic datasets than the real-world ones. The samples in synthetic datasets (OWASP, Juliet Java, and Juliet C/C++) are self-contained. Hence, they contain all the information required to predict if the target method is vulnerable. In contrast, the real-world datasets (CVEFixes Java and CVEFixes C/C++) are extracted from vulnerability-fixing commits from open-source projects. Hence, the target method often refers to variables and methods that are defined in other parts of the project. In such cases, LLMs (often incorrectly) *guess* the specification of the missing components based on names and other features. For example, consider the *addCommonParams* method in Listing 7 from CVEFixes Java. This method directly passes the values of several keys from `theServletRequest` object to the `theModel` object (Lines 5-6).

```
1 protected IBaseResource addCommonParams(HttpServletRequest theServletRequest, final HomeRequest
       theRequest, final ModelMap theModel) {
2     final String serverId = theRequest.getServerIdWithDefault(myConfig);
3     final String serverBase = theRequest.getServerBase(theServletRequest, myConfig);
4     ...
5     theModel.put("serverId", serverId);
6     theModel.put("base", serverBase);
7     ...
8     return loadAndAddConf(theServletRequest, theRequest, theModel);
9 }
```

Listing 7. XSS vulnerability from CVEFixes Java.

Listing 8 presents GPT-4's response for this code snippet. GPT-4 initially predicts that this snippet is vulnerable to Cross-Site Scripting (XSS) because `addCommonParams` does not sanitize the flow of user-controlled values to this entity. There is, however, no information about what the `theModel` object is and whether it can be a potential sink for XSS. With Self-Reflection, GPT-4 identifies this ambiguity and reverses its prediction, stating that it cannot definitively say that the code is vulnerable to XSS.

```
...
The methods called on the request object could be retrieving configuration data or other non-user-
     controlled data. Furthermore, the ModelMap object (theModel) is not necessarily outputting
     data directly to a web page. It could be used in a way that properly escapes or sanitizes the
      data before it is output. Without knowing how the data in theModel is used, we cannot
     definitively say there is a cross-site scripting vulnerability.

Final Vulnerability analysis verdict: vulnerability: NO | vulnerability type: CWE-79
```

Listing 8. GPT-4's response with Dataflow analysis-based prompt and Self-Reflection on Listing 7.

Because the context length of LLMs is limited, it is practically infeasible to provide project-level code context to the model in the prompt. Moreover, it is unclear whether LLMs can reason about larger code snippets. We believe an intriguing future direction would be to find effective ways to include relevant code context in the LLM prompt.

