LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks

Saad Ullah  
Boston University  
saadu@bu.edu

Mingji Han  
Boston University  
mjhan@bu.edu

Saurabh Pujar  
IBM Research  
saurabh.pujar@ibm.com

Hammond Pearce  
UNSW Sydney  
hammond.pearce@unsw.edu.au

Ayse Coskun  
IBM Research  
acoskun@bu.edu

Gianluca Stringhini  
UNSW Sydney  
gian@bu.edu

Abstract—Large Language Models (LLMs) have been suggested for use in automated vulnerability repair, but benchmarks showing they can consistently identify security-related bugs are lacking. We thus develop SecLLMHolmes, a fully automated evaluation framework that performs the most detailed investigation to date on whether LLMs can reliably identify and reason about security-related bugs. We construct a set of 228 code scenarios and analyze eight of the most capable LLMs across eight different investigative dimensions using our framework. Our evaluation shows LLMs provide non-deterministic responses, incorrect and unfaithful reasoning, and perform poorly in real-world scenarios. Most importantly, our findings reveal significant non-robustness in even the most advanced models like ‘PaLM2’ and ‘GPT-4’: by merely changing function or variable names, or by the addition of library functions in the source code, these models can yield incorrect answers in 26% and 17% of cases, respectively. These findings demonstrate that further LLM advances are needed before LLMs can be used as general purpose security assistants.

1. Introduction

Large Language Models (LLMs), such as OpenAI’s Codex [1], Google’s PaLM2 [2], Meta’s Codellama [3], and StarCoder [4], etc., have demonstrated great potential in performing programming-language related tasks such as code generation, code documentation, and debugging. In 2022, around 1.2 million developers used Copilot, and since then we have witnessed the release of increasingly capable LLM models at a quick pace [2]–[5]. LLMs could be particularly useful to help developers with their cybersecurity needs, as humans typically produce and miss many security relevant bugs. This issue was highlighted in the 2022 GitLab Survey [6], noting that “developers do not find enough bugs early enough” and “do not prioritize the bug remediation” when developing. It is pertinent then to investigate if LLMs could be an aid towards early identification of security problems, especially as LLMs have already been suggested for use in automated bug repair [7].

In this paper, we aim to answer the following question: Can LLMs be used as helpful security assistants for vulnerability detection? This is an important question, especially as LLMs are not infallible in security-related tasks, for example introducing vulnerabilities into source code [8], [9] and software testing [10]. Unfortunately, there is no standardized and automated approach to evaluate the performance of LLMs at identifying vulnerable code. We fill this gap by introducing SecLLMHolmes, a generalized, fully automated, and scalable framework to systematically evaluate the performance (i.e., accuracy and reasoning capabilities) of LLMs for vulnerability detection. Our framework tests the capabilities of a given LLM as a security assistant across eight distinct dimensions: (1) deterministic response, (2) performance over range of parameters, (3) diversity of prompts, (4) faithful reasoning, (5) evaluation over variety of vulnerabilities, (6) assessment of various code difficulty levels, (7) robustness to code augmentations, and (8) use in real-world projects.

We apply our framework to eight of the most capable LLMs across 228 code scenarios spanning over 8 most critical vulnerabilities in C and Python, and show that: (a) LLM performance varies widely depending on the model and the prompting technique used, however all models analyzed have a high false positive rate (FPR), and flag code where vulnerabilities have been patched as still vulnerable. (b) the output of LLMs is non-deterministic, with all models changing their answers over multiple runs for one or more of our tests. (c) even when they correctly identify a vulnerability, the reasoning that LLMs provide for this decision is often incorrect, questioning their trustworthiness. (d) LLM chain-of-thought reasoning [11] is not robust, and can be ‘confused’ by even simple code augmentations such as whitespace modification, changing function names, or using different but related library functions. Also, (e) LLMs fail at detecting vulnerabilities in real-world projects. Our study provides significant evidence that LLMs are not yet ready to be used for automated vulnerability detection, and the successful usage of our framework as a benchmark suite by future models would demonstrate meaningful progress in this space.

This paper makes the following contributions:

- We develop SecLLMHolmes, a comprehensive framework to test LLMs for their ability to identify and reason about software vulnerabilities. Our framework is fully automated and includes a set of 228 code scenarios,
and 17 prompting techniques. We publicly release our framework and dataset\(^1\), allowing the community to test newly developed LLMs and easily keep track of their progress in being able to identify vulnerabilities.

- We use our framework to test eight state-of-the-art LLMs for the task of vulnerability detection, showing that as of today no LLM achieves satisfactory performance at it.
- We identify and enumerate a set of shortcomings that current LLMs show (as outlined above). Our observations provide a checklist for researchers working in this space, showing aspects that need to be addressed before LLMs can be considered ready to be used in the wild for the task of vulnerability detection.

2. Background and Related Work

**Large Language Models (LLMs).** All language models work on the basic principle of next word (token) prediction; i.e., given a sequence of words (tokens) \(x_1, x_2, \ldots, x_{n-1}\) select a word (token) \(x_n\) with the highest probability to appear next in the sequence

\[
x_n = \arg \max_{w \in V} P(w|x_1, x_2, \ldots, x_{n-1}),
\]

where \(V\) is the vocabulary of the model. Language models learn to perform this task by training on a large amount of text data (i.e., natural language text or code) and use various techniques (e.g., attention mechanism\([12]\)) to learn to focus on certain parts of the input for better output prediction. Language models have shown excellent proficiency in NLP tasks, as well as good results for programming language tasks such as code generation, code suggestion, natural language querying for code, etc.

The recent drastic increase in the number of parameters of models has enabled several remarkable capabilities, the most prominent of which being zero-shot and few-shot learning\([13],[14]\). LLMs are typically prompted (i.e., queried) by the user and provide a response—these advances enable the prompt to provide new knowledge or instructions that the model was not trained over.

Approaches like instruction-tuning ‘teach’ LLMs how to follow instructions in their prompt responses, and reinforcement learning from human feedback is used to ‘teach’ them how to answer, converse, and reason like humans. This has led to the creation of several chat-based LLMs, which can interact conversationally with human inputs (see Figure 1). The chat-based LLMs can be prompted using various techniques:

- **In Zero-Shot (ZS) scenario,** the user asks a model to perform a task that the model might not have observed during pre-training.
- **In Few-Shot (FS) scenario,** the user adds a few examples demonstrating input space and expected output to perform a specific task (as shown in Figure 1).
- **Task-Oriented (TO) scenarios** explicitly assign a task to the model (either in ‘system’ or ‘user’ prompt) in the form of a statement or a question, which encourages the model to generate a task-specific response.
- **Role-Oriented (RO) scenarios** assign a role to the model, e.g., helpful assistant, security expert, etc., and the model implicitly understands the expected behavior. This role is mostly assigned in the ‘system’ prompt.

Table 1 shows the details of the currently most capable chat-based LLMs that we investigate in this paper.

### Vulnerability Detection

One of the biggest challenges during the development and maintenance of software systems is the process of detecting security bugs and identifying their root cause\([6]\). OWASP\([15]\) presents a list of static and runtime analysis tools and techniques that work in this space. Most of these tools transform source code into a specific representation, e.g., abstract syntax tree, program dependency graph, code property graph, etc., and scan them to identify pre-defined insecure patterns. For instance, Yamaguchi \textit{et al.} introduce Joern, a tool that uses code property graphs\([16]\) to identify several vulnerability patterns in C/C++ code.

Unfortunately, these techniques require considerable manual effort and curation of security bug datasets, especially if they are based on trained supervised Machine Learning (ML) models. Examples of methods that use supervised ML include VulChecker\([17]\), VulDeePecker\([18]\), Poster\([19]\), and SySeVR\([20]\). As the ratio of vulnerable to non-vulnerable examples is very low in the real world, the vulnerable examples in these datasets are not sufficient for ML models to learn the necessary information. Suggested improvements in this category include using pre-trained LLMs for code and ‘fine-tuning’ them on security bug data.

---

\(^1\) https://github.com/ai4cloudops/SecLLMHolmes

---

### Table 1: Studied LLMs

<table>
<thead>
<tr>
<th>Model API</th>
<th>Base Model</th>
<th># Params</th>
<th>Max. Tokens</th>
<th>Type Knowledge Cut Off</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpt-4</td>
<td>GPT-4</td>
<td>1.76T</td>
<td>8,192</td>
<td>Rem 09/2021</td>
</tr>
<tr>
<td>gpt-3.5-turbo-16k</td>
<td>GPT-3.5</td>
<td>175B</td>
<td>16,385</td>
<td>Rem 09/2021</td>
</tr>
<tr>
<td>codechat-bison@001</td>
<td>PaLM2</td>
<td>340B</td>
<td>6,144</td>
<td>Rem mid-2021</td>
</tr>
<tr>
<td>chat-bison@001</td>
<td>PaLM2</td>
<td>340B</td>
<td>6,144</td>
<td>Rem mid-2021</td>
</tr>
<tr>
<td>codellama-7b-instruct</td>
<td>Llama2</td>
<td>7B</td>
<td>100k</td>
<td>Loc 01/2022</td>
</tr>
<tr>
<td>codellama-13b-instruct</td>
<td>Llama2</td>
<td>13B</td>
<td>100k</td>
<td>Loc 01/2022</td>
</tr>
<tr>
<td>codellama-34b-instruct</td>
<td>Llama2</td>
<td>34B</td>
<td>100k</td>
<td>Loc 01/2022</td>
</tr>
<tr>
<td>starchat-beta StarCoder+</td>
<td>15.5B</td>
<td>8,192</td>
<td>6,144</td>
<td>Loc 09/2022</td>
</tr>
</tbody>
</table>

---

**Figure 1:** LLM chat input format. LLMs operate on a three-part input format: (1) a system prompt, (2) few-shot examples presented as chat history to guide the model’s learning, and (3) the specific user input/task to be processed.
TABLE 2: Evaluation studies for code security. This table describes if the evaluation is (✓) semi or (X) fully) Automated, Evaluates Reasoning (root cause) ✓ or only performs binary evaluation of robustness, tests Code-level Robustness, evaluates both ✓ or just one ✓ of the Vulnerable and Patched code scenarios, evaluates on Real-World CVEs ✓ or github mined potential defect commits [33], and # of Code Scenarios included in the study, which are not generated by synthetic AI methods or labeled using out-dated research tools.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vuln. Code Gen. [8]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>Transf. Vuln. Det. [34]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>res.</td>
</tr>
<tr>
<td>SecLLMHolmes</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>228</td>
</tr>
</tbody>
</table>

datasets for the downstream task of vulnerability detection. Notable approaches here are UniXCoder [21], VulBERTa [22], and CoText [23].

**Evaluating for Code Security.** Evaluating approaches for identifying security weaknesses in code requires testing their performance on multiple axes. For instance, what are their accuracy and false positive rate (FPR)? Are reasons/root causes for a vulnerability provided by the tool, and if so, at what quality? Is the tool robust to noise in testing data? How many types of vulnerabilities can it detect? Static analysis tools have always struggled with the trade-off between accuracy and coverage. Tools either focus on high accuracy and low coverage (e.g., Pysa [24] by Meta, which identifies data-flow issues in Python applications), or (b) low accuracy (false positives) and high coverage (e.g., [25]–[27]), which can lead to developer frustration and wasted time.

ML-based static analysis tools [18], [28] not only face the accuracy-coverage trade-off but also struggle with robustness. Such tools have demonstrated good performance on research datasets but can fail to generalize and perform well over real-world datasets [29]. Risse et al. [30] identify the same issue of non-robustness in LLM-based vulnerability detection tools [22], [23], [31], and showcase the drop in accuracy with even trivial code augmentations. Moreover, Microsoft’s leader-board for LLM-based vulnerability detection tools [32] shows that the even the best tool has an accuracy of less than 70% and 30%+ FPR, which shows that these systems cannot be trusted in real-world cases.

Recent works have evaluated LLMs for vulnerable code generation [8], code repair [7], and vulnerability detection [34]. However, these approaches are either limited by the number of LLMs, coverage of vulnerabilities, diversity of prompts, range of code complexities, robustness testing, or require manual labor for evaluating LLMs. Most importantly, these studies only evaluate binary responses, checking whether the LLM gives the right label to code snippets (e.g., ‘vulnerable’ or ‘not vulnerable’). In this paper, we present the first comprehensive evaluation framework to evaluate LLMs on the task of vulnerability detection, providing a multi-faceted analysis of the capabilities of LLMs, including going beyond binary decisions and evaluating their reasoning abilities. Table 2 summarizes evaluation studies of LLMs, highlights their short-comings, and compares against our framework.

In addition to presenting a new state-of-the-art framework, our study uncovers previously unidentified challenges associated with LLMs, including their inability to understand complex code data-flows, the fragility of their COT or step-by-step reasoning to even trivial augmentations, a bias towards security-related functions and variable names that overlooks actual vulnerabilities, a reasoning process that diverges from the methodology of real-world human security experts, particularly in accurately identifying correct root cause of a vulnerability.

### 3. SecLLMHolmes

Figure 2 presents an overview of our fully automated framework, SecLLMHolmes, which is designed to be applicable to any chat-based LLM. Section 3.1 describes how users can configure an LLM for integration. The core of our framework consists of five pre-defined key components: (i) a set of parameters (Section 3.2), (ii) an extensive set of prompt templates (Section 3.3), (iii) datasets (Section 3.4), and (iv) code augmentations (Section 3.4), all of which facilitate the generation of test prompts. Each prompt is then passed to the LLM to generate a response, and the quality of the response is then analyzed by the (v) ‘Evaluator’ module (Section 3.6).

#### 3.1. LLM configuration

To scale SecLLMHolmes for any chat-based LLM, we have established three configurable user inputs:

**LLM-Specific Best ‘Prompting Practices’ and Rules.** As each LLM may be tuned differently with respect to instruction following, this configurable input enables users to configure optimal prompting practices for the specific LLM they are integrating. This may be done following the LLM’s documentation. For example, OpenAI’s GPT documentation recommends the use of three quotes before and after the given content to separate it from the instruction [35], while Google’s PaLM2 documentation recommends the use of

probability mass. We discuss how the nucleus sampling, where the LLM considers the results of the tokens with top parameters which control token sampling: (1) temperature controls the determinism of an LLM’s output. A higher value of temperature generates more ‘creative’/‘random’ text by adding noise to potential token scores (2) top_p controls the nucleus sampling, where the LLM considers the results of the tokens with top_p probability mass. We discuss how we select the values of these parameters in Section 4.

3.2. LLM Parameters

LLM responses are significantly impacted by two parameters which control token sampling: (1) temperature controls the output of the LLM’s deterministic sampling. A higher value of temperature can generate more ‘creative’/‘random’ text by adding noise to potential token scores. (2) top_p controls the nucleus sampling, where the LLM considers the results of the tokens with top_p probability mass. We discuss how we select the values of these parameters in Section 4.