5.2 Comparison of Prompts

Dataflow analysis-based prompt (CWE-DF) vs CWE specific prompt (CWE) vs Basic prompt. With the Basic prompt, the LLMs have a tendency to focus on irrelevant aspects of the given code snippet. For example, GPT-4, with the Basic prompt, tends to predict that a snippet is vulnerable to CWE-328 (Use of Weak Hash) if the code snippet contains terms like `encrypt` and `hash` even if the target method cannot be exploited for CWE-328. Specifying the target CWE in the prompt ensures that the LLM's response is more focused.

```
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;
}

// 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 9. GPT-4 CWE-DF detects unsanitized paths between sources and sinks.

While the performance of CWE specific prompt and the Dataflow analysis-based prompt prompting strategies is often similar, the response with the Dataflow analysis-based prompt includes vulnerability-specific *explanations* such as the relevant sources, sinks, and (missing) sanitizers, which can be useful in debugging such vulnerabilities. Listing 9 shows a response from GPT-4 using the CWE-DF prompt that correctly identifies the unsanitized flows between sources and sinks. We present more example responses using the Dataflow analysis-based prompt in Appendix A.3, including partially correct data flows that can help developers easily prune out false positives.

Understanding Self Reflection behavior with GPT-4. The Self Reflection step improves GPT-4's performance on most datasets and *self-corrects* some wrong predictions made using CWE-DF prompt. We analyze the cases where GPT-4 flips the predictions made by the Dataflow analysis-based prompt

prompt. Table 11 presents the results of this analysis. Column **Correct Self-Corrections** presents the cases where the CWE-DF+SR prompt changes an incorrect result (obtained using the CWE-DF prompt) to a correct result, and **Incorrect Self-Corrections** presents the opposite scenario. The Vul/Non-Vul ratio further splits this into how many of these samples were originally vulnerable or non-vulnerable.

GPT-4 with the CWE-DF+SR prompt correctly flips 2495 non-vulnerable samples, which were reported as false positives with the Dataflow analysis-based prompt. This indicates that *Self Reflection can efficiently prune false positives.* On the other hand, GPT-4 incorrectly flips 1101 vulnerable samples that were reported as true positives with the Dataflow analysis-based prompt. This incorrect flipping of vulnerable samples is more prominent with the real-world datasets (CVEFixes Java and CVEFixes C/C++).

Table 11. GPT-4 Self-Corrections with Self-Reflection prompt (CWE-DF+SR vs CWE-DF).

Dataset	**Correct Self-Corrections**		**Incorrect Self-Corrections**	
	Vulnerable	Non-Vulnerable	Vulnerable	Non-Vulnerable
OWASP	0	269	32	12
Juliet Java	2	447	85	16
CVEFixes Java	2	465	292	2
Juliet C/C++	4	716	69	4
CVEFixes C/C++	4	598	623	1
Total	12	2495	1101	35

To better understand the self reflection behavior, we manually analyze 10 randomly chosen samples each from Juliet Java and CVEFixes Java, where self reflection flips predictions.

For Juliet Java, GPT-4 only flips its predictions for CWE 190 and CWE 89. In four out of ten cases, GPT-4 reasons that integer manipulations in Java are not vulnerable to Integer Overflow unless they are used in subsequent operations such as accessing array elements. It incorrectly flips its predictions on two samples where the source of Integer Overflow is a random number generator and a value read from the user's home directory, respectively. It reasons that these samples are not vulnerable to Integer Overflow because these sources are highly unlikely to generate large integers.

For CVEFixes Java, we observe that in eight cases, the model responds by saying that more context is needed to detect if the sample is vulnerable because there are calls to methods and references to objects that are not provided in the code snippet. Moreover, in three cases, the original label in the dataset was incorrect. Two of these were test methods that do not take in any user-controlled inputs but were modified in the CVE fix and hence labeled as vulnerable by the dataset, while the third method was a logging utility, which is also irrelevant to the vulnerability.

Our analysis shows that Dataflow analysis-based prompt with Self Reflection often flips predictions when the context is not self-contained, i.e., the target code snippet refers to code elements that are not local to it. We refer curious readers to Appendix A.4 for a more detailed analysis of Self Reflection with examples.

5.3 Lessons Learned

We identify key insights about how LLMs can be useful for vulnerability detection:

LLMs can leverage patterns learned during pre-training to detect security vulnerabilities. In Section 4.3, we observe that LLMs offer competitive performance on various Java and C/C++ datasets only with well-crafted prompting. Because LLMs have not been explicitly trained on the task of vulnerability detection, this capability can be attributed to the code and vulnerability *patterns* they witnessed during pre-training. Prior works point out the possibility that because

LLMs have been pre-trained on publicly available data, they might have already seen the code snippets we evaluate them on. However, this is unlikely in our case due to two reasons. First, the lower performance with Basic prompt and CWE specific prompt indicate that the task is difficult for LLMs. Second, our experiments with adversarial attacks (Section 4.8) show that they do not significantly degrade the performance of LLMs.

There are, however, two key challenges in making LLM-based vulnerability detection practical: First, the results, along with the insights in Section 5.1, show that the method-level context is often insufficient to predict if the given snippet is vulnerable. Second, it is practically infeasible to run LLMs on all methods within a codebase owing to their massive computational costs. Future work can explore these directions to make LLM usage more practical and effective. It would also be interesting to see if Static Analysis tools (such as CodeQL) can be used to extract relevant context for the models.

LLMs can complement static analysis tools in detecting more diverse vulnerability classes. In Section 4.5, we show that GPT-4 with the Dataflow analysis-based prompt with Self Reflection performs better than CodeQL on two datasets. We also observe that certain CWEs (such as CWE-78) are easier to detect with GPT-4 but are difficult to model with CodeQL queries and vice-versa. This observation presents an exciting opportunity to combine the strengths of these two different techniques to detect broader classes of vulnerabilities.

Prompting techniques such as Dataflow analysis-based prompt can make the predictions more accessible. In Section 5.2, we show that prompting techniques such as Dataflow analysis-based prompt can provide human-readable vulnerability specifications and labels. This is in major contrast to the existing Deep Learning-based vulnerability detection tools, which either only provide a binary label or flag vulnerable lines of code independently. Moreover, simple self-validation checks, such as the Self Reflection presented in Section 3.5 can prune out several false positives, which is typically a major issue with existing static analysis tools. We are working on identifying other forms of extrinsic feedback (prompt-based, or program analysis-based) that can further improve the performance of LLMs.

Vulnerability Detection datasets need to be revamped for better analysis. In Section 5.1, we investigate why LLMs perform worse on real-world datasets such as CVEFixes Java and CVEFixes C/C++. We find that one reason for this is the lack of context due to the method-level granularity of the datasets. This clearly indicates that method-level vulnerability detection may not be ideal for better performance on real-world datasets. The synthetic datasets, on the other hand, use common templates which are very easy to internalize and are not an indicator of the difficulty of real-world datasets. Moreover, the manual analysis of 10 random samples from CVEFixes Java that were incorrectly identified in Section 5.2 found 3 cases where the label was incorrect. This calls for the need for more carefully curated real-world datasets where the labels are verified for correctness. Cleaner datasets would also translate to better learning during fine-tuning because noisy labels are very likely to confuse the model's understanding of the task.