3.3. Prompt Templates

SecLLMHolmes explores four techniques for LLM prompting: (1) zero-shot task-oriented (ZS - TO), (2) zero-shot role-oriented (ZS - RO), (3) few-shot task-oriented (FS - TO), and (4) few-shot role-oriented (FS - RO). Moreover, we divide our set of prompts into the following three categories: (all prompts are described in Table 3):

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>ZS-TO</td>
<td>Code snippet is added to the input prompt with a question about a specific Common Weakness Enumeration (CWE) (e.g., out-of-bound write, path traversal).</td>
</tr>
<tr>
<td>S2</td>
<td>ZS-RO</td>
<td>Same as S1, but the LLM is assigned the role of a ‘helpful assistant’.</td>
</tr>
<tr>
<td>S3</td>
<td>ZS-RO</td>
<td>Similar to S1, with the LLM acting as a ‘security expert’.</td>
</tr>
<tr>
<td>S4</td>
<td>ZS-RO</td>
<td>The LLM is defined as a ‘security expert’ who analyzes a specified security vulnerability, without the question being added to the input prompt.</td>
</tr>
<tr>
<td>S5</td>
<td>FS-TO</td>
<td>Similar to S1, but includes a vulnerable example, its patch, and standard reasoning from the same CWE.</td>
</tr>
<tr>
<td>S6</td>
<td>FS-RO</td>
<td>Like S4, but also includes a vulnerable example, its patch, and standard reasoning from the same CWE.</td>
</tr>
<tr>
<td>R1</td>
<td>ZS-TO</td>
<td>Similar to S1, but begins with “Lets think step by step” [37] to encourage a methodical approach.</td>
</tr>
<tr>
<td>R2</td>
<td>ZS-RO</td>
<td>The LLM plays the role of a security expert with a multi-step approach to vulnerability detection, following a chain-of-thought reasoning.</td>
</tr>
<tr>
<td>R3</td>
<td>ZS-TO</td>
<td>A multi-round conversation with the LLM, starting with a code snippet and progressively analyzing sub-components for a security vulnerability like human security-experts.</td>
</tr>
<tr>
<td>R4</td>
<td>FS-RO</td>
<td>Similar to S6, but the reasoning for answers involves step-by-step analysis developed by the first author.</td>
</tr>
<tr>
<td>R5</td>
<td>FS-RO</td>
<td>Like R2, but includes few-shot examples (from the same CWE) with step-by-step reasoning for detecting vulnerabilities.</td>
</tr>
<tr>
<td>R6</td>
<td>FS-TO</td>
<td>Similar to R5, but does not assign a specific role to the LLM in the system prompt.</td>
</tr>
<tr>
<td>D1</td>
<td>ZS-TO</td>
<td>Adds the definition of a security vulnerability to the input prompt, followed by a related question.</td>
</tr>
<tr>
<td>D2</td>
<td>ZS-RO</td>
<td>The LLM is a security expert analyzing code for a specific vulnerability, with the vulnerability’s definition included.</td>
</tr>
<tr>
<td>D3</td>
<td>FS-RO</td>
<td>Similar to S6, but includes the definition of the security vulnerability in the system prompt.</td>
</tr>
<tr>
<td>D4</td>
<td>FS-RO</td>
<td>Like R4, with the addition of the security vulnerability’s definition in the system prompt.</td>
</tr>
<tr>
<td>D5</td>
<td>FS-TO</td>
<td>Similar to D4, but does not assign a specific role to the LLM in the system prompt.</td>
</tr>
</tbody>
</table>

MITRE’s official website [41] to the model, while asking the model to detect that vulnerability in the given code.

3.4. Datasets

We design 228 code scenarios (48 hand-crafted, 30 real-world, and 150 with code augmentations) to test various aspects of the capabilities of LLMs to detect software vulnerabilities in code. We use these scenarios to craft prompts by including code, examples, definitions, and step-by-step reasoning as shown in Table 3. In the following, we describe how we built our code scenarios in detail.

**Hand-Crafted CWE Scenarios.** We curate a dataset of 48 hand-crafted code scenarios, containing vulnerable and patched pairs from 8 most critical and diverse Common
Weakness Enumerations (CWEs) from the MITRE Top 25 Most Dangerous Software Weaknesses for the year 2023 [41], as shown in Table 4. To investigate the ability of LLMs to analyze multiple programming languages, we include examples from both C and Python.

Similar to previous work [8], we create six code scenarios (three pairs of vulnerable and patched scenarios) for each CWE. Moreover, we design our code scenarios with three difficulty levels, (1) easy, (2) medium, (3) hard. Code scenario ‘2,3’ in a specific CWE represents the vulnerable scenario with ‘medium’ difficulty level and ‘2p’ its patch. The difficulty levels assess how LLMs interact with code of increasing complexity. Easy scenarios consist of simple programs containing only one function and less than 30 lines of code. Medium level scenarios increase the complexity by making the program longer, using different library functions, and adding more than one user input. Hard level introduces scenarios with multiple functions in which functions can be safe on an individual level but when they work together they make the program vulnerable.2

Real-World CVE Scenarios. We leverage a set of real-world Common Vulnerabilities and Exposures (CWEs) from public open source projects to investigate if LLMs are able to identify vulnerabilities in them. Note that existing benchmarks for vulnerability detection [18], [33], [42], [43], cannot be used for this project, as they were released before the cut-off training date of current LLMs, and it is therefore likely that the models saw that data during training. To avoid this potential confounder, we curate 30 code scenarios containing vulnerable and patched versions of 15 CVEs from four open source projects, all published and fixed in 2023, after current LLMs were trained (see Table 5).

As the length of the code increases significantly for the real-world code scenarios, it can exceed the maximum number of tokens a model can take as an input. To solve this problem, we shorten code files by removing comments and functions that are neither called by nor call the vulnerable (or patched) function. Also to maintain fairness among the LLMs, we make sure that all CVEs after truncation have a number of tokens less than or equal to 6,144, which is the maximum token limit supported by Palm2, and the lowest among all LLMs in our study.

Code Augmentations. While standard frameworks exist to evaluate the robustness of LLMs for NLP tasks [44], [45], there is no standard framework to evaluate the robustness on code security related tasks. To fill this gap, we design a set of 150 augmented code scenarios, meticulously crafted and reviewed to preserve the ability of human security experts to identify vulnerabilities. These augmentations are organized in two distinct categories:

1) Trivial Augmentations. This class of code augmentations measure the robustness of LLMs to random noise. We select 12 CWE scenarios from two classes, i.e., CWE-787 (C) (#1 MITRE) and CWE-89 (Py) (#3 MITRE) of our hand-crafted dataset, and apply seven trivial augmentations (Table 6) on them and create a total of 84 different code scenarios (12 per augmentation). We choose these two CWEs as (1) they can lead to the most catastrophic impacts like root privilege escalation, and data loss, etc, (2) they belong to two completely different levels of abstractions i.e., “lower-level” (C) and “higher-level” (Python) languages, and (3) the presence of their instances can be determined directly from code without any need for external information.

2) Non-Trivial Augmentations. We design non-trivial code augmentations to perform stress-tests on LLMs to measure their robustness and bias towards semantics of function or variable names, specific library functions, or code security practices. We use combinations of all CWEs defined in Table 4 and the six non-trivial code augmentations from Table 6 and design 66 code scenarios (12 per NT1-NT4 and 9 per NT5 and NT6) 3.

3. See Appendix B for details on creation of non-trivial augmented code scenarios.

### Table 4: Hand-crafted dataset.

<table>
<thead>
<tr>
<th>CWE ID</th>
<th>Description</th>
<th>MITRE Rank</th>
<th>Lang.</th>
</tr>
</thead>
<tbody>
<tr>
<td>787</td>
<td>Out-of-bounds Write</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>2</td>
<td>Py</td>
</tr>
<tr>
<td>89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
<td>3</td>
<td>Py</td>
</tr>
<tr>
<td>416</td>
<td>Use After Free</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
<td>8</td>
<td>C</td>
</tr>
<tr>
<td>476</td>
<td>NULL Pointer Dereference</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>190</td>
<td>Integer Overflow or Wraparound</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>77</td>
<td>Improper Neutralization of Special Elements used in a Command ('Command Injection')</td>
<td>16</td>
<td>C</td>
</tr>
</tbody>
</table>

### Table 5: Real-world CVEs and their details including Original and Truncated Lines of Code (LoC).

<table>
<thead>
<tr>
<th>Project</th>
<th>CVE ID</th>
<th>CWE Description (ID)</th>
<th>Orig. LoC</th>
<th>Trunc. LoC</th>
<th>Pub. Date (2023)</th>
<th>Fix Date (2023)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpac</td>
<td>2023-1452</td>
<td>Out-of-Bound Write (787)</td>
<td>4.5k</td>
<td>243</td>
<td>Mar</td>
<td>May</td>
</tr>
<tr>
<td></td>
<td>2023-3012</td>
<td>NULL Pointer Deref (476)</td>
<td>2.5k</td>
<td>398</td>
<td>May</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-2131</td>
<td>Out-of-Bound Write (787)</td>
<td>12.3k</td>
<td>117</td>
<td>Jan</td>
<td>May</td>
</tr>
<tr>
<td></td>
<td>2023-2134</td>
<td>Integer Overflow (190)</td>
<td>439</td>
<td>389</td>
<td>Jan</td>
<td>May</td>
</tr>
<tr>
<td>libtiff</td>
<td>2023-2908</td>
<td>NULL Pointer Deref (476)</td>
<td>2.3k</td>
<td>629</td>
<td>Jan</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-3316</td>
<td>NULL Pointer Deref (476)</td>
<td>159</td>
<td>159</td>
<td>Jan</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-26966</td>
<td>Out-of-Bound Write (787)</td>
<td>1.8k</td>
<td>238</td>
<td>Jun</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-40745</td>
<td>Integer Overflow (190)</td>
<td>2.2k</td>
<td>757</td>
<td>Oct</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-41175</td>
<td>Integer Overflow (190)</td>
<td>779</td>
<td>748</td>
<td>Oct</td>
<td>Nov</td>
</tr>
<tr>
<td>linux</td>
<td>2023-40283</td>
<td>Use-After-Free (416)</td>
<td>1.9k</td>
<td>515</td>
<td>Aug</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-42753</td>
<td>Integer Overflow (190)</td>
<td>628</td>
<td>623</td>
<td>Sept</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-42754</td>
<td>NULL Pointer Deref (476)</td>
<td>3.7k</td>
<td>177</td>
<td>Oct</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-45863</td>
<td>Out-of-Bound Write (787)</td>
<td>1.1k</td>
<td>565</td>
<td>Oct</td>
<td>Nov</td>
</tr>
<tr>
<td></td>
<td>2023-45871</td>
<td>Out-of-Bound Write (787)</td>
<td>10.1k</td>
<td>386</td>
<td>Oct</td>
<td>Nov</td>
</tr>
<tr>
<td>ppzip</td>
<td>2023-27585</td>
<td>Out-of-Bound Write (787)</td>
<td>784</td>
<td>737</td>
<td>Mar</td>
<td>Aug</td>
</tr>
</tbody>
</table>

2. Appendix A shows examples of all difficulty levels for ‘CWE-22’.

3. See Appendix B for details on creation of non-trivial augmented code scenarios.
Table 6: Code Augmentations.

<table>
<thead>
<tr>
<th>ID</th>
<th>Trivial</th>
<th>Non-Trivial</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Rename function parameters randomly</td>
<td>NT1</td>
</tr>
<tr>
<td>T2</td>
<td>Rename function randomly</td>
<td>NT2</td>
</tr>
<tr>
<td>T3</td>
<td>Add random unreachable code</td>
<td>NT3</td>
</tr>
<tr>
<td>T4</td>
<td>Add random code in comments</td>
<td>NT4</td>
</tr>
<tr>
<td>T5</td>
<td>Insert whitespaces</td>
<td>NT5</td>
</tr>
<tr>
<td>T6</td>
<td>Add a useless function</td>
<td>NT6</td>
</tr>
<tr>
<td>T7</td>
<td>Add next-line character</td>
<td>NT7</td>
</tr>
</tbody>
</table>

3.5. Ground-Truth Reasoning \( G_r \)

In addition to ground truth labels indicating if a code snippet contains a vulnerability, we also need explanations for these vulnerabilities, as we aim to evaluate the reasoning capabilities of LLMs and assess whether they can justify their decisions. To this end, we randomly sample 48 code scenarios out of the 228 total scenarios, and have three security experts, including the first author of the paper equally divide the sampled code scenarios amongst each other and create a 100-word ground truth reason for each scenario using MITRE’s official CWE documentation [41] as a guide. The experts then compare and discuss each others’ reasoning and develop consensus for each ground-truth reasoning (Fleiss’ kappa with \( K = 0.93 \), meaning almost perfect agreement [46]). After establishing that the criteria for ground truth reasoning are well laid out and understood, the first author proceeds to develop the remaining 180 vulnerability explanations. This ground truth reasoning \( G_r \) is then used by the ‘Evaluator’ module in the next step.

3.6. Evaluator

The output of an LLM for a specific test is passed to the Evaluator (see Figure 2). As SecLLMHolmes is fully automated, we leverage GPT-4 to analyze the response. First, the response is passed to GPT-4, with an additional role-based instruction prompt \( P_r \) (shown in Figure 8 in the Appendix) in the ‘system’ input to extract two pieces of information from the raw response. The first one is the binary answer, which is ‘yes/no’ based on whether the LLM found a vulnerability in the given code or not. We find that in some cases the LLM does not provide a definite answer, therefore we include a third verdict, i.e., ‘n/a.’

Provided by the LLM \( (P_r) \), to extract it, we ask GPT-4 to summarize the reason described by the model output on why a vulnerability is present in the code or not, in 100 words. We extract the summary of root cause of vulnerability from raw responses of LLMs to maintain consistency for further evaluation methods, and to avoid contents like suggestions, fixed code, etc. to be analyzed. In the rest of this section we provide more details on the evaluation metrics used to evaluate the final answer, and its reasoning provided by LLM for the vulnerability detection task.

Accuracy Score. To evaluate the accurate detection of a vulnerability in source code, we compare the answer extracted from the response of the LLM (i.e., “yes/no/n/a”) with the ground truth labels and use the “accuracy” metric, i.e., if LLM’s answer (binary) matches the ground-truth or not, to assess the correctness of the final answer.

Reasoning Score. Automatically evaluating if the textual reasoning provided by the LLM on whether a vulnerability is present is a challenging task. To solve this task we use GPT-4 to summarize the reason \( (P_r) \) provided by an LLM and compare it with the ground-truth reasoning \( (G_r) \) generated by the authors, using the following three metrics:

1) Rouge [47] score is traditionally used in NLP to measure the similarity between a machine-translated summary and reference summaries using overlapping n-grams. In this case, we use it to measure similarity between the summaries \( P_r \) and \( G_r \). We first sample 50 pairs of \( P_r \) and their corresponding \( G_r \), as our reasoning score validation set (\( R_{val} \)), and manually check the consistency and alignment of their reasonings. We find that at the optimal threshold \( \text{Rouge}_{\text{thres}} \) of 0.34, 43 out of 50 \( P_r \) are consistent with \( G_r \). We therefore use this threshold and mark two summaries as similar if their Rouge score exceeds it.

2) Cosine Similarity is a metric commonly used in NLP to measure how similar two documents are irrespective of their sizes. We first convert the summaries \( P_r \) and \( G_r \) into fixed length vectors using OpenAI’s embedding model ‘text-similarity-davinci-001’ and calculate the cosine similarity between them. Similar to \( \text{Rouge}_{\text{thres}} \), we find that the optimal threshold \( \text{Cos}_{\text{thres}} \) is 0.84, and consider two summaries similar if their cosine similarity exceeds this value.

3) GPT-4 is prompted to evaluate if the reasoning in \( P_r \) and \( G_r \) align. If the reasonings are similar and they align with each other, GPT-4 responds ‘yes’ and we assign a reasoning score of 1, otherwise we assign 0. We find that GPT-4 successfully classifies 48 out of 50 \( P_r \) correctly to their corresponding \( G_r \), when validated on \( R_{val} \).

We then determine whether the reasoning by the LLM is correct or not by majority vote. That is, if two or more of the above criteria match, we consider the reasoning as similar to the ground truth reasoning \( G_r \).
4. Experimental Investigation

In this section, we use our framework to investigate all LLMs listed in Table 1. We first investigate which values of the LLM parameters are most likely to produce consistent (Section 4.1) and best performing (Section 4.2) output. We then perform the rest of our investigations using the most suitable parameter values.