5.4 Threats to Validity

The choice of LLMs and datasets may bias our evaluation and insights. To address this threat, we choose multiple 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. Hence, our study is very comprehensive. However, our insights may not generalize to other languages or datasets not studied in this paper. Our evaluation code and scripts may have bugs, which might bias our results. To address this threat, multiple co-authors reviewed the code regularly and actively fixed issues. Our choice of prior vulnerability detection tools may not be complete. To address this threat, we compare the LLMs against one of the most popular and widely-used static analysis tool, CodeQL, as well as a state-of-the-art deep learning-based tool (LineVul) that has previously demonstrated the best vulnerability detection performance.

6 RELATED WORK

Static Analysis Tools for Vulnerability Detection. Static analysis tools search for pre-defined vulnerability patterns in code. Tools such as FlawFinder [FlawFinder 2023] and CppCheck [CPPCheck 2023] use syntactic and simple semantic analysis techniques to find vulnerabilities in C++ code. More advanced tools like CodeQL [Avgustinov et al. 2016], Infer [Fb Infer 2023], and CodeChecker [Code Checker 2023] 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, we demonstrate in this work that LLMs can automatically identify relevant code patterns through statistically learned rules from training data, and help detect security vulnerabilities in code.

Deep Learning-based Vulnerability Detection. Several works have focused on using Deep Learning techniques for vulnerability detection. Earlier works such as Devign [Zhou et al. 2019], Reveal [Chakraborty et al. 2020], LineVD [Hin et al. 2022] and IVDetect [Li et al. 2021] leveraged Graph Neural Networks (GNNs) for modeling aataflow graphs, control flow graphs, abstract syntax trees and program dependency graphs. Concurrent works explore alternate model architectures: VulDeePecker [Li et al. 2020] and SySeVR [Li et al. 2018] 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 [Fu and Tantithamthavorn 2022], UnixCoder). DeepDFA [Steenhoek et al. 2023] and ContraFlow [Cheng et al. 2022] 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. [2022] 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 Software Engineering Problems. Several recent approaches have demonstrated that LLMs can be effectively leveraged for improving the state-of-the art performance in various traditional software engineering tasks such as automated program repair [Joshi et al. 2023; Xia et al. 2023; Xia and Zhang 2022], test generation [Deng et al. 2023; Lemieux et al. 2023], code evolution [Zhang et al. 2023], and fault localization [Yang et al. 2023]. In contrast to these approaches, we focus on applying LLMs for detecting security vulnerabilities. However, in line with these works, we also find that LLMs can detect code patterns that lead to security vulnerabilities and provide a promising direction for future explorations.

7 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, show impressive results on synthetic datasets across languages. However, handling real-world vulnerable code remains a challenging task, even with fine-tuning. Hence, we believe that an interesting future direction is to develop more specialized strategies and datasets for improving the performance of LLMs on vulnerability detection.

DATA-AVAILABILITY STATEMENT

We share all our code and experiment scripts at https://figshare.com/s/4a573ad116575c7267cc. We will open-source a replication package on acceptance.

REFERENCES

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*. https://api.semanticscholar.org/CorpusID:13385963

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). https://api.semanticscholar.org/CorpusID:236087844

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.

Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* (2012).

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]

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. https://api.semanticscholar.org/CorpusID:221703797

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). https://api.semanticscholar.org/CorpusID:235755472

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). https://api.semanticscholar.org/CorpusID:250562410

Code Checker 2023. https://github.com/Ericsson/codechecker.

CodeQL CPP CWE 2023. https://codeql.github.com/codeql-query-help/cpp-cwe/.

CodeQL Java CWE 2023. https://codeql.github.com/codeql-query-help/java-cwe/.

CPPCheck 2023. https://cppcheck.sourceforge.io/.

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

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

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.

Fb Infer 2023. https://fbinfer.com/.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

FlawFinder 2023. https://dwheeler.com/flawfinder

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.

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

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

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), 596–607. https://api.semanticscholar.org/CorpusID:247362653

Hugging Face 2023. https://huggingface.co/.

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*, Vol. 37. 5131–5140.

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

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

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)*.

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.

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).

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). https://api.semanticscholar.org/CorpusID:235490574

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* 19 (2020), 2821–2837. https://api.semanticscholar.org/CorpusID:210064554

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. https://api.semanticscholar.org/CorpusID:49869471

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).

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/.

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

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

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).

Leonard Salewski, Stephan Alaniz, Isabel Rio-Torto, Eric Schulz, and Zeynep Akata. 2023. In-Context Impersonation Reveals Large Language Models' Strengths and Biases. *ArXiv* abs/2305.14930 (2023). https://api.semanticscholar.org/CorpusID:258866192

Semgrep. 2023. The Semgrep Platform. https://semgrep.dev/.

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

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366* 14 (2023).

Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2023. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. arXiv:2212.08108 [cs.SE]

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). https://api.semanticscholar.org/CorpusID:248006164

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. 2022a. Emergent Abilities of Large Language Models. *Trans. Mach. Learn. Res.* 2022 (2022). https://api.semanticscholar.org/CorpusID:249674500

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. 2022b. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv* abs/2201.11903 (2022). https://api.semanticscholar.org/CorpusID:246411621

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.*

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.

Aidan ZH Yang, Ruben Martins, Claire Le Goues, and Vincent J Hellendoorn. 2023. Large Language Models for Test-Free Fault Localization. *arXiv preprint arXiv:2310.01726* (2023).

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *ArXiv* abs/2210.03629 (2022). https://api.semanticscholar.org/CorpusID:252762395

Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-Evolution Using Large Language Models. *arXiv preprint arXiv:2307.14991* (2023).

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*. https://api.semanticscholar.org/CorpusID:202539112

A EXAMPLES

A.1 Comparison with CodeQL

```
// GPT-4 gets this wrong (the single \& doesn't prevent the second condition from being checked)
     while CodeQL gets it right

// CODE SNIPPET
#include "std_testcase.h"
void f16591()
{
    if(globalReturnsTrue())
    {
        {
            twoIntsStruct *twoIntsStructPointer = NULL;
            if ((twoIntsStructPointer != NULL) & (twoIntsStructPointer->intOne == 5))
            {
                printLine("intOne == 5");
            }
        }
    }
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:
A. Sources:
1. The function globalReturnsTrue() - it is an external API and its output is not sanitized.
2. The variable twoIntsStructPointer - it is initialized as NULL.

B. Sinks:
1. The condition check (twoIntsStructPointer != NULL) & (twoIntsStructPointer->intOne == 5) - it
     is where the NULL pointer could potentially be dereferenced.

C. Sanitizers:
1. The condition check (twoIntsStructPointer != NULL) - it is a sanitizer as it checks if the
     pointer is not NULL before dereferencing it.

D. Unsanitized Data Flows:
1. (twoIntsStructPointer, condition check, the pointer 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.)

E. Vulnerability analysis verdict: $$ vulnerability: NO | vulnerability type: CWE-476 |
     vulnerability name: NULL Pointer Dereference | explanation: The pointer twoIntsStructPointer
     is initialized as NULL and then checked if it's not NULL before dereferencing. However, the
     pointer is not assigned any value other than NULL, so it's not actually a risk of NULL pointer
      dereference. $$
```

Listing 10. CodeQL correctly predicts Null Pointer Dereferencing while GPT-4 couldn't

```
// GPT-4 gets this wrong while CodeQL gets it right (++data may lead to an Integer Overflow)

// CODE SNIPPET
#include "std_testcase.h"
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);
        }
    }
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:
A. Sources:
1. RAND64() function call
B. Sinks:
1. printLongLongLine(result) function call
C. Sanitizers:
No sanitizers are present in the code.
D. Unsanitized Data Flows:
1. (RAND64(), printLongLongLine(result), The data from RAND64() is directly used in
     printLongLongLine() without any sanitization or checks for integer overflow.)
E. Vulnerability analysis verdict: $$ vulnerability: NO | vulnerability type: CWE-190 |
     vulnerability name: Integer Overflow or Wraparound | explanation: The code does not perform
     any arithmetic operations that could lead to an integer overflow. The only operation
     performed on the data variable is incrementing it by one (++data), which is unlikely to cause
      an integer overflow given that data is a 64-bit integer. $$
```

Listing 11. CodeQL correctly predicts Integer Overflow while GPT-4 couldn't

Listings 10 and 11 present two examples where CodeQL correctly predicts that the target snippets are vulnerable to Null Pointer Dereferencing and Integer Overflow respectively while GPT-4 with the Dataflow analysis-based prompt couldn't. In the first case, the model overlooks the & which would cause both the expressions to be evaluated. In the second case, the model incorrectly states that no arithmetic operation is being performed on the variable data that is initialized with a random integer even when the code includes a data++ operation.

A.2 Poor performance on Improper Authentication

We manually examined samples corresponding to CWE-287 in the CVEFixes Java dataset and found that all false negatives correspond to CVE-2021-39177 where the patch upgrades to a newer version of a dependency that introduces some checks to prevent against Improper Authentication. The changes are therefore parameter order swaps in methods that use this dependency and it is not possible to detect that they are vulnerable. The false positives, on the other hand, are samples where the model is suspicious of every external input and hence errs on the side of always finding the snippet vulnerable. In Listing 12, for example, the model claims that the session and subject variables could be tainted when they are actually being created from an Access Control Context which might be safe but it is not evident from context. In Listing 13, the model claims that the "Authorization" header of the request may be tainted. Interestingly, the model corrects its responses with Self Reflection. In the first case, GPT-4 mentions that more context may be needed to identify if the snippet is vulnerable. In the second case, the model corrects its response by identifying sanitizers for the information collected using this header.