4.1. Evaluation for Deterministic Responses

To perform a rigorous comparison between LLMs and assess their capabilities, it is of critical importance that their responses are consistent, meaning that running the same test multiple times under identical parameters should provide the same final verdict. Therefore, we first investigate whether this consistency is achievable at all, and what LLM parameters deliver the most consistent results. OpenAI’s documentation [48] recommends a temperature of 0.2 and ‘top_p’ of 0.1 to achieve the most deterministic output for code related tasks. Similarly, the recommended ‘temperature’ value for all LLMs in our evaluation is 0.2. When experimenting with modifying these values, both the OpenAI documentation [49] and previous research [7] recommend to keep the value of ‘top_p’ constant and modify the value of ‘temperature’. We therefore fix ‘top_p’ to default value specific to an LLM, and perform experiments using two different ‘temperature’ values: 0.2 (‘default’) and 0.0. We perform this experiment on two vulnerable and two patched medium code difficulty level scenarios (2\textsubscript{v} and 2\textsubscript{p}) from two distinct vulnerabilities, “out-of-bound write” (CWE-787) in C and “SQL injection” (CWE-89) in Python (for the same reasons as discussed in Section 3.4). For the input prompts we select the set of Standard prompts (see Section 3.3). We run each experiment ten times, and record how many times the model provides the same answer. We consider a model to be consistent if it always provides the same binary answer, regardless of whether it is correct.

Observations. Table 7 shows that all LLMs provide inconsistent responses for one or more of the tests at the recommended ‘temperature’ value of 0.2. ‘codechat-bison@001’ even provides a wrong answer with the most basic ‘S1’ prompt (as shown in Figure 3). This suggests that the default ‘temperature’ is not a good choice to evaluate LLMs for vulnerability detection. Using 0.0 as temperature improves consistency, as shown in Table 8: ‘codechat-bison@001,’ ‘codellama34@$\text{b}$,’ and ‘gpt-3.5-turbo-16k’ provide consistent responses for all tests at this temperature. However, two LLMs (‘chat-bison@001’ and ‘gpt-4’) still provide inconsistent results. Based on these results, we find that 0.0 is the best ‘temperature’ value to get consistent responses from an LLM, although we note that even at this setting some LLMs fail in delivering consistent responses.

6. We only report the results for the five best performing LLMs in the main body of the paper. The ones for the remaining three LLMs can be found in the Appendix.

<table>
<thead>
<tr>
<th>Models</th>
<th>S\textsubscript{1}</th>
<th>S\textsubscript{2}</th>
<th>S\textsubscript{3}</th>
<th>S\textsubscript{4}</th>
<th>S\textsubscript{5}</th>
<th>S\textsubscript{6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>chat-bison</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
</tr>
<tr>
<td>codechat-bison</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
</tr>
<tr>
<td>codellama34@$\text{b}$</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
</tr>
<tr>
<td>gpt-3.5</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
</tr>
<tr>
<td>gpt-4</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
<td>10/10</td>
</tr>
</tbody>
</table>

4.2. Performance Over Range of Parameters

While lower temperatures increase the consistency in results, setting a higher temperature is supposed to increase the creativity in LLMs. In this section, we aim to investigate whether increasing the temperature for LLMs increases their performance in identifying vulnerabilities, both with respect to their accuracy and the correctness of their reasoning.

For this experiment, we select the same two classes of security weaknesses used in the previous section, and select two vulnerable and two patched code scenarios from these classes. We choose scenarios with the highest code difficulty level (3\textsubscript{v} and 3\textsubscript{p}) as they would need more creativity to be correctly solved by LLMs.

As running experiments on all prompts is not feasible due to budget constraints, we perform the experiment on only one prompt, S4 (ZS-RO), based on three reasons: (1) this prompt does not provide any additional information like definition or step-by-step reasoning instructions to the LLM, so the response will be mainly based on the intrinsic
... MAX_SIZE = 32;
char* encode_input(char *user_supplied_string) {
  int i, dst_index;
  char* dst_buf = (char*)malloc((4 * sizeof(char) * MAX_SIZE) + 1);
  if (MAX_SIZE <= strlen(user_supplied_string)) {
    exit(1);
  }
  dst_index = 0;
  for (i = 0; i < strlen(user_supplied_string); i++) {
    if (*user_supplied_string[i] == 'p') {
      dst_buf[dst_index++] = 'p';
    }
    if (*user_supplied_string[i] == 'm') {
      dst_buf[dst_index++] = 'm';
    }
    if (*user_supplied_string[i] == 'a') {
      dst_buf[dst_index++] = 'a';
    }
    if (*user_supplied_string[i] == 't') {
      dst_buf[dst_index++] = 't';
    }
  }
  return dst_buf;
}

(a) CWE-787 (Out-of-Bound Write) 2:
This code scenario encodes certain characters in the user input string. The program assumes that encoding expansion will only expand a given character by a factor of 4, however the amperand encoding expands by 5. If the attacker provides a string of many ampersands, the string will overflow the destination buffer.

Yes, the code contains instances of the security vulnerability known as out-of-bounds write. The code allocates a buffer of size (4 * sizeof(char) * MAX_SIZE) + 1 bytes. However, the code does not check to ensure that the length of the input string does not exceed the size of the buffer. This could lead to an out-of-bounds write if the input string is longer than the buffer.

No, the code does not contain instances of the security vulnerability known as out-of-bounds write. The code is safe because the 'dst_index' variable is used to keep track of the current position in the destination buffer. This variable is incremented each time a character is copied from the source buffer to the destination buffer. This ensures that no characters are written outside of the bounds of the destination buffer.

(b) 'codechat-bison@001' responses.

Figure 3: Non-deterministic and inconsistent responses by 'codechat-bison@001'.

knowledge of the model from its training data and no external instruction or information will influence its reasoning or creativity at a higher temperature, (2) being a zero-shot prompt it does not limit the creativity or reasoning of the model as in few-shot prompting might influence the model to mimic the few-shot examples, and (3) this prompt is the most non-deterministic zero-shot prompt as shown in Tables 8 and 7, allowing it to show more randomness or creativity at higher temperature. We evaluate LLMs on six temperature values: their recommended value 0.2, and over a range of values from 0 to 1 i.e., 0, 0.25, 0.5, 0.75, 1.0. We run each experiment ten times, and show how many times an LLM provides a correct answer (i.e., accuracy) and correct reasoning (i.e., reasoning score). The results are summarized

<table>
<thead>
<tr>
<th>Table 9: Evaluation of LLMs Over a Range of Temperature Values (CWE-787). The table shows results for each temperature value in the format of # correct / # total answered out of 10.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>chat-bison</td>
</tr>
<tr>
<td>codechat-bison</td>
</tr>
<tr>
<td>gpt-3.5</td>
</tr>
<tr>
<td>gpt-4</td>
</tr>
</tbody>
</table>

(a) Accuracy (3p)

(b) Reason (3p)

(c) Accuracy (3p)

(d) Reason (3p)

in Tables 9 and 10.

Observations. Our results do not show a general trend of better performance with the increase in model temperature. Since increasing the temperature does not present a general improvement of results across our models, to prioritize result consistency we elected to use 0.0 as the ‘temperature’ value for the remaining of our experiments, and set ‘top_p’ to LLM specific default value.

4.3. Diversity of Prompts

In this part of investigation, we test the LLMs on their ability to detect vulnerabilities in the 48 hand-crafted code scenarios described in Section 3.4, by using the 17 prompts ranging over three categories and four prompting techniques, as described in Section 3.3. This experiment allows us to evaluate the capabilities of LLMs over a wide input spectrum and answer questions like (1) what kind of prompting techniques work best for the model, (2) whether emulating the multi-step reasoning process followed by human security experts improves performance, and (3) whether providing
extra information or examples helps LLMs in decision making? Table 11 shows the results of this experiment based on the following three metrics:

(1) **Response Rate**: Measures how often the model provides an answer to a given input at all. E.g., for prompts ‘S5’ and ‘S6,’ ‘codechat-bison@001’ provides correct answers to 36 out of 48 inputs and for the rest it responds ‘I’m not able to help with that, as I’m only a language model. If you believe this is an error, please send us your feedback.’

\[
\text{ResponseRate} = \frac{\#\text{InputsAnswered}}{\text{TotalInputs}}
\]

(2) **Accuracy Rate**: Measures the correctness of the model’s response, regardless of the provided reasoning. E.g., for prompt ‘D2,’ ‘codechat-bison@001’ provides correct answers to 24 inputs out of the 48 answered inputs.

\[
\text{AccuracyRate} = \frac{\#\text{CorrectAnswers}}{\#\text{InputsAnswered}}
\]

(3) **Correct Reasoning Rate (CRR)**: Evaluates how often the model’s correct answers also have the correct reasoning. E.g., for prompt ‘D2,’ ‘codechat-bison@001’ provides reasoning for 15 answers out of the 24 correct answers and out of those 15 reasonings 14 are correct.

\[
\text{CRR} = \frac{\#\text{ReasoningswithCorrectAnswers}}{\#\text{ReasoningswithCorrectReasoning}}
\]

**Best Prompts**: Based on the above three metrics, we choose the best overall prompts for each model from four categories of prompts (described in Section 3.3) i.e., ZS - TO, ZS - RO, FS - TO, and FS - RO. We calculate a Scoreprompt, which is the weighted sum of the three metrics where each metric is assigned an equal weight of 0.33. We then select the best prompt from each of the above defined four categories as the one that maximizes Scoreprompt, as shown in Table 11.

**Observations**. ‘gpt-4’ performs the best among the tested LLMs, with a maximum accuracy of 89.5%. There is no prompt for which all LLMs perform consistently better, but instead they show different success for different types of prompts. GPT models and ‘codechat-bison’ perform better when prompted to follow a human-like step-by-step reasoning process (as shown in Figure 4), i.e., R4, R6, and R2 prompts, respectively. ‘chat-bison’ performs best when assigned a ‘security expert’ role, while ‘codellama34b’ works best with the S1 prompt, which simply asks if the code snippet contains a certain vulnerability. While ‘gpt-4’ and ‘codellama34b’ show an increase in accuracy when provided with a vulnerability definition, compared to standard prompts, the same trend is not found in the other LLMs.

### 4.4. Faithful Reasoning

Faithful reasoning is the quality of an LLM to provide the right reasoning for the right answer or vice versa [50]. The more faithful an LLM’s reasoning is to its final answer, the more a user can trust its response. Table 11 shows that even when they provide the right response, LLMs sometimes provide the wrong reason for this decision. In this section, we further analyze the faithful reasoning of LLMs on their decisions from the experiment discussed in the previous section, focusing on five aspects: (1) for how many cases does the LLM provide a reasoning for the presence of a vulnerability in a code snippet at all, (2) for how many correct answers the LLM also provides a correct reasoning, (3) for how many correct answers the LLM provides an incorrect reasoning, (4) for how many incorrect answers the LLM provides an incorrect answer, and (5) for how many cases the LLM provides the wrong answer, but a correct reasoning. Table 12 provides an overview of the results of this experiment.

**Observations**. While in the vast majority of cases the answer and reasoning for the tested LLMs align, every LLM presents cases where it provides the correct reasoning but this leads to a wrong answer (as shown in Figure 5). Similarly, we find cases where an LLM provides a correct reasoning, but this leads to a wrong answer (as shown in Figure 5b). We also find that Google’s PaLM2 models have overall lower reasoning rate as they do not explain their decisions in detail.
... void read_file(char* file_path) {
    char* resolved_path = realpath(file_path, NULL);
    if (resolved_path == NULL) {
        printf("Error: Invalid Path\n");
        return;
    }
    char* dir = "/Users/user1/";
    /* safely creates full_path by concatenating file_path with dir */
    FILE* fp = fopen(full_path, "r");
    char* data = malloc(256);
    while (fgets(data, 256, fp) != NULL) {
        /* processes file */
    }
}

(a) CWE-476 (null pointer dereference) 1:
This code scenario sanitizes the user provided file path, opens file at that file path, and processes it. The code does not check for the NULL value of fopen in case an error is raised.

(b) GPT-3.5 Responses.
Figure 4: ‘GPT-3.5’ responses to standard ‘S1’ and security-expert like multi-step reasoning ‘R2’ for CWE-476 1v code scenario.

... many cases, while GPT models show comparatively higher reasoning rates, especially ‘gpt-4’ and ‘codellama34b’ provide a reason for every answer. Our findings suggest that, in certain cases, current LLMs’ responses might not fully rely on faithful and accurate reasoning.

4.5. Evaluation Over Variety of Vulnerabilities
In this section, we focus on analyzing LLMs ability to correctly identify both vulnerable and patched code for different types of vulnerabilities, based on the eight CWEs that we used to build our hand-crafted dataset. Similar to Section 4.3, we find and use the best performing prompts for each CWE using Score_{cwe}, with equal weight to all factors, from four prompting categories. The results are summarized in Table 13.

![Image](https://example.com/figure5)

**Figure 5:** ‘chat-bison@001’ (PaLM2) response for CWE-77 3c scenario (see Appendix Figure 11) using prompt ‘D3’ shows unfaithfulness between provided reasoning and final answer.

**Observations.** Most models show poor performance in classifying the patched versions correctly, which makes these LLMs non-suited for real-world cases as they will mostly flag safe code as vulnerable, causing many false alarms. We observe that few-shot prompting performs significantly better than zero-shot prompting for almost all models (p-value = 0.003), and role-oriented prompts perform slightly better than task-oriented prompts (p-value = 0.1). The reason for this is that assigning a role to the model grounds its knowledge for the given task and prevents it from hallucinating, which can be seen in the increase in reasoning score for role-oriented prompts. However, Table 13 shows that ‘codechat-bison@001’ does not provide answers for ‘CWE-787’ and ‘CWE-416’ for few-shot prompts.

4.6. Code Difficulty Levels
In this section, we investigate the capabilities of LLMs to handle different complexities of code. Similar to the previous sections, we find the best performing prompts for each difficulty level using Score_{diff}, with equal weight to all factors, from four prompting categories. Table 14 summarizes the results of this experiment.

**Observations.** LLMs generally perform better on the easy code scenarios, with limited exceptions (e.g., ‘codechat-
bison’ correctly recognizing more medium than easy vulnerabilities in the FS - TO setting). By manually looking at the misclassified scenarios, we make two observations; (1) LLMs are not familiar with the safe practices of library functions, and (2) LLMs cannot handle complex multifunction and multi-variable data flow patterns. For example, in the CWE-89 (SQL Injection) ‘3p’ scenario, the code creates an SQL query and its parameters in one function ‘create_query’, and returns them to the ‘login’ function where they are passed to ‘pymysql’ as separate arguments. None of the LLMs were able to understand this flow (e.g., the response from ‘gpt-4’ is shown in Figure 6), as they mistakenly assume that the ‘login’ function only passes one argument to the `execute` function. The LLMs also seem to be unaware that ‘pymysql’ itself sanitizes the output.

4.7. Robustness to Code Augmentations

In this section we test the robustness of LLMs by testing them against the code augmentations described in Section 3.4. Our results are summarized in Table 6. For each input augmentation we show the change in accuracy and reasoning score as compared to the original non-augmented version of the input. For each LLM, we test each augmentation using three prompts: standard prompt ‘S1’, and the best zero-shot (ZS) and few-shot (FS) prompts.

7. Since we show in Sections 4.5 and 4.6 that role-oriented prompts work better than task-oriented ones, we do not run experiments on all four categories of prompts.

TABLE 13: Evaluation of LLMs over a wide range of eight most critical vulnerabilities. Each bar represents count of correctly classified vulnerable (black bar) and patched (gray bar) code scenarios, and each circle marks count of correctly reasoned vulnerable (white circle) and patched (black circle) code scenarios, out of total answered scenarios by each LLM.

<table>
<thead>
<tr>
<th>Model</th>
<th>ZS - TO</th>
<th>ZS - BO</th>
<th>FS - TO</th>
<th>FS - BO</th>
</tr>
</thead>
<tbody>
<tr>
<td>chat-bison S1</td>
<td>7/16</td>
<td>R2 10/16</td>
<td>R6 14/16</td>
<td>S6 16/16</td>
</tr>
<tr>
<td>codechat-bison S1</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
</tr>
<tr>
<td>codellama34b S1</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
</tr>
<tr>
<td>gpt-3.5 S1</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
</tr>
<tr>
<td>gpt-4 S1</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
<td>0/16</td>
</tr>
</tbody>
</table>

TABLE 14: Evaluation of LLMs over a Range of Code Difficulty Levels. (Same color coding as Table 13).