```
// Example where the model incorrectly predicts that the snippet is prone to Improper
     Authentication and corrects itself after Self Reflection
// TARGET CWE: CWE-287 (Improper Authentication)

// CODE SNIPPET
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException {
    HttpSession session = request.getSession(false);
    if (session == null) {
      AccessControlContext acc = AccessController.getContext();
      Subject subject = Subject.getSubject(acc);
      if (subject == null) {
        Helpers.doForbidden(response);
        return;
      }
      session = request.getSession(true);
      session.setAttribute("subject", subject);
    } else {
      Subject subject = (Subject) session.getAttribute("subject");
      if (subject == null) {
        session.invalidate();
        Helpers.doForbidden(response);
        return;
      }
    }

    String encoding = request.getHeader("Accept-Encoding");
    boolean supportsGzip = (encoding != null && encoding.toLowerCase().indexOf("gzip") > -1);
    SessionTerminal st = (SessionTerminal) session.getAttribute("terminal");
    if (st == null || st.isClosed()) {
        st = new SessionTerminal(getCommandProcessor(), getThreadIO());
        session.setAttribute("terminal", st);
    }
    String str = request.getParameter("k");
    String f = request.getParameter("f");
    String dump = st.handle(str, f != null && f.length() > 0);
    if (dump != null) {
        if (supportsGzip) {
            response.setHeader("Content-Encoding", "gzip");
            response.setHeader("Content-Type", "text/html");
            try {
                GZIPOutputStream gzos = new GZIPOutputStream(response.getOutputStream());
                gzos.write(dump.getBytes());
                gzos.close();
            } catch (IOException ie) {
                LOG.info("Exception writing response: ", ie);
            }
        } else {
            response.getOutputStream().write(dump.getBytes());
        }
    }
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
```

```
Here is a data flow analysis of the given code snippet:
A. Sources:
1. HttpServletRequest request
2. HttpSession session
3. AccessControlContext acc
4. Subject subject
5. String encoding
6. SessionTerminal st
7. String str
8. String f

B. Sinks:
1. HttpServletResponse response
2. HttpSession session
3. GZIPOutputStream gzos

C. Sanitizers:
1. The code checks if the session is null and if the subject is null, and if either is true, it
     sends a forbidden response.
2. The code checks if the SessionTerminal is null or closed, and if true, it creates a new
     SessionTerminal.

D. Unsanitized Data Flows:
1. (HttpServletRequest request, HttpServletResponse response, The request parameters "k" and "f"
     are directly used to handle the session terminal and the result is written to the response
     without any sanitization. This could lead to potential security vulnerabilities if the input
     is malicious.)
2. (HttpSession session, HttpServletResponse response, The session attribute "subject" is directly
      used without any sanitization. If the session attribute is manipulated, it could lead to
     potential security vulnerabilities.)

E. Vulnerability analysis verdict: $$ vulnerability: YES | vulnerability type: CWE-287 |
     vulnerability name: Improper Authentication | explanation: The code does not properly
     authenticate the user before allowing access to sensitive information. The session and
     subject are checked for null values, but there is no check to ensure that the user is who they
      claim to be. This could allow an attacker to impersonate a legitimate user and gain
     unauthorized access to sensitive information. $$

// RESPONSE WITH SELF REFLECTION
The analysis is incorrect. The code does perform authentication checks. It checks if the session
     and the subject are null and if they are, it sends a forbidden response. This is a form of
     authentication. The session attribute "subject" is not directly used without sanitization, it
      is checked for null values. The request parameters "k" and "f" are used in the handle method
      of the SessionTerminal object, without knowing the implementation of this method, we cannot
     definitively say that there is no sanitization happening there. Therefore, we cannot confirm
     that there is an Improper Authentication vulnerability.
Final Vulnerability analysis verdict: $$ vulnerability: NO | vulnerability type: CWE-287 $$
```

Listing 12. GPT-4 incorrectly predicts that a snippet is prone to Improper Authentication (corrected by Self Reflection)