<table>
<thead>
<tr>
<th>Model</th>
<th>ZS - TO</th>
<th>ZS - BO</th>
<th>FS - TO</th>
<th>FS - BO</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpt-4 R1</td>
<td>12/16</td>
<td>R2 12/16</td>
<td>R6 14/16</td>
<td>D3 13/16</td>
</tr>
<tr>
<td>gpt-3.5 S1</td>
<td>10/16</td>
<td>S5 8/16</td>
<td>S6 9/16</td>
<td>D3 11/16</td>
</tr>
<tr>
<td>gpt-4 D1</td>
<td>9/16</td>
<td>R2 11/16</td>
<td>R6 10/16</td>
<td>R4 10/16</td>
</tr>
<tr>
<td>gpt-3.5 D1</td>
<td>8/16</td>
<td>R1 10/16</td>
<td>R6 10/16</td>
<td>R4 10/16</td>
</tr>
</tbody>
</table>

Figure 6: GPT-4 response. (a) CWE-89 (SQL Injection) ‘3p’ code scenario. This code is not vulnerable to SQL Injection as it passes the ‘query’ and ‘parameters’ separately to the ‘pymysql’ function. (b) GPT-4 response.
TABLE 15: Evaluation for Code-Level Augmentations. The tables show $\Delta_1$ (# of answers that are correct in non-augmented scenarios but incorrect in this specific augmentation case) and $\Delta_2$ (# of reasoning that are correct in non-augmented scenarios but incorrect in this specific augmentation case) for each code augmentation and for three prompts (standard ‘S1’, best zero-shot, and best few-shot) of each LLM.

<table>
<thead>
<tr>
<th>M</th>
<th>P9</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\Delta_1$</td>
<td>$\Delta_2$</td>
<td>$\Delta_3$</td>
<td>$\Delta_4$</td>
<td>$\Delta_5$</td>
<td>$\Delta_6$</td>
<td>$\Delta_7$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0/1</td>
<td>1/0</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>5</td>
<td>0/1</td>
<td>1/0</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>5</td>
<td>0/1</td>
<td>1/0</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>

**Observations.** Table 15a shows that even trivial augmentations like the addition of whitespaces (Figure 7a) and newline characters lead all LLMs to an incorrect answer and reasoning in some cases, and further breaks their chain-of-thought reasoning. Furthermore, changing function or variable names or the presence of unreachable code lead to incorrect answers. When looking at non-trivial augmentations, Table 15b shows that LLM performance is also affected by function and variable names, e.g., changing a variable name to ‘buffer’ in NT1 leads to the wrong detection of a buffer overflow and changing a function name to ‘non_vulnerable’ or to a safe function name increases the chances to be detected as non-vulnerable. Most importantly, LLMs present a bias towards library functions that are usually used for sanitization or are considered potentially vulnerable. E.g., all LLMs would declare the safe usage of ‘strcat’ in C as vulnerable, and unsafe uses of ‘strcat’ would be flagged as safe (Figure 7b). Similarly, the unsafe use of sanitizing library functions like ‘reapath’ in C or ‘escape’ in Python is perceived as vulnerable.

---

(a) Example of a complete change in ‘chat-bison’ decision by just adding whitespaces (T5) in code scenario CVE-787 ²₉ (shown in Figure 3a) after line 22.

(b) Example of NTS augmentation to CVE-787 ²₉, where even the addition of safely used ‘strcat’ successfully confuses gpt-4 to classify the code as vulnerable merely on the basis of the presence of ‘strcat’. However, the addition of ‘strncat’ leads to the classification of vulnerable code as safe.

(c) Example of NTS augmentation to CVE-79.2, where even an addition of an unsafe ‘escape’ function call makes ‘codellama34b’ believe that the code is safe.

---

4.8. Real-World Cases

Finally, we investigate the ability of LLMs to identify real-world vulnerable code, by leveraging our CVE dataset (see Table 5) using the best prompts listed in Table 11. The results are summarized in Tables 16 and 17.

**Observations.** Overall, the evaluation with our real-world CVEs samples highlights that LLMs face challenges in detecting vulnerabilities in real-world projects, with all studied LLMs providing incorrect answers for several of our test cases. In addition to providing wrong answers for vulnerable code, LLMs frequently mistakenly identify patched examples as vulnerable, which would be particularly problematic if these models were used in production, as it would make the number of false positives skyrocket. We also observe that few-shot prompting does not work in case of real-world scenarios, likely because LLMs fail to extrapolate (Figure 7c) are detected as non-vulnerable. Our experiments show that there is no prompting technique that is completely robust as our robustness tests break even the best types of prompting techniques and chain-of-thought for all LLMs, leading to incorrect responses (17% of cases for GPT-4).
TABLE 16: Evaluation on real-world CVEs for Linux and pjsip. This table shows results for both vulnerable and patched versions of every CVE, given by the best prompts of every LLM. ✓ (no answer), ✗ (correct answer but wrong reasoning), ○ (correct answer with correct reasoning), × (wrong answer and no reasoning), and ✗ (wrong answer and no or wrong reasoning).

TABLE 17: Evaluation on Real-World CVEs for gpac and libtiff.

5. Discussion

SecLLMHolmes allows users to evaluate any chat-based LLM for its ability to identify software vulnerabilities. Further, we have publicly released our framework, enabling the community to evaluate LLMs released in the future. For example, researchers will be able to compare performance between different releases of an LLM, or study if changing properties like model architecture or number of parameters improves their ability to detect vulnerabilities. 8

8. We include these case studies for ‘codellama7B’, ‘codellama13B’, and ‘starchat-beta’ in tables in Appendix.

AI companies are taking steps to address some of the issues highlighted in this work. For example, the latest release of ‘GPT-4 Turbo (preview)’ introduces the use of a ‘seed’ during inference to enable deterministic output. While a step in the right direction to ensure reliable output, this approach still presents the problem that different seeds might produce different answers for the same input.

Limitations. As any research project, our work presents some limitations. In the following, we discuss them in detail.

Answer and Reason Extraction. We use GPT-4 to parse the LLM output and extract the final answer and reasoning, using a prompt that requires GPT-4 to answer in a given format (as shown in Figure 8 in the Appendix), otherwise further steps of our analysis would fail. We manually analyze 100
6. Conclusion

This work presents the first scalable and fully automated framework to evaluate the efficiency and reasoning capabilities of chat-based LLMs across eight distinct dimensions for the task of vulnerability detection. We performed an evaluation of state-of-the-art LLMs using this framework, showing that they are currently unreliable at this task and will answer wrongly when asked to identify vulnerabilities in source code. Based on these results, we conclude that state-of-the-art LLMs are not yet ready to be used for vulnerability detection and urge future research to address and resolve the highlighted issues. Our framework and benchmarks will be a useful tool for the community to evaluate the progress of future LLM versions in vulnerability detection.

Acknowledgments

We would like to thank Syed Qasim and Pujan Paudel for their help in generating-ground truth reasoning for the code scenarios. This work was supported by the Red Hat Collaboratory and by the NSF under grants CNS-1942610 and CNS-2127232. Any opinions, findings, and conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

References


9. We note that using the newest ‘GPT-4 Turbo,’ which provides responses in json format, could eliminate even these two anomalies.


### Appendix A.

#### Examples of Code Difficulty Levels

**Easy:** CWE-22 `‘_v’` (see Figure 9a) takes a file path as an input, which is then concatenated with an absolute directory path, and then the file is read and displayed on the console. The file path provided by the user is not sanitized, leading to a directory traversal vulnerability.

**Medium:** CWE-22 `‘v’` (see Figure 9b) takes four inputs: file path, flag, data, and directory path (set using an environment variable). Based on the flag, data and file are processed. The program also calls the `‘realpath’` function to sanitize the input, but only applies it to the directory path, leaving the file path vulnerable to directory traversal.

**Hard:** CWE-22 `‘3’` (see Figure 9c) contains two functions: `resolve_path` that takes in a path and replaces all white
spaces with hyphens, and 'print_file' that takes in the directory path from an environment variable and the file path from the user, concatenates them and calls 'resolve_path' on it. The 'resolve_path' function call in 'print_file' semantically appears to be sanitizing the given path, but actually it fails in doing so.

Appendix B.

Robustness to Code Augmentations

A1: In this case, we design 12 CWE scenarios for the top two vulnerabilities in C i.e., CWE-787 (out-of-bound write) and CWE-416 (use-after-free) and replace the name of the variables with 'buffer' and ask the the model to analyze the code for 'out-of-bound write / buffer overflow'.

A2: In this case, we select 12 patched versions of CWE-scenarios from the four most dangerous classes of vulnerabilities i.e., CWE-787 (C), CWE-79 (Py), CWE-89 (Py), and CWE-416 (C), and change the names of functions to 'vulnerable_func'.

A3: Opposite to ‘A2’, in this case, we select 12 vulnerable versions of CWE-scenarios from the same four classes of vulnerabilities and change the names of functions to 'non_vulnerable_func'.

A4: In this case, we safely add 'strncpy' and 'strcat' library functions, which are famous to be vulnerable but these are only vulnerable if they are not used properly, into 12 manually crafted code scenarios of CWE-787 and CWE-416.

A5: Opposite to ‘A4’, we add library functions which are commonly used to sanitize inputs. We add ‘realpath’, which is used to sanitize input file path and prevents path traversal attack, to three vulnerable CWE-22 scenarios. For python, we add ‘escape’, which is used to sanitize the input for any scripts and prevents cross-site scripting attack, to three vulnerable CWE-79 scenarios. Finally we add ‘strcat’ and ‘strncpy’, which are considered safer functions to prevent out-of-bound write, to three vulnerable CWE-787 scenarios.

A6: In this case, we add respective '#define' expressions (as shown in figure 10) to 9 CWE scenarios from CWE-77, CWE-22, and CWE-787.
TABLE 22: Evaluation Diversity of Prompts (Ext. Table 11).

<table>
<thead>
<tr>
<th>Model</th>
<th>Reas. Rate</th>
<th>Details</th>
<th>M Standard</th>
<th>Step-by-Step</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>3/6</td>
<td>R1</td>
<td>24/48</td>
<td>D1</td>
<td>24/48</td>
</tr>
<tr>
<td>S2</td>
<td>3/6</td>
<td>R2</td>
<td>24/48</td>
<td>D2</td>
<td>24/48</td>
</tr>
<tr>
<td>S3</td>
<td>3/6</td>
<td>R3</td>
<td>24/48</td>
<td>D3</td>
<td>24/48</td>
</tr>
<tr>
<td>S4</td>
<td>3/6</td>
<td>R4</td>
<td>24/48</td>
<td>D4</td>
<td>24/48</td>
</tr>
<tr>
<td>S5</td>
<td>3/6</td>
<td>R5</td>
<td>24/48</td>
<td>D5</td>
<td>24/48</td>
</tr>
<tr>
<td>S6</td>
<td>3/6</td>
<td>R6</td>
<td>24/48</td>
<td>D6</td>
<td>24/48</td>
</tr>
</tbody>
</table>

TABLE 23: Faithfulness of LLMs (Ext. Table 12).

<table>
<thead>
<tr>
<th>Model</th>
<th>Reas. Rate</th>
<th>Details</th>
<th>M Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>codellama7b</td>
<td>814 /815</td>
<td>S1</td>
<td>381</td>
</tr>
<tr>
<td>codellama13b</td>
<td>815 /816</td>
<td>S1</td>
<td>391</td>
</tr>
<tr>
<td>starchat</td>
<td>809 /811</td>
<td>S1</td>
<td>398</td>
</tr>
</tbody>
</table>

TABLE 24: Evaluation range of vulnerabilities (Ext. Table 13).

<table>
<thead>
<tr>
<th>Model</th>
<th>ZS - TO</th>
<th>ZS - RO</th>
<th>FS - TO</th>
<th>FS - RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>codellama7b</td>
<td>S1</td>
<td>4/6</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>codellama7b</td>
<td>S1</td>
<td>4/6</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>starchat</td>
<td>S1</td>
<td>4/6</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>codellama13b</td>
<td>S1</td>
<td>4/6</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>starchat</td>
<td>S1</td>
<td>4/6</td>
<td>4/6</td>
<td>4/6</td>
</tr>
</tbody>
</table>

(a) CWE-22 (1a) Easy Code Difficulty Level

```c
... void read_file(char* file_name) {
    char* dir = "/Users/user1/";
    int file_path_len = strlen(dir) + strlen(file_name) + 1;
    char* file_path = malloc(file_path_len);
    ...
    strncpy(file_path, dir);
    strcat(file_path, file_name);
    FILE* f = fopen(file_path, "r");
    ...
    /* read file */
}
```

(b) CWE-22 (2a) Medium Code Difficulty Level

```c
... void resolve_path(char* path) {
    char* p = path;
    while (*p != '\0') {
        if (*p == '/') {
            *p = '-';
        }
        p++;
    }
}
```

(c) CWE-22 (3a) Hard Code Difficulty Level

```c
... void print_file(char* file_name) {
    char* dir = getenv("dir");
    ...
    int path_len = strlen(dir) + strlen(file_name) + 1;
    char* path = malloc(path_len);
    ...
    strncpy(path, dir);
    strcat(path, file_name);
    resolve_path(path);
    FILE* f = fopen(path, "r");
    ...
}
```

Figure 9: Examples of Different Difficulty Levels
#define FGETS(buffer, buffer_len, stdin) gets(buffer)
#define SAFE_EXECUTION(cmd) system(cmd)
#define STRNCPY(dest, src, n) strncpy(dest, src, n)

Figure 10: #define expressions for ‘NT6’ code augmentations

... char* validate_cmd(char* cmd) {
    cmd_ix = 0;
    while (cmd_ix < strlen(cmd)) {
        if (cmd[cmd_ix] == ';') {
            break;
        }
        new_cmd[cmd_ix] = cmd[cmd_ix];
        cmd_ix++;
    }
    return new_cmd;
}
...

Figure 11: CWE-77 (Command Injection) 3a code scenario.

TABLE 25: Evaluation code difficulties (Ext. Table 14).

TABLE 26: Evaluation Code-Level Augmentations (Ext. Table 15).

TABLE 27: Evaluation real-world CVEs (Ext. Tables 16 and 17).
Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper suggests an automated framework for evaluating LLMs (Large Language Models) for identifying and reasoning about security vulnerabilities. The authors conducted experiments on various options of LLMs (e.g., prompts, models, test data) and presented various interesting findings.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Provides a New Data Set For Public Use

C.3. Reasons for Acceptance

1) This paper suggests a fully automated framework to evaluate various aspects of LLM's capability in identifying vulnerabilities, which is gaining more attention from the community.
2) The paper examines several aspects that have been less studied previously, including code complexity, vulnerability reasoning, and use of chat-based LLMs (e.g., GPT-4) in vulnerability detection.
3) The authors have developed a new benchmark consisting of 228 code scenarios. This includes 48 hand-crafted examples (starting from most critical CWEs), 30 examples from real-world CVEs (taken from open source projects) and 150 codes obtained by trivial and non-trivial augmentation. If released, this dataset can help in carrying out further research, allowing an alignment for future analysis.

C.4. Noteworthy Concerns

1) Some findings seem well-known to the community (e.g., non-determinism, and non-robustness by specific code augmentations).
2) The size of the data set is still limited, especially for the real-world CVEs (the data set only includes 15 CVEs now). Moreover, the dataset includes many augmented codes (150/228). The small data set may lead to significant bias in the measurement results, making it less reliable.

Appendix D.
Response to the Meta-Review

We thank the reviewers for their valuable feedback and the shepherd for helping us improve the paper. We would like to respond to the meta-review concerns as follows:

1) While instances of non-determinism and non-robustness to specific code augmentations have been found in previous work, to the best of our knowledge we are the first ones to systematically evaluate these issues in the context of vulnerability detection.
2) The size of our dataset is comparable to previous work like [8] which only used 12 CVEs. In our study, the number of suitable CVEs was constrained by the knowledge cut-off date of LLMs (to maintain the temporal consistency of our evaluation [30]), by the CWE types in our evaluation study, and by needing CVEs with high-quality patch commits.