```
// Example where the model incorrectly predicts that the snippet is prone to Improper
     Authentication but corrects its response with Self Reflection
// TARGET CWE: CWE-287 (Improper Authentication)

// CODE SNIPPET
```

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
     IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse rsp = (HttpServletResponse) response;
    String authorization = req.getHeader("Authorization");

    if (authorization!=null) {
                   String uidpassword = Scrambler.descramble(authorization.substring(6));
        int idx = uidpassword.indexOf(':');
        if (idx >= 0) {
            String username = uidpassword.substring(0, idx);
            try {
                Jenkins.getInstance().getSecurityRealm().loadUserByUsername(username);
            } catch (UserMayOrMayNotExistException x) {
                               } catch (UsernameNotFoundException x) {
                                   chain.doFilter(request, response);
                return;
            } catch (DataAccessException x) {
                throw new ServletException(x);
            }
            String password = uidpassword.substring(idx+1);

                           User u = User.get(username);
            ApiTokenProperty t = u.getProperty(ApiTokenProperty.class);
            if (t!=null && t.matchesPassword(password)) {
                                                      SecurityContext oldContext = ACL.impersonate
     (u.impersonate());
                try {
                    request.setAttribute(ApiTokenProperty.class.getName(), u);
                    chain.doFilter(request,response);
                    return;
                } finally {
                    SecurityContextHolder.setContext(oldContext);
                }
            }
        }
    }

    chain.doFilter(request,response);
}

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:
A. Sources:
1. HttpServletRequest req = (HttpServletRequest) request;
2. String authorization = req.getHeader("Authorization");

B. Sinks:
1. chain.doFilter(request, response);
2. chain.doFilter(request,response);

C. Sanitizers:
1. Scrambler.descramble(authorization.substring(6));
2. Jenkins.getInstance().getSecurityRealm().loadUserByUsername(username);
3. ApiTokenProperty t = u.getProperty(ApiTokenProperty.class);
4. t.matchesPassword(password);
```

```
D. Unsanitized Data Flows:
1. (req.getHeader("Authorization"), chain.doFilter(request, response), The authorization header is
      directly used without any sanitization or validation. If the header is manipulated, it could
      lead to improper authentication.)
2. (req.getHeader("Authorization"), chain.doFilter(request,response), The authorization header is
     directly used without any sanitization or validation. If the header is manipulated, it could
     lead to improper authentication.)

E. Vulnerability analysis verdict: $$ vulnerability: YES | vulnerability type: CWE-287 |
     vulnerability name: Improper Authentication | explanation: The code is prone to CWE-287
     because it uses the "Authorization" header directly without any validation. If the header is
     manipulated, it could lead to improper authentication. $$

// RESPONSE AFTER SELF-REFLECTION
The initial analysis is incorrect. The code does perform authentication checks. The "Authorization
     " header is not used directly without validation. It is first descrambled using the Scrambler.
     descramble() method. Then, the username is extracted and checked if it exists using the
     loadUserByUsername() method. If the user exists, the password is checked using the
     matchesPassword() method. Therefore, the code is not prone to CWE-287 as it does not allow
     improper authentication.
Final Vulnerability analysis verdict: $$ vulnerability: NO | vulnerability type: CWE-287 $$
```

Listing 13. GPT-4 incorrectly predicts that a snippet is prone to Improper Authentication (lack of context indicated by Self Reflection)

A.3 Dataflow analysis-based prompt

We now present two examples where the dataflow analysis from the CWE-DF prompt is correct and partially correct respectively. First, consider the code snippet in Listing 14. The model correctly identifies that the method creates a directory using the input parameters without sanitizing for CWE-22 (path traversal). On the other hand, consider the code snippet in Listing 15. Here the model correctly identifies the sources, sinks and sanitizers but includes an incorrect data flow path under unsanitized paths. The final response is correct since the second unsanitized path found by the model is indeed tainted.

Some other example responses from GPT-4 using the CWE-DF prompt are presented in Listing 16 and Listing 17

```
// 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 14. Correct response by GPT-4 on a Path Traversal vulnerability

```java
// CODE SNIPPET
protected List<String> getRawCommandLine( String executable, String[] arguments )
{
    List<String> commandLine = new ArrayList<String>();
    StringBuilder sb = new StringBuilder();

    if ( executable != null )
    {
        String preamble = getExecutionPreamble();
        if ( preamble != null )
        {
            sb.append( preamble );
        }

        if ( isQuotedExecutableEnabled() )
        {
            char[] escapeChars = getEscapeChars( isSingleQuotedExecutableEscaped(),
     isDoubleQuotedExecutableEscaped() );

            sb.append( StringUtils.quoteAndEscape( getExecutable(), getExecutableQuoteDelimiter(),
      escapeChars, getQuotingTriggerChars(), '\\', false ) );
        }
        else
        {
            sb.append( getExecutable() );
        }
    }
    for ( int i = 0; i < arguments.length; i++ )
    {
        if ( sb.length() > 0 )
        {
            sb.append( " " );
        }

        if ( isQuotedArgumentsEnabled() )
        {
            char[] escapeChars = getEscapeChars( isSingleQuotedArgumentEscaped(),
     isDoubleQuotedArgumentEscaped() );

            sb.append( StringUtils.quoteAndEscape( arguments[i], getArgumentQuoteDelimiter(),
     escapeChars, getQuotingTriggerChars(), getArgumentEscapePattern(), false ) );
        }
        else
        {
            sb.append( arguments[i] );
        }
    }

    commandLine.add( sb.toString() );

    return commandLine;
}

// PARTIALLY CORRECT RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:
A. Sources:
1. The "executable" parameter of the getRawCommandLine method.
2. The "arguments" array parameter of the getRawCommandLine method.

B. Sinks:
1. The commandLine list that is returned by the getRawCommandLine method.

C. Sanitizers:
1. The StringUtils.quoteAndEscape method is used to sanitize the "executable" and "arguments"
     parameters.

D. Unsanitized Data Flows:
// INCORRECT DATAFLOW
1. (executable, commandLine, The executable parameter is appended to the commandLine without
     sanitization when isQuotedExecutableEnabled returns false)
// CORRECT DATAFLOW
2. (arguments, commandLine, The arguments parameter is appended to the commandLine without
     sanitization when isQuotedArgumentsEnabled returns false)

// PARTIALLY CORRECT RESPONSE WITH SELF REFLECTION
E. Vulnerability analysis verdict: $$ vulnerability: YES | vulnerability type: CWE-78 |
     vulnerability name: OS Command Injection | explanation: The code is vulnerable to OS command
     injection because it directly appends the "executable" and "arguments" parameters to the
     commandLine without sanitization when isQuotedExecutableEnabled and isQuotedArgumentsEnabled
     return false. An attacker could potentially inject malicious commands through these
     parameters. $$
```

Listing 15. Partially Correct response by GPT-4 on an OS Command Injection vulnerability

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

// CODE SNIPPET
```

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

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

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

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

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

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

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

Listing 16. GPT-4 with CWE-DF prompt on a real-world Out of Bounds vulnerability

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

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

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

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

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

B. Sinks:
1. The 'write' function

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

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

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

Listing 17. GPT-4 with CWE-DF prompt on a real-world Integer Overflow vulnerability

A.4 Dataflow analysis-based prompt with Self Reflection

```
// TARGET CWE: CWE-22 (Improper Limitation of a Pathname to a Restricted Directory ('Path
     Traversal'))

// CODE SNIPPET
public void process(InputStream in, ZipEntry zipEntry) throws IOException {
    String root = getRootName(zipEntry.getName());
    if (rootDir == null) {
    rootDir = root;
    }
    else if (!rootDir.equals(root)) {
    throw new ZipException("Unwrapping with multiple roots is not supported, roots: " + rootDir +
     ", " + root);
    }

    String name = mapper.map(getUnrootedName(root, zipEntry.getName()));
    if (name != null) {
        File file = new File(outputDir, name);

        if (name.indexOf("..") != -1 && !file.getCanonicalPath().startsWith(outputDir.
     getCanonicalPath())) {
          throw new ZipException("The file "+name+" is trying to leave the target output directory
      of "+outputDir+". Ignoring this file.");
        }

        ...

        FileUtils.copy(in, file);
    }
}

// RESPONSE WITH DATAFLOW INSPIRED PROMPT
Here is a data flow analysis of the given code snippet:
A. Sources:
1. InputStream in
2. ZipEntry zipEntry

B. Sinks:
1. File file

C. Sanitizers:
1. The check for ".." in the name and the comparison of the canonical paths of the file and output
      directory.

D. Unsanitized Data Flows:
1. (InputStream in, File file, The InputStream 'in' is directly copied to the file without any
     sanitization. If the InputStream contains malicious data, it could lead to various
     vulnerabilities.)
2. (ZipEntry zipEntry, File file, The name of the ZipEntry is used to create a file without any
     sanitization. If the ZipEntry name is manipulated, it could lead to path traversal attacks.)

E. Vulnerability analysis verdict: $$ vulnerability: YES | vulnerability type: CWE-22 |
     vulnerability name: Improper Limitation of a Pathname to a Restricted Directory ('Path
     Traversal') | explanation: The code does check for path traversal attacks by looking for ".."
      in the file name and comparing the canonical paths of the file and output directory. However,
      this check is not foolproof and can be bypassed. For example, an attacker could use encoded
     dot characters (%2E%2E) to bypass the check. Therefore, the code is vulnerable to CWE-22. $$

// RESPONSE WITH DATAFLOW ANALYSIS INSPIRED PROMPT + SELF REFLECTION
No
```

```
explanation: The analysis is not entirely correct. The code does indeed check for path traversal
     attacks by looking for ".." in the file name and comparing the canonical paths of the file
     and output directory. This is a common and effective way to prevent path traversal attacks.
     The claim that this check can be bypassed by using encoded dot characters (%2E%2E) is
     incorrect because Java's File and Path APIs do not automatically decode URL-encoded
     characters.
Final Vulnerability analysis verdict: $$ vulnerability: NO | vulnerability type: CWE-22$$
```

Listing 18. Self Reflection helps model correct its reasoning

What about the cases that Self Reflection correctly flips the label? Consider the code snippet in Listing 18. The method `process` takes an `InputStream in` and a `ZipEntry zipEntry` as arguments, creates a `File file` with `name` from `zipEntry` and copies the content of `in` to `file`. This method would be prone to CWE-22, Path traversal, if it did not sanitize for ".." in `name` (restricting access to any restricted directory outside the permitted path) or checked whether the canonical path of `file` starts with the permitted output directory (restricting access to absolute restricted paths such as `/root`). Since the method incorporates sanitizers for these loopholes, it is not vulnerable to path traversal. While GPT-4 with the CWE-DF prompt correctly reasons about the sources, sinks and sanitizers, it predicts that this snippet is vulnerable since the checks can be bypassed using encoded dot characters. While this is true in many cases (and can often lead to CWE-22), Java's File APIs do not automatically decode encoded dot characters and hence the encoded paths cannot be used to access unrestricted directories as correctly pointed out by the response using the CWE-DF+SR prompt.

B DISTRIBUTION OF POPULAR SECURITY VULNERABILITIES IN SELECTED DATASETS

Table 12 presents the top 25 most dangerous vulnerabilities from MITRE.org [MITRE Top 25 CWEs 2023]. Column **(CWE ID) Description** presents the ID and name of each type of vulnerability, which is typically referred to by a Common Weakness Enumeration (or CWE) ID. The next columns present the distribution of vulnerable and non-vulnerable samples for each CWE across datasets. We observe that all datasets have a good distribution of vulnerable/non-vulnerable samples for most CWEs. Some CWEs, such as CWE 863 and CWE 798, may be rarer in real-world datasets like CVEFixes Java and CVEFixes C/C++.

Table 12. Top 25 most vulnerable CWEs 2023 (vulnerable/non-vulnerable).

(CWE ID) Description	OWASP	Juliet Java	CVEFixes Java	Juliet C++	CVEFixes C++
(787) Out-of-bounds Write	-	-	19/19	-	10/24
(79) Cross Site Scripting	246/209	-	56/148	-	-
(89) SQL Injection	272/232	331/313	5/50	-	-
(416) Use After Free	-	-	-	66/74	98/130
(78) OS Command Injection	126/125	58/74	23/26	449/488	-
(20) Improper Input Validation	-	-	32/63	-	377/254
(125) Out-of-bounds Read	-	-	18/18	-	47/46
(22) Path Traversal	133/135	-	111/191	-	4/4
(352) Cross-Site Request Forgery	-	-	42/104	-	-
(434) Unrestricted Upload of File with Dangerous Type	-	-	3/4	-	-
(862) Missing Authorization	-	-	69/76	-	2/3
(476) NULL Pointer Dereference	-	34/30	7/8	49/40	84/111
(287) Improper Authentication	-	-	126/146	-	4/4
(190) Integer Overflow	-	593/567	3/4	404/430	16/26
(502) Deserialization of Untrusted Data	-	-	40/62	-	-
(77) Command Injection	-	-	19/44	-	-
(119) Improper Restriction of Operations within the Bounds of a Memory Buffer	-	-	-	-	164/211
(798) Use of Hard-coded Credentials	-	-	2/6	-	-
(918) Server-Side Request Forgery (SSRF)	-	-	60/64	-	-
(306) Missing Authentication for Critical Function	-	-	20/17	-	-
(362) Race Condition	-	-	6/18	-	148/187
(269) Improper Privilege Management	-	-	-	-	13/16
(94) Code Injection	-	-	12/24	-	-
(863) Incorrect Authorization	-	-	15/30	-	1/2
(276) Incorrect Default Permissions	-	-	-	-	5/9